Raport Verificare Formala

Anghel Bogdan, Moise Alexandra, Stancu Maria, and Szekrenyes Benjámin

Facultatea de Matematică și Informatică, Universitatea de Vest din Timișoara, Timișoara, Romania

Rezumat În acest raport, analizăm un SAT solver (MiniSat) folosit pentru rezolvarea problemelor de satisfiabilitate (SAT). Raportul conține procesul de instalare și arhitectura MiniSat, analiza algoritmilor utilizați, CDCL (Conflict-Driven Clause Learning), și diagrama de clase a arhitecturii programului. De asemenea, sunt prezentate provocările întâmpinate, concluziile la care s-a ajuns după parcurgerea tuturor etapelor și optimizările viitoare care pot spori eficiența solver-ului prin modificarea codului.

Keywords: MiniSat · SAT · CDCL · DPLL · Satisfiabilitate · clauze · DIMACS · solver

1 Introducere

Problema satisfiabilității booleene (SAT - studiată în domeniul calculului) constă în determinarea existenței unei atribuiri de valori adevărat/fals (1/0) pentru variabilele unei formule booleene, astfel încât rezultatul interpretării formulei să fie adevărat.

Există probleme din viața reală (Sudoku, planificarea orarelor, rutelor, gestionarea resurselor, verificarea software-ului, criptografie și analiza de date) ce pot fi transpuse sub formă de relații între elemente și constrângeri. Acestea trebuie respectate simultan prin exprimarea lor ca formule logice în care fiecare variabilă reprezintă o condiție sau o decizie. Prin traducerea acestor probleme în logică propozițională și utilizarea unui solver SAT (MiniSat), se poate determina rapid și eficient dacă există o soluție (model) care satisface toate constrângerile impuse.

În acest raport se prezintă structura și funcționalitățile MiniSat (unul dintre cei mai folosiți SAT solvers), analizând performanța acestuia pe un set de benchmark-uri din competiția SAT 2024.

2 Descrierea problemei

Problema satisfiabilităţii, notată prescurtat cu SAT, cere determinarea existenţei pentru o formulă booleană F a unei artibuiri satisfiabile. O formulă booleană F este compusă din variabile logice (x, y, ...) , operatori logici (şi, sau,

non, implicaţie şi echivalenţă), şi paranteze. O atribuire de valori booleene pentru variabilele acestei expresii se numeşte atribuire satisfiabilă dacă evaluarea expresiei după atribuirea valorilor dă ca rezultat valoarea de adevăr. [Neg08]

Orice formulă booleană poate fi transformată în două forme:

- Forma normal conjunctivă în care expresia este exprimată ca o conjuncţie de propoziţii, iar fiecare propoziţie este formată din o disjuncţie de literali. Un literal este o variabilă care poate fi sau nu negată.
- Forma normal disjunctivă este formată ca o disjuncţie de propoziţii, fiecare dintre aceste propoziţii fiind o conjuncţie de literali. [Neg08]

Problema satisfiabilității folosește pentru datele de intrare o formulă propozitionala adusă în formă normal conjunctivă, numită și formă clauzală. Pe disjuncțiile astfel obținute (numite clauze) se aplică diverse metode pentru a găsi asignări ale variabilelor (interpretări) care să conducă la evaluarea "true" a formulei și obținerea unui "model" pentru formula evaluată.

Enumerarea interpretărilor, încercarea tututor combinațiilor de asignări, este o metodă simplă care întotdeauna va da rezultate dar nu este o abordare neapărat eficientă în cazul unui număr mare de literali și implicit al unui set mare de combinații posibile. Astfel, îmbunătățiri ale acesteia au fost implementate, bazându-se pe ideea "propagării unitare a clauzelor" pentru procesarea asignărilor posibile într-un mod performant. MiniSat este un solver pentru problemele de satisfiabilitate booleeană (SAT), care face parte din categoria algoritmilor de tip DPLL (Davis-Putnam-Logemann-Loveland).

MiniSat este un proiect open-source și ajuta la dezvoltarea altor solvere SAT și SAT-modulo-Theories (SMT). Codul acestuia este scris în C++ și este implementat în așa fel încât resursele disponibile să fie alocate și folosite într-un mod cât se poate de eficient.

3 Instalare MiniSat

Am optat pentru instalarea MiniSat-ului atât pe Windows, cat și pe Linux, pentru a putea verifica performanțele pe diferite sisteme de operare.

3.1 Instalare Windows

1. Am descărcat sursele MiniSat din repository-ul GitHub al proiectului.
2. Am configurat mediul de dezvoltare, verificând că este instalat un compilator C++, apoi am instalat CMake pentru build.
3. După ce am extras arhiva descărcată, am navigat la directorul MiniSat, apoi am rulat comanda *mingw32-make*.
4. După compilare, s-a generat executabilul MiniSat (*minisat.exe*).
5. Pentru a verifica funcționarea corectă, am rulat un benchmark din competiția SAT 2024.

3.2 Instalare Linux

1. Folosind comanda în CMD: *wsl –install*, am reușit să instalez cu succes ultima versiune de Ubuntu. Odată ce am avut acces la Ubuntu, a rămas de instalat doar MiniSat-ul.
2. A trebuit folosită doar comanda: *sudo apt install minisat*. În final, a trebuit să verificăm dacă funcționează totul bine și dacă MiniSat-ul poate rula direct din terminalul Ubuntu. Comanda: *minisat* a fost suficientă, iar din output am putut vedea clar că este totul în regulă.
3. Acum, trecând de toate aceste etape, am rulat un benchmark din competiția SAT 2024, același benchmark ce a fost folosit la Windows.

4 Rezultate experimentale

La prima familie de teste, Polynomial Multiplication, nu prea pot să intru în detaliu când vine vorba de rezultate, pentru că, așa cum am menționat în secțiunea de provocări, două teste din patru au mers, iar la celelalte două a apărut eroarea: "Process Killed", în ciuda faptului că foarte multă memorie a fost alocată.

Totuși, să vorbim despre a doua familie, pentru că, mulțumită acestor teste, am putut să realizăm sarcina până la capăt.

Familia: Software Verification. Această familie a avut 15 inputuri, iar toate testele au rulat până la capăt. Lucrurile au mers surprinzător de repede aici, a durat mai puțin de 6 ore să avem rezultatele la teste, un interval de timp, considerabil, mai scurt în comparație cu timpul așteptat inițial la cealaltă familie (peste 27 de ore).

Totuși, încă un lucru surprinzător s-a întâmplat aici. Din toate aceste teste, nu a existat un rezultat care să fie SAT. Toate testele, în final, au arătat că rezultatul a fost UNSAT. Rezultatele sunt prezentate în tabelul 1.

5 Provocări întâmpinate

Au fost întâmpinate niște obstacole greu de depășit, iar în unele cazuri a trebuit să alegem cu totul altă familie de teste pentru a putea merge mai departe.

Prima familie de teste aleasă a fost Polynomial Multiplication. Aici au apărut problemele. Testele din această familie au necesitat o cantitate uriașă de memorie, peste 8GB, iar laptopul pe care au rulat testele era inițial setat să folosească doar 8GB de memorie.

Pentru a depăși această problemă, s-a recurs la metoda de a folosi un Swap File. Swap File permite folosirea memoriei de pe disc, iar făcând asta a putut să aloce peste 16GB de memorie pentru a rula familia Polynomial Multiplication.

După ce au trecut peste 27 de ore, din păcate, nu am putut să rulăm două teste din această familie până la capăt, întâmpinând eroarea: "Process Killed". Cauza ei a fost una destul de neașteptată: "Not enough Memory Allocated". Cu

Test Name	Memory	Cpu Time(in seconds)	Result
02f6	5824.00 MB	392.686 s	unsat
0d31	5385.00 MB	276.014 s	unsat
0f26	6090.00 MB	383.723 s	unsat
1e0d	4908.00 MB	252.987 s	unsat
25cc	5367.00 MB	1726.24 s	unsat
44fd	739.00 MB	1487.08 s	unsat
573c	7617.00 MB	461.481 s	unsat
6008	7202.00 MB	392.071 s	unsat
878b	2495.00 MB	90.9715 s	unsat
aaa3	4783.00 MB	263.501 s	unsat
c03f	5772.00 MB	327.535 s	unsat
c21c	5877.00 MB	327.225 s	unsat
c8b1	7489.00 MB	418.657 s	unsat
e8ab	3779.00 MB	217.114 s	unsat
f54b	3635.00 MB	172.186 s	unsat

Tabela 1. Interpretare rezultate experimentale

toate că a fost folosită memorie de pe disc, nu a fost suficient pentru a duce până la capăt aceste două teste.

Așa că am recurs la altă metodă, o abordare destul de simplă, dar care a mers cel mai rapid. Am ales o cu totul altă familie de teste și am luat-o de la capăt din nou. Familia aleasă de data aceasta a fost: Software Verification.

Din fericire, lucrurile au decurs mult mai bine aici, toate testele putând să ruleze până la capăt. Niciun test nu a rulat mai mult de o oră, iar memoria nu a mai fost o problemă. Rezultatele testelor sunt analizate mai în detaliu la secțiunea: "Rezultate experimentale".

6 Prezentarea algoritmilor principali

6.1 DPLL (Davis-Putnam-Logemann-Loveland)

DPLL este un algoritm clasic folosit pentru rezolvarea problemelor de satisfiabilitate booleană (SAT). Acesta extinde metoda de rezolvare prin backtracking prin introducerea unei tehnici esențiale: propagarea unitară.

Algoritmul funcționează recursiv, alegând o variabilă de decizie, atribuie o valoare (true sau false) și propagă consecințele acestei decizii prin verificarea clauzelor unitare. Dacă apare un conflict (o clauză devine imposibil de satisfăcut), algoritmul face backtracking pentru a explora alte ramuri ale spațiului de soluții.

6.2 CDCL (Conflict-Driven Clause Learning)

CDCL este o extindere a DPLL care introduce învățarea clauzelor (clause learning) și backtracking non-cronologic, ceea ce îl face mult mai eficient pentru probleme complexe.

Pe lângă propagarea unitară, CDCL analizează conflictele atunci când apar și generează clauze suplimentare pentru a preveni repetarea acestora în viitor. Aceste clauze învățate reduc semnificativ spațiul de căutare.

Backtracking-ul non-cronologic permite algoritmului să sară direct la nivelul de decizie relevant pentru conflict, în loc să refacă toate deciziile intermediare. CDCL folosește, de asemenea, euristici avansate pentru alegerea variabilelor și gestionarea priorităților acestora, ceea ce îl face standardul pentru SAT-solvers moderne.

7 Arhitectura MiniSat

7.1 Structura MiniSat

1. Fișiere CNF
   MiniSat folosește fișiere de input în format DIMACS CNF: [NE07]
   - se scrie formula în forma normală conjunctivă.
   - asociem fiecărui literal indexul său (un număr întreg, începând de la 1).
   - prima linie din fișier (linia problemei): *p cnf <nr_lit> <nr_clauze>*.
   - rescriem fiecare clauză pe o linie din fișier:
     - atomii sunt în forma de index cu '-' în față dacă sunt negați și cu spații între ei.
     - '0' la sfârșit de clauză (ultimul caracter de pe linie).
2. Procesare
   - Set de date de intrare
     Pentru a se forma setul de clauze din fișierul de intrare de tip CNF, este implementat un 'DIMACS parser' ce conține metode pentru: [NE07]
     - citirea clauzei: parcurgerea atomilor din fișier până la întâlnirea '0' (sfârșit de clauză) și salvarea literalilor găsiți într-o listă.
     - parcurgerea fișierului: linie cu linie, se apelează metoda pentru citirea clauzei, ca apoi fiecare dintre ele să fie adăugată de solver.
   - Tipuri de date folosite
     Propriile abstractizări pentru tipurile de date care reprezintă literalii și clauzele:
     - tipuri atomice (transmise prin valoare):
       - lbool: l_true, l_false, l_undefined (dacă încă nu a fost interpretat).
       - lit: var = 2 * index + sign (codificare pentru reprezentarea literalului pozitiv/negativ).
     - tipuri compuse:
       - Clause: *header* (learnt, size, etc.) + *data* (referință CRef, literali).
       - Vec, Queue - pentru interpretări, Map - variabile euristice, Heap - pentru prioritatea literalilor, etc.
   - API (funcționare generală) Interfața, metodele de interacționare cu Solver-ul, este folosită astfel:
     - variabile introduse apelând *newVar()*.

- construirea clauzelor din variabile și adăugarea lor folosind *addClause()*: se detectează existența „conflictelor triviale" (două clauze unitare contradictorii) - caz în care se returnează *FALSE*.
- rezolvarea modelului: dacă la adăugarea de clauze nu au fost detectate conflicte, se apelează funcția *Solve()* cu o listă de „assumptions" (presupuneri) goală. Aceasta returnează:
  * *FALSE* dacă problema este UNSAT.
  * *TRUE* dacă problema este SAT, iar din vectorul *model* se citește interpretarea (modelul) problemei.
- simplificarea setului de constrângeri: propagarea unității și eliminarea clauzelor satisfăcute: *simplify()*.

– Prezentare generală
  - SAT-solver bazat pe DPLL, backtracking la întâlnirea conflictelor și învățarea clauzelor.
  - categorii, concepte de analizat:
    * reprezentare: clauze și asignări.
    * deducere (din engleză: „inference"): propagarea unitară.
      · pentru fiecare literal, se păstrează liste de constrângeri care ar putea conduce la propagarea unitară prin asignarea variabilei.
      · pentru clauze se verifică să nu fie clauze unitare (se pot propaga direct) și se setează 2 literali ca fiind observați („watched"), se creează un obiect de tip *Watcher* care stochează referința clauzei și celălalt literal din pereche, obiect care va fi adăugat în lista de „watchers" a literalului negat (opus).
      · sistemul de „watcher" aduce ca avantaj, în cazul clauzelor, că la *backtracking* nu este necesară ajustarea listelor de *watchers*, rezultând un backtracking eficient din punct de vedere al resurselor.
    * învățare: pe baza conflictelor.
      · când are loc un conflict (o constrângere devine imposibil de satisfăcut), MiniSat analizează ce combinație de variabile l-a generat – variabilele sunt fie decizii (asignări) sau rezultate ale unei propagări.
      · cauzele conflictului sunt urmărite pas cu pas înapoi (backtracking) de pe stiva de asignări (*trail*) până la îndeplinirea unei condiții de oprire, rezultând setul de variabile care a produs conflictul.
      · aceste variabile sunt atașate unei clauze care interzice asignarea ce a dus la conflict (conjuncție negată) -> clauza învățată care va fi adăugată în baza de date a problemei.
    * căutare: proces iterativ.
      · Pasul 1: alegerea unei variabile de decizie neatribuită.
      · Pasul 2: se propagă consecințele asignării variabilelor.
      (a) toate variabilele asignate ca și consecință sunt pe același „nivel de decizie".

(b) interpretarile sunt stocate pe o stivă de decizie (numită „trail") împărțită pe nivele de decizie și folosită pentru backtracking.

· Pasul 3: se iau decizii până când toate variabilele sunt asignate (=> SAT și model) sau are loc un conflict (se apelează procedura de învățare).

∗ euristica de activitate.

· variabile: sunt ordonate după un indicator de activitate. Inițial, se setează o valoare default (0,95) care crește atunci când variabila apare într-o clauză de conflict generată și scade după ce conflictul este înregistrat -> variabilele implicate în conflicte recente au o activitate mai mare decât cele implicate cu ceva timp în urmă.

· clauze: similar variabilelor.

7.2 Diagrama de clase

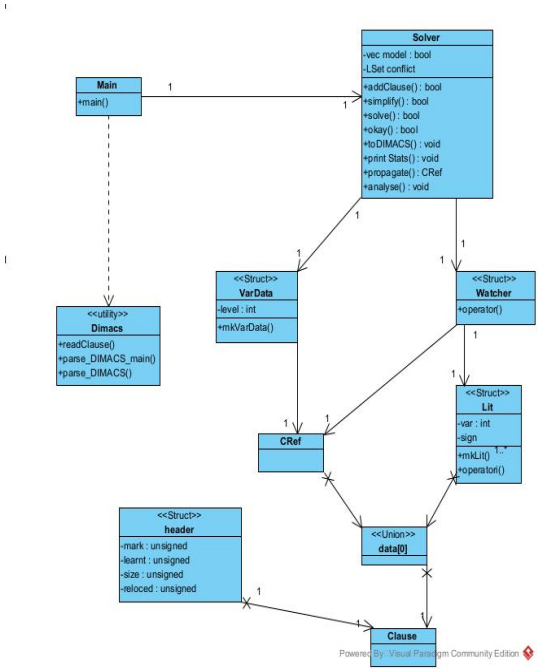

Figura 1. Diagrama de clase

1. **Clasa Main**

Aceasta este punctul de intrare al programului. Metoda `main()` este responsabilă pentru inițializarea programului, citirea formulelor SAT din fișiere în format DIMACS și pregătirea soluției prin utilizarea clasei `Solver`.

2. **Clasa Dimacs «utility»**
   Această clasă conține funcții esențiale pentru citirea și parsarea formulelor SAT din fișiere în format DIMACS:
   - `readClause()` – citește clauzele individuale din fișiere.
   - `parse_DIMACS_main()` – interpretează fișierele și extrage clauzele principale.
   - `parse_DIMACS()` – transformă datele în structuri de clauze gestionabile de către `Solver`.
3. **Clasa Solver**
   Aceasta este clasa principală a programului, care implementează logica de rezolvare a problemelor SAT. Conține următoarele atribute și metode:
   - Atribute:
     - `vec model` – un vector care stochează soluția sau „modelul" problemei.
     - `LSet conflict` – un set utilizat pentru gestionarea stării conflictelor în procesul de backtracking.
   - Metode principale:
     - `addClause()` – adaugă o clauză în baza de date a clauzelor.
     - `simplify()` – simplifică problema eliminând clauzele satisfăcute.
     - `solve()` – metoda principală de rezolvare care implementează logica DPLL și CDCL.
     - `okay()` – verifică dacă problema poate fi satisfăcută în starea curentă.
     - `propagate()` – implementează propagarea unitară pentru a reduce căutarea.
     - `analyze()` – analizează conflictele și determină clauzele noi care trebuie învățate.
     - `toDIMACS()` – exportă rezultatul în format DIMACS.
     - `printStats()` – afișează statistici despre performanța algoritmului (timp, numărul de conflicte, etc.).
     - `cancelUntil()` - backtracking pentru revenirea la un nivel specific
     - metode pentru managementul prioritătii variabilelor și clauzelor pe baza euristicii de activitate
4. **Structura VarData**
   Stochează informații despre nivelul unei variabile în arborele de decizie.
   - Atributul `level` indică nivelul deciziei asociate cu variabila.
   - Metoda `mkVarData()` creează un obiect `VarData`, asigurând inițializarea corectă a datelor.
5. **Clasa CRef**
   Aceasta acționează ca un identificator sau referință către clauze. Este utilizată pentru asocierea dintre `Solver` și clauzele din baza de date.
6. **Structura header**
   Reprezintă informațiile suplimentare despre o clauză. Conține următoarele atribute:

- `mark` – indică starea unei clauze (satisfăcută, învechită, etc.).
- `learnt` – specifică dacă o clauză a fost învățată în timpul analizei conflictelor.
- `size` – dimensiunea clauzei (numărul de literali).
- `relocated` – pentru a gestiona relocarea clauzelor în memorie.

7. **Clasa Clause**
   Reprezintă clauzele din problemă, compuse dintr-un set de literali. Aceasta este constituită din setul de literali (uniunea data[0]) și proprietățile din header.
8. **Structura Watcher**
   Implementarea tehnicii "two-watchers" pentru eficientizarea propagării unitare. Un watcher ține referința către o clauză și un literal, iar pentru a fi folosiți ca mecanism de urmărire a stării clauzelor s-a implementat supraîncărcarea operatorilor de comparare (== și !=).
9. **Structura Lit**
   Reprezintă un literal (o variabilă cu semn asociat).
   - Atribute:
     - `var` – variabila literalului.
     - `sign` – semnul literalului (pozitiv sau negativ).
   - Metode:
     - `mkLit()` – un constructor pentru crearea unui literal nou.
     - supraîncărcarea operatorilor de comparație (==, != și <) pentru a compara și organiza literalii în funcție de valoare sau prioritate.

Relația de asocierea dintre clasele Main și Solver indică faptul că metoda main() a programului utilizează clasa Solver pentru a rezolva probleme SAT. Clasa Solver are relații de asociere cu VarData, CRef, Clause, Watcher și Lit, ceea ce arata că aceste structuri sunt utilizate pentru gestionarea clauzelor și propagarea unitară în timpul căutării soluției.

7.3 Analiza metodelor principale

Metodele principale ale MiniSat sunt fundamentale pentru funcționarea eficientă a solver-ului. Acestea sunt structurate în jurul a doi algoritmi esențiali: DPLL și CDCL, împreună cu o metodă principală pentru căutarea modelului.

- **DPLL (Davis–Putnam–Logemann–Loveland)**
  - *pickBranchLit()*: Aceasta este metoda care decide alegerea unei variabile de decizie neasignate. Alegerea este făcută pe baza euristicilor implementate, cum ar fi activitatea variabilelor sau alte criterii care prioritizează variabilele ce pot conduce rapid la o soluție. Alegerea unei variabile bine optimizate este esentială pentru reducerea numărului total de pași necesari pentru determinarea rezultatului SAT/UNSAT.
  - *propagate()*: Propagarea unitară este procesul prin care se analizează clauzele din setul de constrângeri pentru a deduce valori necesare pentru anumite variabile. Dacă o clauză devine unitară (toți literalii, cu excepția

unuia, sunt negați), valoarea literalului rămas este determinată automat. Această metodă contribuie la simplificarea problemei prin reducerea numărului de variabile neasignate. Propagarea este eficientă datorită utilizării mecanismului de „watcher" pentru urmărirea schimbărilor relevante în clauze.

- **CDCL (Conflict-Driven Clause Learning)**
  - *attachClause() și removeClause()*: Aceste metode sunt responsabile pentru gestionarea bazei de date a clauzelor. *attachClause()* asociază o clauză nouă (fie o clauză învățată, fie una introdusă de utilizator) la lista de clauze relevante pentru propagare. De asemenea, folosește mecanismele de „watcher" pentru eficiență. *removeClause()* gestionează eliminarea clauzelor care au devenit satisfăcute. Acest management este esențial pentru a menține dimensiunea bazei de date sub control și pentru a preveni consumul excesiv de resurse.
  - *cancelUntil()*: Metoda de backtracking utilizată pentru a reveni la un nivel anterior de decizie în cazul unui conflict. Backtracking-ul poate fi total (revenire la nivelul 0) sau parțial (revenire la un nivel de decizie anterior), în funcție de natura conflictului. În acest proces, se elimină toate asignările realizate după nivelul de decizie selectat, dar structurile auxiliare rămân în mare parte intacte.
  - *analyze()*: Aceasta este metoda principală de învățare a conflictelor. În momentul unui conflict, metoda analizează traseul deciziilor și propagărilor care au dus la conflict. Urmând o strategie de învățare, identifică un set minim de variabile și condiții care au provocat conflictul, generând o clauză nouă. Această clauză învățată previne reapariția aceleiași situații de conflict în iterațiile viitoare. Această strategie contribuie semnificativ la eficiența solver-ului, prin reducerea dimensiunii spațiului de căutare.
- ***search()*: căutarea modelului** Metoda *search()* este responsabilă pentru implementarea logicii principale de căutare a unei soluții. Procesul începe cu alegerea unei variabile de decizie utilizând *pickBranchLit()* și propagarea efectelor deciziei utilizând *propagate()*. Dacă se ajunge la o stare de conflict, metoda apelează *analyze()* pentru a învăța o clauză de conflict și *cancelUntil()* pentru a efectua backtracking. Căutarea continuă iterativ, fie până când toate variabilele sunt asignate, indicând că problema este satisfiabilă (SAT), fie până când devine clar că problema nu are soluție (UNSAT). Procesul integrează atât deducerea, cât și învățare pentru a îmbunătăți eficiența.

7.4 Detalii de implementare - eficientizarea procesului

MiniSat integrează mai multe tehnici și structuri de date optimizate pentru a asigura un proces rapid și eficient de rezolvare a problemelor SAT. Acestea sunt descrise mai jos:

- **Stiva de asignări („trail")** Stiva de asignări este o structură utilizată pentru urmărirea stării variabilelor în timpul procesului de căutare.

  - Permite urmărirea și salvarea ordinii în care variabilelor le sunt asignate valori, împreună cu nivelul de decizie corespunzător fiecărei asignări. Aceasta facilitează analiza conflictelor și învățarea clauzelor.
  - Oferă suport pentru revenirea rapidă la un nivel anterior de decizie în timpul backtracking-ului. Eliminând toate asignările efectuate după nivelul identificat ca declanșator al unui conflict, solver-ul poate reveni eficient și continua procesul fără pierderi semnificative de performanță.
- **"Watchers"** Sistemul de "watchers" este o optimizare cheie pentru verificarea rapidă a stării clauzelor și pentru propagarea unitară.
  - Fiecare clauză are doi "watchers" care monitorizează o pereche de literali. Aceștia sunt actualizați numai atunci când unul dintre literalii monitorizați își schimbă valoarea, reducând numărul total de verificări necesare.
  - Această abordare eficientizează procesul de propagare, deoarece elimină necesitatea verificării complete a clauzei atunci când un literal este satisfăcut sau devine fals.
- **Baza de date a clauzelor** Baza de date a clauzelor stochează atât clauzele originale, cât și cele învățate în timpul procesului de rezolvare. Gestionarea eficientă a acestei baze de date este esențială pentru performanță, pentru alocarea de memorie și manipularea eficientă a spațiului disponibil.
  - Solver-ul asigură acces rapid la clauze, utilizând structuri de date optimizate care permit inserarea, ștergerea și modificarea clauzelor în timp redus.
  - Metodele *attachClause()*, *removeClause()* și *simplify()* sunt utilizate pentru gestionarea bazei de date. În special, *simplify()* elimină clauzele satisfăcute și simplifică structura bazei de date, reducând astfel complexitatea viitoarelor verificări.
- **Activitatea clauzelor și variabilelor** Sistemul de prioritizare al variabilelor și clauzelor optimizează selecția acestora pentru deciziile următoare, accelerând procesul de căutare.
  - **Variabile:** Activitatea variabilelor este măsurată și ajustată în funcție de frecvența lor de implicare în conflicte recente:
    * *varBumpActivity()*: crește activitatea unei variabile atunci când aceasta contribuie la un conflict, sporind șansele ca aceasta să fie selectată pentru decizie în iterațiile următoare.
    * *varDecayActivity()*: reduce activitatea tuturor variabilelor în mod gradual, astfel încât variabilele implicate recent în conflicte să primească prioritate în fața celor implicate mai demult.
  - **Clauze:** Similar variabilelor, clauzele beneficiază de un mecanism de ajustare a activității lor:
    * *claBumpActivity()*: crește activitatea unei clauze atunci când aceasta devine relevantă în determinarea unui conflict.
    * *claDecayActivity()*: reduce activitatea clauzelor învechite, menținând un echilibru între clauzele recent utilizate și cele mai vechi.

8 Concluzii

Până în această etapă a proiectului, am reușit să înțelegem ideea de bază a rezolvării problemelor SAT și modul în care pot fi utilizate SAT-solvers pentru a aborda astfel de probleme.

De asemenea, am explorat la nivel conceptual modul în care MiniSat implementează soluția, de la citirea formulelor în format DIMACS, până la utilizarea algoritmului CDCL pentru determinarea satisfiabilității. Această înțelegere ne oferă o bază pentru a avansa în proiect.

În etapa următoare, ne propunem să analizăm mai în profunzime algoritmul, să studiem în detaliu implementarea și să explorăm metode posibile de optimizare a performanțelor solverului, pentru a obține soluții mai eficiente.

9 Optimizări viitoare

10 Contribuția fiecărui membru

În calitate de echipă, primul pas a fost să stabilim o distincție clară între sarcinile pe care le vom aborda împreună și cele care vor fi alocate individual fiecărui membru al echipei.

În primul rând, sarcinile la care am lucrat împreună au fost următoarele:

-Instalarea MiniSat-ului: Fiecare a trebuit fie prin Windows, fie prin terminalul Ubuntu, să instaleze aceeași versiune a MiniSat-ului.

-Colaborarea pe GitHub: Pentru a colabora in cadrul acestui proiect am folosit GitHub. Benjámin a fost cel care a creat repository-ul proiectului, folosit pentru a posta și avea acces la rezultatele testelor care au rulat pe MiniSat.

-Rularea de teste: După cum am menționat la Instalarea MiniSat-ului, în prima etapă a proiectului, toți am rulat familii de teste.

În al doilea rând, trebuie menționat că multe dintre sarcinile pe care le-am avut au fost realizate individual. Acum, pe scurt, pentru a fi cât mai clar modul în care a contribuit fiecare, voi enumera numele fiecărui membru și sarcinile pe care le-a realizat:

- Anghel Bogdan
  - Am colaborat cu Stancu Maria la redactarea raportului, realizând capitolele: Contribuția Fiecărui Membru, Provocări Întâmpinate și Rezultate Experimentale.
  - Am introdus rezultatele de pe GitHub într-un tabel care prezinta outputul fiecărui test.
- Moise Alexandra
  - A analizat arhitectura MiniSat-ului, documentându-se din diverse surse disponibile online, apoi a realizat un set de explicații detaliate care au fost discutate cu întreaga echipă.
- Stancu Maria
  - A contribuit la redactarea raportului abordând secțiunile de Abstract, Introducere, Descrierea problemei și Concluzii.

  - A realizat diagrama de clase și explicatia acesteia.
- Szekrenyes Benjámin
  - A realizat partea de experimente și a reușit să ruleze până la capăt familia de teste: Software Verification, în ciuda dificultăților întâmpinate pe parcurs cu prima familie de teste Polynomial Multiplication.
  - Cu ajutorul rezultatelor încărcate pe GitHub, ne-a furnizat informațiile necesare pentru elaborarea secțiunilor de rezultate experimentale și provocări întâmpinate.

11 Link GitHub

`https://github.com/beni0104/proiectVF`

Bibliografie

[NE07] Niklas Sorensson Niklas Een. An extensible sat-solver, 2007.
[Neg08] Cosmin Negruşeri. Problema satisfiabilităţii formulelor logice, 2008.