Analizarea MiniSat in rezolvarea problemelor SAT

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) 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 (variabile propoziționale) și constrângeri (expresii logice formate din variabile propoziționale și operatori logici). 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 astfel încât să utilizeze resursele disponibile într-un mod cât mai 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

4.1 Analiza benchmark-urilor utilizate

Testele experimentale s-au concentrat pe două familii de benchmark-uri distincte: Polynomial Multiplication și Software Verification. Acestea au fost selectate pentru a evalua performanțele solverului MiniSat în condiții diferite de complexitate, atât din punct de vedere al numărului de variabile, cât și al naturii constrângerilor impuse. Rezultatele obținute și dificultățile specifice fiecărei familii sunt detaliate într-o manieră comparativă mai jos.

Benchmark-ul Polynomial Multiplication Această familie de teste implică o complexitate semnificativă datorită numărului mare de variabile generate în timpul multiplicării termenilor polinomiali.

În timpul rulării testelor, au fost identificate următoarele dificultăți:

- **Resurse insuficiente de memorie:** Două dintre testele din această familie au necesitări ridicate de memorie, depășind 8GB, ceea ce a condus la utilizarea unui swap file pentru a extinde memoria disponibilă până la 16GB. Chiar și așa, testele au eșuat cu eroarea "Process Killed" din cauza lipsei de resurse suficiente. Swap file este o metodă prin care putem aloca memorie de pe disk astfel încât solverul să aibă acces la mai multă memorie.
- **Durată lungă de execuție:** Rularea testelor a durat peste 27 de ore pentru două cazuri specifice, dar tot au eșuat cu eroarea "Process Killed" din cauza lipsei de memorie alocată (16GB în total).

Aceste limitări au influențat negativ capacitatea de a colecta rezultate complete pentru toate testele, de aceea am ales să continuăm cu o altă familie de benchmark-uri.

Benchmark-ul Software Verification Spre deosebire de Polynomial Multiplication, benchmark-ul Software Verification a fost mai accesibil din punct de vedere al complexității resurselor. Această familie a implicat un număr controlat de variabile și constrângeri, ceea ce a permis finalizarea tuturor testelor într-un interval de timp rezonabil.

4.2 Rezultate obținute

- Toate cele 15 teste din această familie au rulat cu succes, după cum se vede in tabela 1, fără a necesita extinderi suplimentare ale memoriei disponibile. Niciunul dintre teste nu a depășit o oră de execuție, un contrast puternic față de benchmark-ul anterior.
- Un aspect notabil este faptul că toate testele au avut rezultate "UNSAT", indicând că nu există soluții satisfiabile pentru constrângerile impuse. Această uniformitate a rezultatului evidențiază robustețea și eficiența algoritmilor MiniSat pentru acest tip de problemă.

Test Name	Memory	CPU Time (in seconds)	Result	Variables	Constraints
02f6	5824.00 MB	392.686 s	unsat	2705815	33181429
0d31	5385.00 MB	276.014 s	unsat	2487639	29784400
0f26	6090.00 MB	383.723 s	unsat	2931243	36758951
1e0d	4908.00 MB	252.987 s	unsat	2380624	28151394
25cc	5367.00 MB	1726.24 s	unsat	3275344	16977189
44fd	739.00 MB	1487.08 s	unsat	498723	2928183
573c	7617.00 MB	461.481 s	unsat	3345193	43459910
6008	7202.00 MB	392.071 s	unsat	3106374	39564424
878b	2495.00 MB	90.9715 s	unsat	1509852	15518860
aaa3	4783.00 MB	263.501 s	unsat	2330441	27359607
c03f	5772.00 MB	327.535 s	unsat	2650725	32315652
c21c	5877.00 MB	327.225 s	unsat	2762304	34061325
c8b1	7489.00 MB	418.657 s	unsat	3226168	41493467
e8ab	3779.00 MB	217.114 s	unsat	2072718	23530185
f54b	3635.00 MB	172.186 s	unsat	1925993	21375298

Tabela 1. Interpretare rezultate experimentale

5 Provocări întâmpinate

În timpul rulării experimentelor, am întâmpinat o serie de provocări, care au influențat atât desfășurarea testelor, cât și interpretarea rezultatelor. Aceste provocări au fost diferite în funcție de familia de benchmark-uri.

5.1 Dificultăți în cazul: Polynomial Multiplication

Această familie de teste a fost extrem de solicitantă din punct de vedere computațional. Problema principală a fost necesitatea unor resurse mari de memorie și timp de procesare. Pentru a rezolva această problemă, am configurat și utilizat un swap file, extinzând memoria disponibilă la 16GB. Cu toate acestea, două dintre teste s-au finalizat cu eroarea *Process Killed*.

O altă provocare majoră a fost timpul de execuție. Chiar și cu memoria extinsă, testele au rulat timp de peste 27 de ore înainte de a se opri cu eroarea de mai sus. Acest timp ridicat de execuție a limitat posibilitatea de a rula mai multe teste și a influențat decizia de a ne orienta spre familia **Software Verification**.

5.2 Provocări asociate cu: Software Verification

Această familie de teste a fost mai ușor de gestionat în ceea ce privește resursele. Spre deosebire de familia **Polynomial Multiplication**, nu a fost nevoie să extindem memoria sau să folosim swap files. Toate cele 15 teste (toate având rezultat UNSAT) au fost finalizate într-un timp rezonabil, de sub 6 ore.

5.3 Comparativ: Polynomial Multiplication vs. Software Verification

Privind în ansamblu, diferențele dintre cele două familii de benchmark-uri sunt semnificative. **Polynomial Multiplication** a pus accent pe testarea limitelor hardware, necesitând memorie extinsă și timp de procesare ridicat. Pe de altă parte, **Software Verification** a demonstrat că MiniSat se poate descurca eficient cu probleme de dimensiuni moderate, finalizând toate testele într-un timp scurt și oferind soluții clare.

Această comparație evidențiază importanța selectării corespunzătoare a benchmark-urilor în evaluarea performanței unui solver. Fiecare familie oferă o perspectivă diferită, iar folosirea ambelor oferă o imagine mai completă asupra capacității unui solver de a gestiona diverse tipuri de probleme.

6 Algoritmul CDCL (Conflict-Driven Clause Learning)

Algoritmul **CDCL** (Conflict-Driven Clause Learning) [Guo24] este o extensie a algoritmului clasic **DPLL** (Davis-Putnam-Logemann-Loveland) utilizat pentru rezolvarea problemelor de satisfiabilitate booleană (SAT). CDCL introduce tehnici avansate care îmbunătățesc semnificativ eficiența și permit rezolvarea unor probleme complexe mult mai rapid comparativ cu metoda originală.

6.1 Principii generale

CDCL se bazează pe fundamentele algoritmului DPLL, care include următoarele operații de bază:

- **Propagarea unitară**: Algoritmul identifică clauze unitare și determină valoarea literaliilor rămași pentru a satisface clauza respectivă.
- **Backtracking cronologic**: DPLL revine pas cu pas la deciziile anterioare pentru a explora alte ramuri ale spațiului de soluții, dacă o atribuire curentă cauzează un conflict.
- **Explorarea spațiului de soluții**: Se evaluează toate posibilitățile prin atribuire iterativă a valorilor variabilelor.

6.2 Extensiile CDCL

CDCL adaugă următoarele elemente-cheie care îmbunătățesc performanța algoritmului:

1. **Învățarea clauzelor (Clause Learning)**:
   - Atunci când apare un conflict, algoritmul analizează deciziile și propagările care au dus la acesta.
   - Se generează o *clauză învățată*, care interzice reapariția conflictului în viitor.
   - Această clauză este adăugată în baza de date a problemei, reducând spațiul de căutare.
2. **Backtracking non-cronologic**:
   - În loc să revină pas cu pas la deciziile anterioare, algoritmul sare direct la nivelul decizional relevant pentru conflictul identificat.
   - Acest mecanism accelerează rezolvarea prin eliminarea explorării inutile a unor ramuri din spațiul de soluții.
3. **Euristici de decizie**:
   - Alegerea următoarei variabile și a valorii acesteia se face pe baza unui scor de activitate.
   - Variabilele implicate frecvent în conflicte recente sunt prioritizate, ceea ce crește probabilitatea de a găsi soluții rapid.
4. **Gestionarea dinamică a clauzelor**:
   - Algoritmul menține o bază de date dinamică, eliminând clauzele inutile sau *„învechite"* pentru a optimiza utilizarea memoriei.
   - Clauzele relevante sunt actualizate constant pentru a reflecta starea curentă a problemei.

6.3 Fluxul algoritmului CDCL

1. Alegerea unei variabile neasignate folosind euristici (de exemplu, VSIDS - Variable State Independent Decaying Sum).
2. Propagarea unitară a literaliilor, aplicând constrângerile impuse de clauze.
3. Verificarea conflictelor:
   - Dacă apare un conflict, se analizează cauza acestuia și se generează o clauză învățată.
   - Se efectuează backtracking non-cronologic până la nivelul decizional adecvat.
4. Continuarea procesului până când:
   - Toate variabilele sunt asignate, indicând că problema este satisfiabilă (SAT).
   - Nu mai există decizii posibile și problema este nesatisfiabilă (UNSAT).

6.4 Avantaje față de DPLL

Avantajele pot fi vizualizate în tabelul 6.4 care prezintă analiza comparativă a celor doi algoritmi.

Caracteristică	DPLL	CDCL
Backtracking	Cronologic	Non-cronologic
Învățarea clauzelor	Nu	Da
Performanța	Limitată	Ridicată
Spațiul de căutare	Explorare completă	Reducere dinamică

Tabela 2. DPLL vs CDCL

6.5 Concluzie

Algoritmul CDCL reprezintă standardul modern pentru SAT solvers, datorită capacității sale de a gestiona eficient probleme complexe prin utilizarea învățării clauzelor și a backtracking-ului non-cronologic. Acesta construiește pe baza DPLL, adăugând optimizări esențiale care reduc semnificativ timpul de execuție și cerințele de memorie.
Procesul este, de asemenea, prezentat în figura 1.

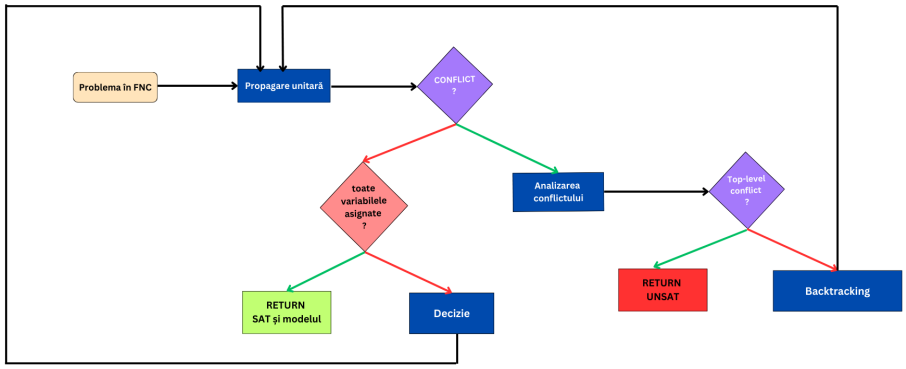

Figura 1. Schema algoritmului CDCL

7 Arhitectura MiniSat

7.1 Structura MiniSat

1. Fișiere CNF
   MiniSat folosește fișiere de input în format DIMACS CNF: [ES03]
   - 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: [ES03]
     - 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-ul MiniSat este bazat pe algoritmul CDCL și integrează două caracteristici principale: backtracking-ul la întâlnirea conflictelor și învățarea de clauze pentru optimizarea procesului de căutare. [BHvMW09]
     (a) *Categorii și concepte analizate*
     Reprezentarea problemei SAT se realizează prin clauze și asignări, acestea fiind structurile de bază folosite în procesul de rezolvare.

(b) *Deducere (din eng. "inference"): propagarea unitară*
Pentru fiecare literal din problemă, se păstrează liste de constrângeri care pot conduce la propagarea unitară prin asignarea literalului respectiv. Aceste liste sunt esențiale pentru a permite solver-ului să identifice rapid variabilele care pot fi propagate fără alte decizii.
În cazul clauzelor, se verifică dacă acestea sunt clauze unitare. Dacă sunt unitare, ele pot fi propagate direct, ceea ce simplifică procesul. Altfel, se selectează doi literali care sunt marcați ca observați (*"watched"*). Acest mecanism este gestionat printr-un obiect de tip *Watcher*, care stochează referința către clauză și celălalt literal din pereche. Obiectul *Watcher* este apoi adăugat în lista de *watchers* a literalului opus (literalul negat).
Sistemul de *watchers* aduce un avantaj important, deoarece elimină necesitatea ajustării listelor de *watchers* în timpul backtracking-ului.

(c) *Învățarea pe baza conflictelor*
Când are loc un conflict, ceea ce înseamnă că o constrângere devine imposibil de satisfăcut, solver-ul analizează combinația de variabile care a generat acest conflict. Variabilele implicate pot fi decizii directe (asignări făcute de solver) sau rezultate ale propagării unitare. Cauzele conflictului sunt urmărite pas cu pas, printr-un proces de backtracking aplicat asupra stivei de asignări (*"trail"*). Rezultatul acestui proces este identificarea unui set de variabile care a produs conflictul. Acestea sunt utilizate pentru a genera o nouă clauză care interzice asignarea ce a dus la conflict. Noua clauză, denumită *clauză învățată*, este adăugată în baza de date a problemei, contribuind astfel la evitarea repetării conflictului.

(d) Căutarea: Proces Iterativ
Procesul de căutare al MiniSat este iterativ și implică următorii pași:
   i. Alegerea unei variabile de decizie neatribuite.
   ii. Propagarea consecințelor asignării variabilelor:
      A. Toate variabilele asignate ca și consecință se află pe același „nivel de decizie".
      B. Interpretările sunt stocate pe o stivă de decizie (*trail*), împărțită pe nivele de decizie. Această stivă este utilizată pentru backtracking.
   iii. Repetarea procesului până când:
      A. Toate variabilele sunt asignate, ceea ce indică satisfiabilitatea problemei și determinarea unui model (SAT).
      B. Are loc un conflict, moment în care se apelează procedura de învățare.

(e) *Euristica de activitate*
Euristica de activitate optimizează selecția variabilelor și a clauzelor pe baza unui indicator de activitate:
- Variabile: Fiecare variabilă este ordonată pe baza unui indicator de activitate. Inițial, acest indicator are o valoare prestabilită (0.95). Valoarea indicatorului crește atunci când variabila apare

într-o clauză de conflict generată și scade după ce conflictul este înregistrat. Astfel, variabilele implicate în conflicte recente sunt prioritizate.

- Clauze: Sistemul folosește o abordare similară pentru clauze, prioritizând clauzele care au fost implicate în conflicte recente.

7.2 Diagrama de clase

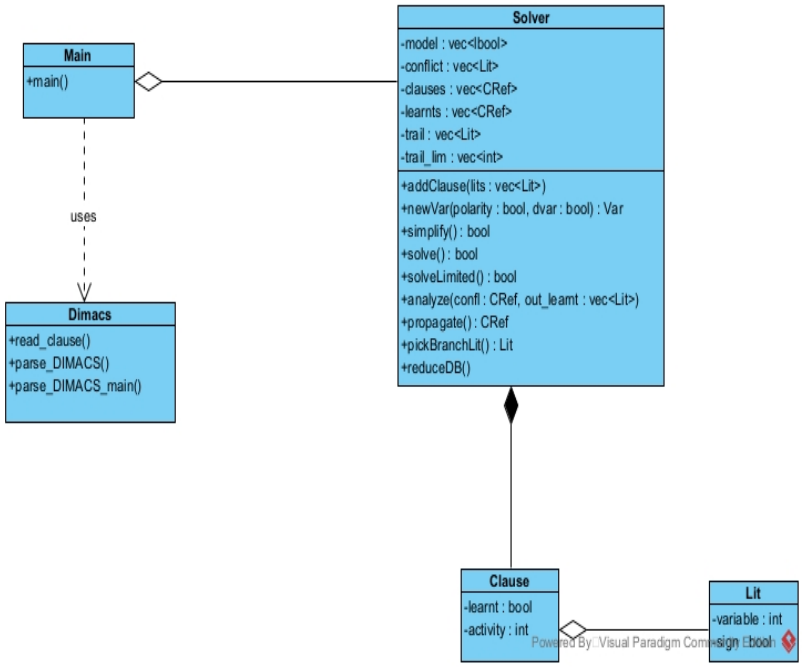

Figura 2. Diagrama de clase

Această diagramă de clase (Figura 2) prezintă arhitectura solverului MiniSat, detaliind componentele sale principale și relațiile dintre ele.

Clasa centrală este Solver, care conține majoritatea atributelor și metodelor necesare pentru procesul de rezolvare SAT. Atributele acesteia includ structuri de date esențiale, precum *model* (o listă de valori booleene pentru variabile), *conflict* (un vector de literali care formează conflicte), *clauses* și *learnts* (vectori de referințe la clauze normale și clauze învățate) și alte structuri precum *trail* și *trail_lim*. Metodele principale ale clasei includ: *addClause()* pentru adăugarea clauzelor, *newVar() pentru crearea de noi variabile*, și metode precum *simplify(), solve()*, și *propagate()* care implementează logica de bază a algoritmului.

Clasa Clause reprezintă clauzele individuale, având atribute precum learnt (indică dacă o clauză este învățată) și activity (o metrică folosită pentru prioritizarea clauzelor în rezolvarea SAT). De asemenea, există clasa Lit, care definește un literal prin atributele variable și sign, reprezentând variabila și semnul acesteia.

Clasa Dimacs este responsabilă pentru parsarea formulelor în format DIMACS, oferind metode precum read_clause() și parse_DIMACS() pentru citirea și interpretarea datelor de intrare. Clasa Main este punctul de intrare al programului, folosind funcțiile din Solver și Dimacs pentru a rezolva probleme SAT.

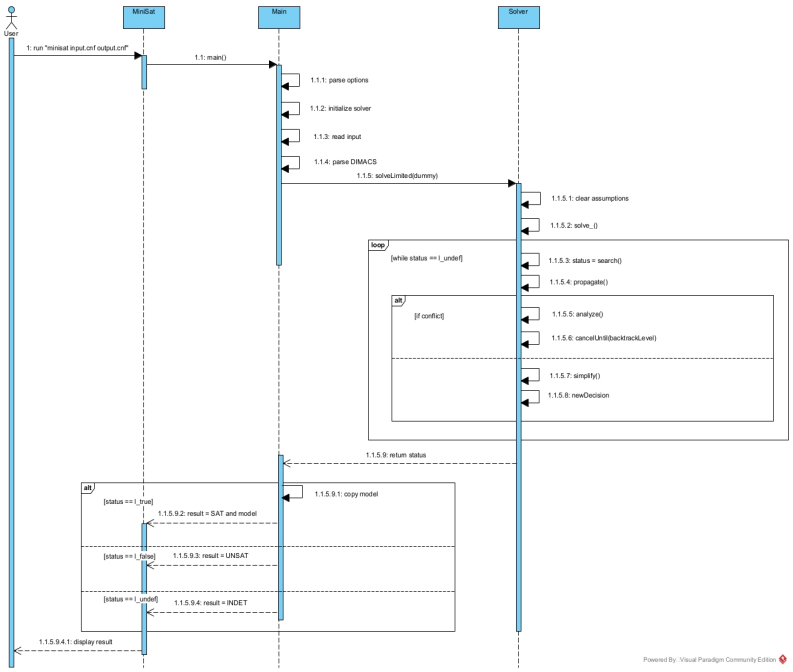

Figura 3. Diagrama de secvente

7.3 Diagrama de secvente

Această diagramă de secvențe (Figura 3) prezintă procesul de funcționare al MiniSat pentru rezolvarea problemelor de satisfiabilitate booleană (SAT). Diagrama urmărește interacțiunile dintre utilizator, modulul principal (Main) și componenta Solver, evidențiind fiecare pas al procesului și alternările logice implicate.

Procesul începe atunci când utilizatorul execută comanda `minisat input.cnf output.cnf`. Opțional, se poate adăuga `> result.stats 2&>1` pentru a redirecționa

fluxurile de ieșire standard și eroare standard ale aplicației către un fișier separat, `result.stats`, care va conține statistici detaliate despre rularea solver-ului. Aceasta specifică fișierul de intrare, care conține problema SAT în format DIMACS, și fișierul de ieșire unde va fi stocat rezultatul. Execuția declanșează metoda main() din modulul principal al MiniSat. Modulul Main parcurge mai multe etape inițiale: analizează opțiunile din linia de comandă, inițializează solverul, citește problema SAT din fișierul de intrare și o parsează în format DIMACS. După această pregătire, solverul este apelat pentru a rezolva problema prin metoda solveLimited().

Solverul începe prin curățarea presupunerilor anterioare și rezolvarea problemei inițiale. În cadrul unui ciclu principal, solverul verifică starea curentă a problemei. Dacă problema este încă nerezolvată (status == l_undef), se alternează între etapele de căutare, propagare și gestionarea conflictelor. În cazul unui conflict, solverul analizează cauza, anulează deciziile până la un nivel de backtracking (cel al deciziei care a produs conflictul) și simplifică problema pentru a continua căutarea. Dacă nu există conflicte, solverul ia decizii noi pentru a avansa în procesul de rezolvare.

Când solverul determină o stare finală, rezultatul este interpretat și returnat către modulul Main. Dacă problema este satisfiabilă (status == l_true), se construiește un model SAT și se copiază soluția. Dacă problema este nesatisfiabilă (status == l_false), rezultatul este UNSAT. În cazuri în care solverul nu poate determina un rezultat clar, statusul rămâne INDET (nedeterminat).

7.4 Analiza metodelor principale

Metodele principale ale MiniSat sunt fundamentale pentru funcționarea eficientă a solver-ului. Întrucât CDCL se bazează pe DPLL, adăugând îmbunătățiri esențiale, metodele implementate pot fi privite din două perspective:

1. Deciziile și propagarea unitară, care formează baza algoritmului
   - *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.
2. Învățarea clauzelor prin analiza conflictelor

- *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.

Nu în ultimul rând, metoda *search()*: căutarea modelului. Aceasta 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.

7.5 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" reprezintă o optimizare esențială în MiniSat, având ca scop accelerarea procesului de verificare a stării clauzelor și de propagare unitară. Acest mecanism ajută la reducerea numărului de verificări necesare în timpul procesului de propagare.
  - Fiecare clauză are asociate două *"watchers"* care monitorizează o pereche de literali. Acești literali sunt selectați pentru a verifica starea clauzei atunci când un literal din pereche își schimbă valoarea.
  - Atunci când un literal monitorizat devine fals, se verifică restul clauzei pentru a determina dacă poate fi propagat un alt literal. Dacă clauza devine unitară (toți ceilalți literali sunt satisfăcuți sau falși), valoarea literalului rămas este determinată automat.
  - Principala utilitate a acestui sistem este eficientizarea procesului de propagare. Datorită faptului că watchers sunt actualizați doar atunci când un literal se modifică, nu este necesar să se verifice întreaga clauză pentru fiecare schimbare, ci doar cei doi literali monitorizați. Astfel, clauzele sunt procesate mai rapid, iar algoritmul beneficiază de o performanță îmbunătățită.
- **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

MiniSat a demonstrat eficiență în rezolvarea problemelor SAT de complexitate moderată, dar întâmpină dificultăți majore la benchmark-uri complexe precum Polynomial Multiplication. Acestea au necesitat resurse semnificative de memorie și timp, ducând la eșecuri pentru unele teste. În schimb, benchmark-urile mai simple, cum ar fi Software Verification, au fost gestionate eficient, toate testele fiind finalizate într-un timp rezonabil.

Algoritmul CDCL reprezintă un punct forte al MiniSat, îmbunătățind performanța prin învățarea clauzelor și backtracking non-cronologic. Aceste tehnici reduc spațiul de căutare și gestionează conflictele rapid, făcând MiniSat potrivit pentru probleme complexe de dimensiuni medii.

Arhitectura MiniSat, bazată pe structuri optimizate precum „trail" și „watchers", permite o gestionare eficientă a resurselor. Totuși, extinderea funcționalității sale ar putea beneficia de optimizări suplimentare pentru a aborda mai bine problemele foarte mari și complexe.

9 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
  - A participat la redactarea raportului, realizând capitolele: Contribuția Fiecărui Membru, Provocări Întâmpinate și Rezultate Experimentale.
  - A introdus rezultatele de pe GitHub într-un tabel care prezinta output-ul 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.
  - A colaborat cu Maria la realizarea diagramelor.

- Stancu Maria
  - A contribuit la redactarea raportului abordând secțiunile de Abstract, Introducere, Descrierea problemei, Algoritmul CDCL, Arhitectura Minisat și Concluzii.
  - A realizat diagrama de clase și cea de secvențe împreună cu explicațiile acestora.
- 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.

10 Link GitHub

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

Bibliografie

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[Guo24] Mengyu Guo. A review of research on algorithms for solving sat problems. *Mathematical Modeling and Algorithm Application*, 2(1):6–10, 2024.

[Neg08] Cosmin Negruşeri. Problema satisfiabilităţii formulelor logice, 2008.