

SATISFIABILITATEA FORMULELOR PROPOZIȚIONALE

Anghel Costel Junior, Covercă Elena-Daniela, Coroian Dan-Marius, and
Bogdan Simina

West University of Timisoara

Rezumat Această lucrare analizează implementarea și performanța MiniSat, un solver SAT pentru determinarea satisfiabilității formulelor booleene. Studiul se concentrează pe implementarea algoritmilor DPLL și CDCL, împreună cu structurile de date necesare. Evaluarea a fost realizată pe două familii de benchmark-uri, acestea fiind: tseitin-formulas și equivalence-chain-principle, cu timpi de execuție diferiți, contribuind la înțelegerea unui solver SAT.

Keywords: Satisfiability · MiniSat · Benchmarks

1 Introducere

1.1 Context

Problema satisfiabilității (SAT) urmărește să determine dacă variabilele unei formule booleene pot primi valori astfel încât formula să fie adevărată. Această problemă are multiple aplicații practice în verificarea formală și optimizarea sistemelor.

Minisat, care este folosit pentru a rezolva probleme de satisfiabilitate booleană, a câștigat mai multe competiții SAT și este utilizat pe scară largă. Acesta s-a dovedit capabil să gestioneze probleme complexe ce implică foarte multe variabile [7]. Arhitectura sa, dezvoltată de Niklas Eén și Niklas Sörensson, reprezintă un punct de referință pentru implementarea soluțiilor SAT. Diagrama UML a MiniSat reprezintă structura solver-ului, așa cum se poate observa în Figura 1.

MiniSat este, în mare parte, ușor de utilizat. Utilizatorul oferă solver-ului un fișier ce specifică clauzele în forma normală conjunctivă (CNF), iar acesta afișează informații despre satisfiabilitate: fie indică faptul că formula este satisfiabilă, împreună cu o atribuire a variabilelor care o satisface, fie indică faptul că este nesatisfiabilă, când formula nu poate fi satisfăcută. De asemenea, solver-ul afișează informații despre procesul intern, cum ar fi aspecte ale căutării (DPLL). MiniSat folosește două tehnici principale: propagarea unităților, care determină rapid consecințele alegerilor făcute, și învățarea din conflicte, care ajută la evitarea repetării aceluiași greșeli [6].

Prin analiza implementării acestuia, putem înțelege mai bine cum funcționează un solver SAT și cum pot fi îmbunătățiți algoritmi existenți. De asemenea, prin documentarea procesului de instalare și evaluarea performanței folosind benchmark-uri vedem cum utilizăm practic acest instrument.

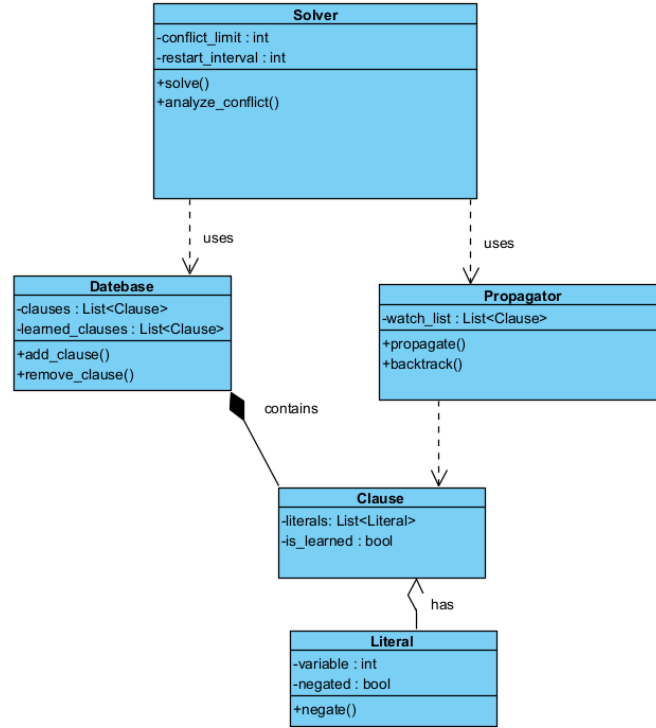


Figura 1. Diagrama UML a componentelor principale MiniSat

1.2 Scopul și obiectivele lucrării

Din punct de vedere tehnic, vrem să studiem structurile de date și algoritmi DPLL și CDCL folosiți în MiniSat, să identificăm posibile îmbunătățiri ale acestora și să documentăm implementarea acestora. De asemenea, vom analiza codul sursă pentru a înțelege structura internă și a propune îmbunătățiri.

Pentru partea practică, ne propunem să instalăm și să configurăm mediul de lucru, să rulăm o familie de benchmark-uri din competiția SAT 2024 cu un timp limită de 24 de ore și să măsurăm performanța obținută.

1.3 Instalare MiniSat

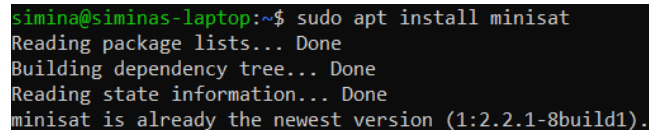
Pentru instalarea MiniSat pe un sistem de operare Windows, am ales să utilizăm Windows subsystem for Linux. În primul rând, a fost necesară instalarea WSL. Pentru aceasta, am deschis PowerShell cu drepturi de administrator și am executat comanda:

```
1 wsl --install
```

După instalare, sistemul a necesitat o repornire pentru a finaliza configurarea. Acest proces instalează Ubuntu, care este suficientă pentru necesitățile noastre.

După configurarea WSL și pornirea terminalului Ubuntu, am actualizat gestiunea de pachete, am instalat Minisat iar la final am verificat instalarea. Am făcut aceasta prin următoarele comenzi:

```
1 sudo apt update
2 sudo apt install minisat
3 minisat
```



```
simina@siminas-laptop:~$ sudo apt install minisat
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
minisat is already the newest version (1:2.2.1-8build1).
```

Figura 2. Instalarea MiniSat

După cum se poate observa în Figură, comenzile au fost executate cu succes, confirmând că MiniSat este instalat și funcțional în sistem.

2 Rezultate experimentale

2.1 Context

Pentru derularea testelor s-au folosit două familii de benchmark-uri preluate din **Global Benchmark Database** [1]. În primul rând, a fost rulată întreaga familie de benchmark-uri **tseitin-formulas**, fiecare test din aceasta având un timp de execuție limitat la 3 ore. Familia de benchmark-uri tseitin-formulas este derivată din transformarea Tseitin, aceasta convertind grafuri în formule propoziționale [5], fiind adesea nesatisfiabile, dar oferind benchmark-uri controlate și structurate, acestea utilizându-se adesea în competiții. În al doilea rând, a fost folosită familia **equivalence-chain-principle**, pentru fiecare dintre teste având timpul de execuție limitat la 6 ore. Familia equivalence-chain-principle a fost dezvoltată pentru testarea solver-elor, aceasta fiind bazată pe lanțuri de echivalențe și implicații [3] și având spații mari de soluții. Pentru a optimiza procesul de rulare a benchmark-urilor a fost folosit un script de python, acesta având ca proprietăți rularea automată a tuturor benchmark-urilor din interiorul unui folder, dându-le un timp de rulare limitat fiecăruia și redirectionând output-ul solver-ului într-un fișier text salvat la trecerea timpului alocat fiecărui proces [2].

2.2 Interpretarea rezultatelor

Familia Tseitin-formulas Pentru a derula testele asupra întregii familii de benchmark-uri **Tseitin-formulas** a fost alocat un interval de timp de 24 de ore. Familia având 8 benchmark-uri, s-a acordat o perioadă de rulare de 3 ore fiecărui benchmark. Un fapt demn de menționat este acela că fiecare benchmark din această familie are ca rezultat UNSAT. În urma rulării acestor teste s-a putut observa că în cadrul majorității, acestea atingeau rapid un progres cu o valoare sub 0.01% iar după aceea nu mai avansau, numărul de conflicte crescând vertiginos, excepție a făcut un singur test care a evoluat până la un progres de 14% după aceea rata de creștere scăzând drastic.

Tabela 1. Rezultate rulare benchmark-uri din familia tseitin-formulas.

Nume	Progres	Număr de conflicte	Rezultat așteptat	Rezultat în urma rulării
0fa8623240b5	14.618 %	17044326	UNSAT	NA
07b476c83150	0.009 %	436832664	UNSAT	NA
24bdfb34654b	0.002 %	291221704	UNSAT	NA
36dbc206c7d9	0.001 %	291221704	UNSAT	NA
556cf803f9de	0.001 %	291221704	UNSAT	NA
35789168b67a	0.000 %	194147731	UNSAT	NA
aa39c6d885d2	0.019 %	291221704	UNSAT	NA
c69c5163c908	0.002 %	291221704	UNSAT	NA

Așa cum se poate observa din tabelul de mai sus nici unul dintre benchmark-uri nu a reușit să ruleze până la final în timpul limită de 3 ore, iar un alt aspect notabil este acela că 5 din cele 8 teste au ajuns la același număr de conflicte înainte de finalizarea rulării, testul care a progresat cel mai mult având de asemenea și un număr semnificativ mai mic de conflicte comparativ cu restul.

Familia equivalence-chain-principle Am folosit un interval de timp de 24 de ore pentru a rula această familie, compusă din 4 benchmark-uri, fiecărui benchmark atribuindu-se câte 6 ore. Se aștepta valoarea UNSAT pentru toate cele 4 seturi de date. Două dintre ele au avut un progres foarte rapid, 10%, respectiv 15%, cel din urmă atingând și o finalitate. Pentru celelalte 3 benchmark-uri nu s-a obținut un rezultat final.

Tabela 2. Rezultate rulare benchmark-uri din familia equivalence-chain-principle.

Nume	Progres	Număr de conflicte	Rezultat așteptat	Rezultat în urma rulării
8d2dfc50c175	0.561 %	129431749	UNSAT	NA
9cd3acdb765c	15.267 %	3134121	UNSAT	UNSAT
63732c330adb	10.034 %	17044326	UNSAT	NA
f18c7c7b8666	0.652 %	129431749	UNSAT	NA

De menționat este că rezultatul celui de-al doilea set de date a fost obținut într-un timp de 671.383 secunde. Se poate observa cum două dintre benchmark-uri s-au oprit după ce au atins același număr de conflicte în timpul acordat.

3 Structura MiniSat

Pentru a determina satisfiabilitatea formulelor propoziționale, din punct de vedere structural, proiectul MiniSat implementează o multitudine de structuri și algoritmi, meniți să ruleze împreună în mediul proiectului.

LBool

Clasa LBool este definită în fișierul SolverTypes.h și are un singur atribut de tipul uint8_t, denumit *value*. Obiectele de tip Lbool sunt folosite pentru reprezentarea a trei stări:

- *l_True*: reprezintă starea *Adevărat* și are valoarea 0
- *l_False*: reprezintă starea *Fals* și are valoarea 1
- *l_Undef*: reprezintă starea *Nedefinit* și are valoarea 2

Clasa implementează 3 constructori, doi expliți și unul default, utilizați în diferite cazuri de creare a obiectelor, după cum urmează:

- Un constructor explicit ce primește drept parametru o variabilă de tip uint8_t, unde atributul de tip value al obiectului LBool este inițializat corespunzător cu valoarea pasată ca parametru
- Un constructor implicit, default, fără parametri
- Un constructor explicit ce primește drept parametru o variabilă de tip bool, unde atributul de tip value al obiectului LBool este inițializat corespunzător în funcție de valoarea logică, 0 pentru Adevărat și 1 pentru Fals, conform mențiunilor anterioare.

Supraîncărcarea operatorilor Clasa Lbool supraîncarcă operatorii ==, !=, &&, || și ^

1. Supraîncărcarea operatorului ==.

Metoda primește drept parametru un obiect de tip lbool și returnează adevărat sau fals, în urma evaluării următoarei expresii:

```
((b.value&2) & (value&2)) |(!((b.value&2)&(value == b.value)))
```

Expresia verifică dacă obiectul curent și cel primit ca parametru sunt ambele nedefinite sau dacă obiectul primit ca parametru este definit și valoarea acestuia este egală cu valoarea obiectului curent. Astfel, în cazul în care cel puțin una dintre aceste condiții este îndeplinită, atunci metoda va returna true, sau false în caz contrar.

2. Supraîncărcarea operatorului `!=`.
Metoda primește drept parametru un obiect de tip `lbool` și returnează adevărat sau fals, în urma evaluării expresiei: `!(*this == b)`. Astfel, metoda va returna false, dacă obiectul primit ca parametru este egal cu obiectul curent și true în caz contrar.
3. Supraîncărcarea operatorului `&&`.
Metoda primește drept parametru un obiect de tip `lbool` și returnează rezultatul operației logice AND pentru obiecte de tip `lbool`, utilizând operații pe biți.
4. Supraîncărcarea operatorului `||`.
Metoda primește drept parametru un obiect de tip `lbool` și returnează rezultatul operației logice OR pentru obiecte de tip `lbool`, utilizând operații pe biți.
5. Supraîncărcarea operatorului `^`.
Metoda primește drept parametru un obiect de tip `bool` și returnează rezultatul operației logice XOR, astfel: dacă valoarea primită ca parametru este false, valoarea operației XOR rămâne neschimbată, iar dacă valoarea primită ca parametru este true, atunci se schimbă valoarea de adevăr utilizând operații pe biți.

Funcții prietene Clasa `lbool` implementează două funcții prietene, `toInt`, respectiv `toLbool`, cu scopul de a converti în mod corespunzător un obiect de tip `lbool` într-un `int`, prin returnarea valorii, respectiv un obiect de tip `int` într-un obiect de tip `lbool`, utilizând constructorul aferent.

Lit

Structura de date `Lit` reprezintă abstractizarea literalilor din contextul satisfiabilității formulelor propoziționale. Aceasta este implementată în fișierul `SolverTypes.h` și are un singur atribut de tip `int`, denumită x , utilizată pentru stocarea internă a valorii literalului.

Din punct de vedere al constructorului, structura utilizează metoda `mkLit`, ce primește drept parametrii un obiect de tip `Var`, definit de un `int`, și un obiect de tip `bool`, care marchează prezența semnului și returnează un obiect `Lit` corespunzător. Se instanțiază un nou obiect `p`, de tip `Lit`, iar atributul său x este calculat după următoarea formulă: $p.x = \text{var} + \text{var} + (\text{int})\text{sign}$. Astfel, dacă semnul este false, adică literalul este pozitiv, valoarea acestuia va fi $p.x = 2 * \text{val}$, iar în cazul în care este negativ, acesta va fi $p.x = 2 * \text{val} + 1$.

Supraîncărcarea operatorilor Structura `Lit` supraîncarcă operatorii `==`, `!=` și `<`, `~` și `^` după cum urmează:

1. Supraîncărcarea operatorului `==`.
Metoda primește drept parametru un obiect de tip `Lit` și verifică dacă atributul său x este egal cu atributul x al obiectului curent, returnând true în caz afirmativ și false în caz contrar.

2. Supraîncărcarea operatorului $!=$.
Metoda primește drept parametru un obiect de tip Lit și verifică dacă atributul său x este diferit de atributul x al obiectului curent, returnând true în caz afirmativ sau false în caz contrar.
3. Supraîncărcarea operatorului $<$.
Metoda primește drept parametru un obiect de tip Lit și verifică dacă atributul x al obiectului curent este mai mic decât atributul x al obiectului primit ca parametru, returnând true în caz afirmativ, respectiv false în caz contrar.
4. Supraîncărcarea operatorului \sim .
Metoda primește ca parametru un obiect de tip Lit și returnează rezultatul negației pe biți al acestuia prin schimbarea semnului acestuia.
5. Supraîncărcarea operatorului \wedge .
Metoda primește ca parametrii un obiect de tip Lit și un obiect de tip bool și returnează rezultatul operației logice XOR, convertind variabila bool la tipul *unsigned int* și utilizând operații pe biți.

Funcții ajutoare Structura de tip Lit implementează și diferite funcții ajutoare, precum funcții de tip getter pentru semnul și valoarea unui literal, cât și funcții de conversie explicită a datelor, precum conversia unui obiect de tip Var în int(*toInt*), conversia unui obiect de tip Lit în int, prin returnarea atributului x a parametrului pasat, cât și conversia în sens invers, a unui obiect de tip int într-un obiect de tip Lit (*toLit*).

Constante Se definesc de asemenea două obiecte de tip Lit constante, unul ce reprezintă literalul nedefinit și are atributul $x = -2$ și unul pentru cazurile de eroare, ce are atributul $x = -1$.

Clause

Clasa Clause din fișierul SolverTypes.h reprezintă o abstractizare a clauzelor din cadrul formulelor propoziționale în formă normal conjunctivă, fiind reprezentată de o disjuncție de literal. Clasa dispune de doi constructori expliți, parametrizați, dintre care unul de copiere, dar care nu pot fi utilizați în mod direct la crearea clauzelor, datorită faptului că nu alocă destulă memorie. Astfel, crearea clauzelor se realizează prin intermediul unei alte clase, mai exact *ClauseAllocator*, care este clasă prietenă a clasei Clause.

Funcții membre

- *calcAbstraction()* : metoda calculează o abstractizare aferentă fiecărei clauze la nivel de biți, utilizând literalii care alcătuiesc clauza
- *size()*: metoda returnează numărul de literali din alcătuirea clauzei
- *shrink(int i)*: metoda primește drept parametru un număr întreg i și reduce dimensiunea clauzei cu i literali

- `pop()`: metoda se folosește de metoda `shrink` pentru a elimina un literal din clauză
- `learnt()`: metoda verifică dacă clauza este învățată
- `has_extra()`: metoda verifică dacă clauza are informații extra
- `const Lit& last()`: metoda returnează ultimul literal din alcătuirea clauzei
- `abstraction()`: returnează abstractizarea clauzei, utilizată în operațiile de subsumes.
- `subsumes(const Clause)`: verifică dacă clauza curentă satisface logic o altă clauză, sau dacă cealaltă clauză poate fi simplificată cu ajutorul primei clauze prin rezoluție. Aceasta returnează:
 - `lit_Error`: Clauza nu se poate simplifica sau satisface cu ajutorul clauzei curente
 - `lit_Undef`: Clauza este tautologie și devine redundantă în determinarea satisfiabilității
 - `Lit p`: Literalul ce se poate elimina din cea de a doua clauză.

Supraîncărcarea operatorilor

- `operator[]`: asigură accesul prin index la literalii care alcătuiesc clauza
- `strengthen(Lit)`: Elimină literalul primit ca parametru din clauza curentă și recalculează abstractizarea acesteia.

ClauseAllocator

Clasa `ClauseAllocator` este o clasă utilizată cu scopul alocării memoriei aferente unei clauze. Aceasta are un atribut `ra` de tip `RegionAllocator<uint32_t>` ce are drept scop gestionarea blocurilor de memorie alocate clauzelor și un atribut `extra_clause_field` de tip `bool` pentru a marca dacă este necesară alocarea de memorie suplimentară clauzei, cu scopul gestionării eficiente a memoriei.

În ceea ce privește constructorul, clasa `ClauseAllocator` dispune de un constructor default, fără parametri, cât și de unul explicit, care inițializează atributul `ra` cu o dimensiune inițială și flag-ul pentru memorie adițională, `extra_clause_field`, cu false.

Funcții membre

- `clauseWord32Size(int, bool)`: funcție care calculează memoria necesară pentru a stoca o clauză de o anumită dimensiune (un anumit număr de literalii), luând în considerare dacă aceasta necesită memorie suplimentară.
- `alloc(const vec<Lit>&, bool)`: alocă memorie pentru o nouă clauză, în funcție de o listă de literalii și dacă clauza este o clauză învățată, informații pasate drept parametrii.
- `alloc(const Clause&)`: alocă memorie pentru o nouă clauză utilizând informațiile altei clauze.
- `size()`: returnează dimensiunea totală a memoriei salvată în atributul `ra`
- `wasted()`: returnează cât este utilizat din memorie
- `free(CRef)`: eliberează memoria unei anumite clauze

ClauseIterator

Clasa `ClauseIterator` este utilizată pentru a putea parcurge clauzele stocate cu ajutorul clasei `ClauseAllocator`. Aceasta are două atribute:

1. Un atribut de tipul `ClauseAllocator&`, denumit *ca*, o referință către obiectul unde sunt stocate clauzele
2. Un atribut de tipul `CRef*`, denumit *crefs*, un pointer pentru accesarea clauzelor

Constructorul este parametrizat și este responsabil de inițializarea corectă a atributelor membre clasei.

Supraîncărcarea operatorilor

- `operator++()`: utilizată pentru a trece către următoarea clauză din listă
- `Clause& operator*()`: utilizată pentru accesarea directă a clauzei curente
- `bool operator==(const ClauseIterator&)`: returnează adevărat dacă atributele lor *crefs* pointează către același bloc de memorie(aceeași clauză)
- `bool operator!=(const ClauseIterator&)`: returnează adevărat dacă atributele lor *crefs* nu pointează către același bloc de memorie(clauze diferite)

TrailIterator

Clasa `TrailIterator` are drept atribut membru un pointer către o listă de literali, *lits*, cu scopul de a facilita parcurgerea acestora. Constructorul este parametrizat și este utilizat pentru a inițializa obiectul corespunzător.

Supraîncărcarea operatorilor

- `operator++()`: utilizată pentru a trece la următorul literal din listă
- `Lit operator*()`: utilizată pentru accesarea directă a literalului curent
- `bool operator==(const TrailIterator&)`: returnează adevărat dacă atributele lor *lits* pointează către același bloc de memorie(același literal)
- `bool operator!=(const TrailIterator&)`: returnează adevărat dacă atributele lor *lits* pointează către blocuri de memorie diferite(literali diferiți)

Solver

Clasa `Solver` din interiorul fișierului `Solver.cc` reprezintă fundamentul întregului proiect, atributele sale fiind responsabile de tehnicile și algoritmi specifici problemei satisfiabilității, precum propagarea și operațiile pe clauze, utilizând variabile și funcții specializate.

Atributele membre clasei

- `var_decay`: contor care prioritizează alegerea unei variabile într-un arbore de decizie, bazat pe conflictele în care este implicată.
- `clause_decay`: contor care prioritizează alegerea unei clauze, bazat pe conflictele în care este implicată.
- `ccmin_mode`: specifică gradul de simplificare a conflictelor utilizat de către solver
- `garbage_frac`: contorizează gradul de ocupare a memoriei pe baza căruia se activează garbage collector-ul.
- variabile care eficientizează procesul de căutare: `random_var_freq`, `phase_saving`, `luby_restart`, `restart_first`, `restart_inc`.

Algoritmul Davis-Putnam-Logemann-Loveland (DPLL) Algoritmul DPLL este un algoritm consacrat de enumerare, bazat pe backtracking și pe propagare pentru a determina satisfiabilitatea formulelor propoziționale. În cadrul MiniSat, elemente fundamentale acestui algoritm sunt implementate utilizând metodele clasei Solver, localizate în fișierul Solver.cc și funcționează conform următorului flux:

1. **Inițializarea datelor:** Funcția *solve()* a clasei Solver este responsabilă de inițializarea datelor și pornește algoritmul de atribuire a variabilelor în vederea identificării unui posibil model pentru formula propozițională în CNF (formă normal conjunctivă).
2. **Atribuirea recursivă:** Funcția *search()* a clasei Solver implementează parcurgerea iterativă prin care variabilele sunt atribuite valorile de adevăr variabilelor. Constrângerile sunt pe rând propagate, utilizând metoda *propagate()*, componentă definitorie a tehnicii DPLL, până la identificarea unui conflict, moment în care acestea sunt trimise spre analizare și algoritmul se întoarce, în urma analizei acestuia, pe nivelul care a cauzat conflictul, utilizând o variantă a tehnicii backtracking și anume, backtracking non-cronologic.

Conflict-Driven Clause Learning (CDCL) Algoritmul CDCL este un algoritm bazat pe DPLL, care eficientizează determinarea satisfiabilității unei formule propoziționale în formă normal conjunctivă (CNF), utilizând backtracking non-cronologic (backjumping) și tehnici specifice de analizare a conflictelor.

Fluxul de execuție :

1. **Inițializarea datelor:** Funcția *solve()* a clasei Solver este responsabilă de inițializarea datelor și pornește algoritmul de atribuire a variabilelor în vederea identificării unui posibil model pentru formula propozițională în CNF (formă normal conjunctivă), sau a conflictului care determină nesatisfiabilitatea acesteia.

2. **Atribuirea recursivă:** Funcția *search()* a clasei Solver implementează parcurgerea iterativă prin care variabilele sunt atribuite valorile de adevăr variabilelor. Constrângerile sunt pe rând propagate, utilizând metoda *propagate()* până la identificarea unui conflict, moment în care acesta este pasat spre funcția *analyze()* pentru a identifica sursa acestuia, dacă nu ne aflăm pe primul nivel de decizie. În caz contrar, căutarea se oprește și formula este nesatisfiabilă.
3. **Propagarea:** Metoda *propagate()* este utilizată pentru a propaga constrângerile pentru clauzele aferente. Această iterează prin lista de literali și pentru fiecare dintre aceștia, iterează prin lista *watches* (clauze în care literalul apare negat) și se verifică dacă există un literal în aceste clauze, care este adevărat și ar putea reduce clauza la o tautologie. Dacă un literal din această listă *watches* devine fals, se încearcă înlocuirea sa cu un alt literal neatribuit din clauză. Dacă nu se poate înlocui, clauza rămâne unitară, iar literalul ori este adăugat în coada literalilor pentru propagare, ori se generează un conflict, așa cum este explicat și în [4].
4. **Analiza conflictelor:** Conflictele întâmpinate pe parcursul propagării sunt pasate către funcția *analyze()*, unde se identifică clauza ce trebuie reținută pentru a evita reproducerea conflictului în următoarele propagări, cât și nivelul pe care trebuie să se reîntoarcă utilizând tehnica de backjumping. Se parcurge graful de implicații care au cauzat conflictul, fiecare literal nou apărut contorizându-se, precum și nivelul de decizie pe care se află. La final, se formează o clauză care se memorează cu ajutorul literalilor și se determină cel mai mic nivel de pe care se poate relua propagarea.
5. **Backjumping:** Tehnica backtracking non-cronologică, cunoscută și ca backjumping, eficientizează căutarea și este implementată în MiniSat prin funcția *cancelUntil()* care resetează nivelul de decizie la nivelul de la care a pornit conflictul, pasat ca parametru.
6. **Euristici de decizie:** Odată cu trecerea la un alt nivel de decizie, funcția *pickBranchLit()* decide ce literal trebuie inițializat în continuare pe baza unei euristici caracteristici. În primul rând se încearcă alegerea unui literal aleatoriu, în funcție de polaritatea definită de către utilizator. Dacă acest lucru nu este însă posibil, se crează o structură de tip coadă cu literalii, prioritizați pe baza unui scor de activitate, care este determinat în mod direct de implicarea acestora în conflicte. Astfel, se alege literalul cu cel mai mare scor de prioritate și i se alege polaritatea corespunzător (adevărat sau fals).
7. **Gestionarea clauzelor învățate:** Pentru a eficientiza alocarea memoriei, cât și complexitatea algoritmului din punct de vedere al timpului, clauzele învățate sunt analizate, pentru a putea șterge clauzele care nu sunt sugestive algoritmului, cu ajutorul funcției *reduceDB()*, unde clauzele sunt sortate în funcție de conflictele în care sunt incluse și ținând cont de valoarea atributului *clause_decay*, clauzele neutilizate fiind eliminate.
8. **Simplificarea:** Funcția *simplify()* este responsabilă de simplificarea clauzelor, prin reducerea tautologiilor și a literalilor care nu influențează valoarea de adevăr a unei clauze prin propagarea constrângerilor aferente, actualiza-

rea listei de literali ce trebuie atribuiți, dar și a clauzelor învățate care sunt deja satisfăcute.

9. **Restart:** Pentru a evita blocarea algoritmului sau vizitarea unor ramuri de decizie prea costisitoare, algoritmul are implementat un mecanism de restart periodic, bazat pe attributele specifice menționate anterior. acesta apelându-se în funcție de numărul de conflicte întâmpinat pe parcursul propagărilor succesive.

4 Organigrama echipei

Anghel Costel Junior: Testarea familiei Equivalence-Chain-Principle, testare cazuri particulare de benchmarks, identificarea șabloanelor în rezultatele obținute, descriere rezultate, realizare bibliografie.

Coroian Dan Marius: Testarea familiei Tseitin-Formulas, identificare potențiale îmbunătățiri ale codului, descriere rezultate, dezvoltarea scriptului utilizat pentru rularea benchmark-urilor.

Covercă Elena Daniela: Analiza și identificarea claselor principale și a structurilor de date utilizate, analiza și identificarea algoritmilor specfici de determinare a satisfiabilității, documentarea informațiilor identificate.

Bogdan Simina Elisa: Realizarea rezumatului și a părții introductive, a procesului de instalare a MiniSat, realizarea diagramei UML pe baza codului sursă.

Bibliografie

1. Global Benchmark Database. https://benchmark-database.de/?track=main_2024
2. Anghel, C.J., Covercă, E.D., Coroian, D.M., Bogdan, S.: Minisat Improvement. <https://github.com/CoroianMarius/minisat>
3. Buss, S.: Fmlachain: Tautologies based on iterated equivalences or implications of boolean formulas.
4. Eén, N., Sörensson, N.: An extensible sat-solver [extended version 1.2]. SAT2003
5. Heule, M.J., Iser, M., Järvisalo, M., Suda, M.: Proceedings of sat competition 2024: Solver, benchmark and proof checker descriptions (2024)
6. of Illinois at Urbana-Champaign, U.: [coursera] vlsi cad: Logic to layout (university of illinois at urbana-champaign) (vlsicad)
7. Wheeler, D.: Minisat user guide: how to use the minisat sat solver. Wonderings of satgeek,[Online]. Available: <http://www.dwheeler.com/essays/minisat-user-guide.html> (2013)