Verificare Formala

Haiosta Michelle, Cerean Bogdan-Ioan, Popovici Adrian-Robert, and Bosna Marinel

West University of Timisoara

Abstract

Lucrarea prezinta o introducere scurta in probleme de tip SAT si analizeaza SAT Solverul MiniSat, prezentant modul de instalare a programului, o analiza asupra codului si posibile modalitati de a imbunatatii acest program

Keywords: MiniSat · SAT Solver · Logica propozitionala.

1 Introducere

In aceasta lucrare este prezentat SAT Solverul MiniSat , un program folosit pentru a rezolva probleme de verificare formala cat si in competitii de rezolvare a problemelor SAT. Lucrarea este impartita in 6 capitole care adreseaza aspectele acestei aplicatii:

1. Primul capitol este o prezentare a MiniSat-ului si o introducere a problemelor de tip SAT.
2. Al doilea capitol este o explicatie a procesului de instalare si pregatire a MiniSat pentru a-l putea folosi in practica.
3. Al treilea capitol descrie modul de folosire a aplicatiei pentru a rula un benchmark, un set de probleme folosite in competitii SAT pentru a testa eficienta unui SAT Solver.
4. Al patrulea capitol confera o analiza asupra codului si logici programului MiniSat, incluzand o diagrama care arata pas cu pas ordinea apelarii functiilor in program si o descriere a fiecarei metode in parte.
5. Al cincilea capitol discuta posibile metode de a imbunatatii SAT Solver-ul, tema competitiilor anuale de SAT unde mai multe echipe prezinta algoritmi modificati sau programe originale de rezolvare a problemelor de tip SAT.
6. Ultimul capitol, capitolul 6, este o incheiere in care sunt discutate dificultati intalnite in timpul compuneri acestei lucrari.

2 Descrierea problemei

Problemele de satisfiabilitate , cunoscute si ca probleme SAT, sunt probleme in care analizam o propozitie logica, formata din una sau mai multe clauze aflate in conjunctie. Fiecare clauza contine literali aflati in disjunctie. Un literal reprezinta fie o variabil fie o negare de variabila. Acest format, in care avem doar conjunctii, disjunctii si negari este cunoscut ca Forma Normal Conjunctiva sau CNF.

In urma analizei , care implica acordarea de valori True sau False variabilelor propozitiei in diferite combinatii, ne asteptam sa gasim o combinatie de variabile care satisface fiecare clauza a propozitiei. Daca fiecare clauza este satisfacuta, propozitia finala este adevarata cea ce inseamna ca aceasta problema este satisfiabila, sau SAT. Daca nu exista o combinatie de variabile in care propozitia sa fie adevarata, adica daca nu avem o solutie a problemei, ea este considerata nesatisfiabila sau UNSAT.

Problemele de tip SAT sunt folosite in domenii cum ar fi inteligenta artificiala, proiectarea de circuite si Automated Theory Proving, ATP. In aceste domenii pot aparea probleme care au zeci de mii de variabile si sute de mii de simboluri. In rezolvarea manuala apar dificultati chiar si cand o problema are doar zece variabile iar complexitatea si timpul necesar pentru a rezolva o problema SAT ar fi enorme chiar si cand aplicam algoritmi cum ar fi CDCL sau DPLL. Din

acest motiv programe de tip SAT Solver, care pot analiza probleme SAT in mod automat si rapid, au fost dezvoltate si inca sunt folosite in prezent. Domeniul problemelor SAT este foarte studiat, existand concursuri anuale in care SAT Solvere sunt puse in competitie pentru a vedea care ruleaza mai eficient pe seturi de date enorme numite benchmark-uri.

MiniSat este un astfel de Sat Solver, gratuit si open-source dezvoltat de Niklas Een si Nikita Sorensson in 2003 si publicat in 2004 folosit atat academic cat si in industrie. Acest program a fost dezvoltat ca fiind un SAT Solver extensibil, care sa poata fi modificat si adaptat la diverse domenii mult mai rapid comparabil cu alte Solvere contemporane sau alternativa de a crea un Solver nou de la zero. MiniSat ruleaza in Linux si instalarea sa este prezentata in capitolul urmator

3 Instalarea MiniSat

Pentru a rula MiniSat, este necesar sa folosim o distributie Linux. Putem folosi o masinarie virtuala pentru a rula MiniSat, dar o metoda mult mai simpla este „Windows Subsystem for Linux", abreviat WSL, care ne permite sa rulam un environment Linux fara a folosi o masinarie virtuala sau paralel booting.

Instalarea WSL se poate executa direct din PowerShell sau Windows Command Prompt, folosind comanda „WSL –install" care, dupa repornirea sistemului, va instala pe calculator o distributie Ubuntu. Daca dorim sa folosim alta distributie, este suficient sa folosim comanda: `WSL -install <Numele distributiei>`

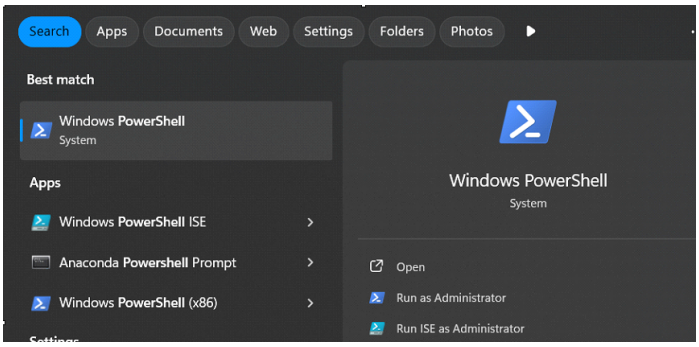


Fig. 1. Fereastra selectare WindowsPowerShell

Odata ce am instalat WSL si am repornit calculatorul, putem folosi aplicatia WSL pentru a accesa distributia Ubuntu pe care o avem. Odata ce accesam linia de comanda, fie folosind WSL, fie o masina virtuala, MiniSat poate fi instalat folosind managerul de pachete Linux. Putem actualiza managerul de pachete folosind comanda `sudo apt update`, dupa care instalarea MiniSat necesita o singura comanda: `sudo apt install minisat`

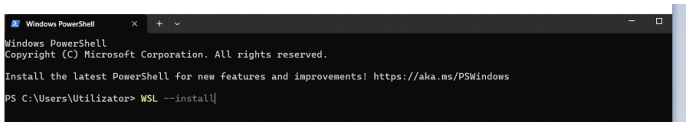
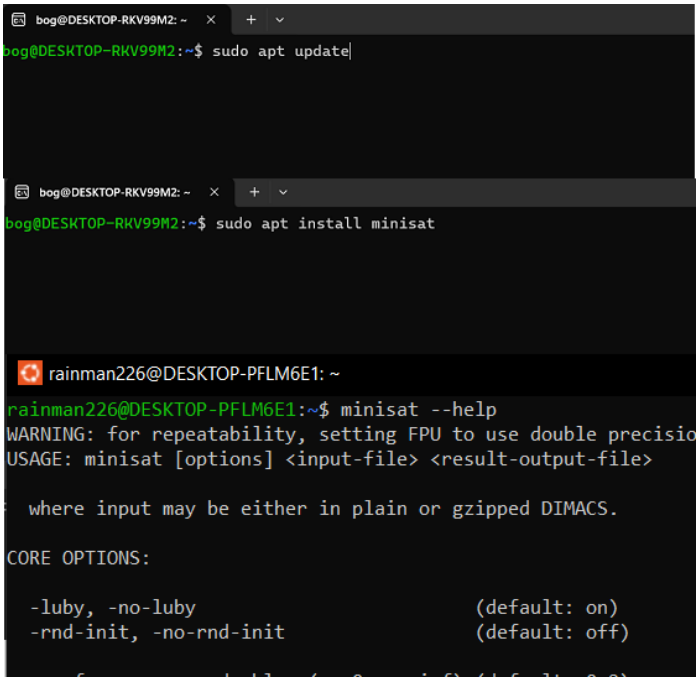


Fig. 2. Instalare WSL



4 Rularea benchmark

MiniSat este un SAT solver simplu, dar foarte eficient. Pentru a intelege mai bine cum functioneaza si cat de bine face fata problemelor complexe, am rulat un benchmark din competitia SAT pentru a analiza performanta si comportamentul sau.

Competiția SAT este un concurs international in care cei mai buni solvers SAT sunt evaluati pe un set de benchmark-uri standardizate, pentru a testa eficienta si corectitudinea algoritmilor. Aceste benchmark-uri sunt gandite pentru a acoperi o varietate de probleme complexe si pentru a stimula dezvoltarea unor solutii mai rapide si mai eficiente.

In cadrul acestui test, am folosit un benchmark bazat pe "Simon Cipher" — o schema criptografica de tip block cipher, care utilizeaza o structura echilibrata de tip Feistel si operatii pe blocuri de biti folosind porti logice AND si XOR. Problema a fost simplificata prin reducerea numarului de runde si folosirea unor

stari initiale prestabilite, ceea ce faciliteaza transformarea problemei intr-o formula SAT. Benchmark-ul include diverse instante cu blocuri de biti si runde variabile, concepute pentru a fi moderate ca dificultate.

Pentru acest test, am avut la dispozitie 10 fisiere de benchmark, pe care le-am rulat intr-un interval total de 24 de ore, cu un timp limitat de 8460 secunde per fisier (miniSat -cpu-lim=8640 "*benchmark*""output"). Experimentele au fost realizate pe un calculator echipat cu un procesor AMD Ryzen 5 4600H si 8 GB RAM, resurse suficiente pentru a monitoriza performanta MiniSat pe aceste benchmark-uri complexe.

Pentru a evalua performanta MiniSat pe setul de benchmark-uri, am analizat log-urile generate de solver, concentrandu-ne pe unele statistici. Datele colectate pot fi impartite in doua parti : statisticile problemei si statisticile rezolvarii. Statisticile problemei ofera informatii despre configuratia initiala a fiecarei instante SAT, in timp ce statisticile rezolvarii urmeaza progresul solverului in timpul procesului de rezolvare.

Urmatorii parametri reprezinta caracteristicile de baza ale instantei SAT analizate:

- **Variables**: numarul total de variabile din formula SAT, care indicand complexitatea problemei si dimensiunea spatiului de cautare a solutiei.
- **Clauses**: Reprezinta numarul total de clauze din formula SAT, ceea ce determina cate relatii logice exista intre variabile. Cu cat sunt mai multe clauze, cu atat problema devine mai complexa si dificila.

Iar statisticile relevante rezolvarii:

- **Restarts**: Restarturile sunt strategii de reinnoire a procesului de cautare pentru a evita blocajele in solutionare si pentru a explora mai eficient spatiul solutiilor.
- **Conflicts**: Numarul total de conflicte intampinate. Conflictele apar atunci cand solverul identifica o contradictie in setul curent de alocari, ceea ce forteaza revenirea (backtracking) si ajustarea traseului de cautare.
- **Decisions**: Numarul total de decizii facute. Reprezinta alegerile pe care solverul le face pentru a atribui valori variabilelor. Numarul ridicat de decizii reflecta o cautare extinsa a solutiei.
- **Propagations**: Numarul de propagari efectuate. Pasi intermediari prin care o decizie influenteaza alte variabile, reducand spatiul posibil al solutiilor si avansand solutionarea.
- **Conflict literals**: Numarul total de literali conflictuali. Sunt literali care au condus la conflicte si care pot fi eliminati pentru a simplifica problema.
- **Memory used**: Memoria utilizata de solver in timpul executiei.
- **CPU time**: Timpul total de procesare alocat executiei.
- **Result**: Rezultatul final al solverului.

Nume	Variabile	Clauze	Restarturi	Conflicte	Decizii	Propagări	Conflict literals	Memory used	CPU Time	Result
simon-r16-1	2688	8768	167930	138170998	158228770	24237526879	5685755929	112.00 MB	8639.73 s	INDET
simon-r18	3008	9856	89623	68282104	74031332	26373262692	3444985236	90.00 MB	6782.97 s	INDET
simon-r19	3168	10400	99258	77457500	84090316	34167417356	4086868821	90.00 MB	8641.55 s	INDET
simon-r24	3968	13120	94203	71501545	78398286	45866751062	3335259956	125.00 MB	8758.36 s	INDET
simon-r17	2848	9312	131071	105273474	116555843	28292265813	4632609162	104.00 MB	8812.92 s	INDET
simon-r21	3488	11488	98302	76070974	83292761	40060287405	4081479941	110.00 MB	9542.12 s	INDET
simon-r20-0	3328	10944	106490	81624815	89009973	40151260779	4422187495	117.00 MB	9593.14 s	INDET
simon-r23	3808	12576	98302	76372129	83845930	45533419860	3819277711	126.00 MB	9596.26 s	INDET
simon-r22	3648	12032	98302	76621624	83912970	43235900085	3965226602	122.00 MB	9231.1 s	INDET
imon-r25	4128	13664	90938	69626160	76191213	48057983031	3081086325	134.00 MB	8639.62 s	INDET

Rezultatele obtinute arata ca MiniSat a fost capabil sa proceseze probleme complexe in limitele de timp si memorie specificate, dar cu toate acestea, nu a reusit sa determine solutii satisfiabile in limita de timp alocata pentru majoritatea benchmark-urilor, indicand un rezultat *INDETERMINATE*. Numarul ridicat de conflicte si decizii sugereaza ca problema a fost dificila de rezolvat, necesitand multiple restarturi si propagari pentru a avansa in cautarea solutiei.

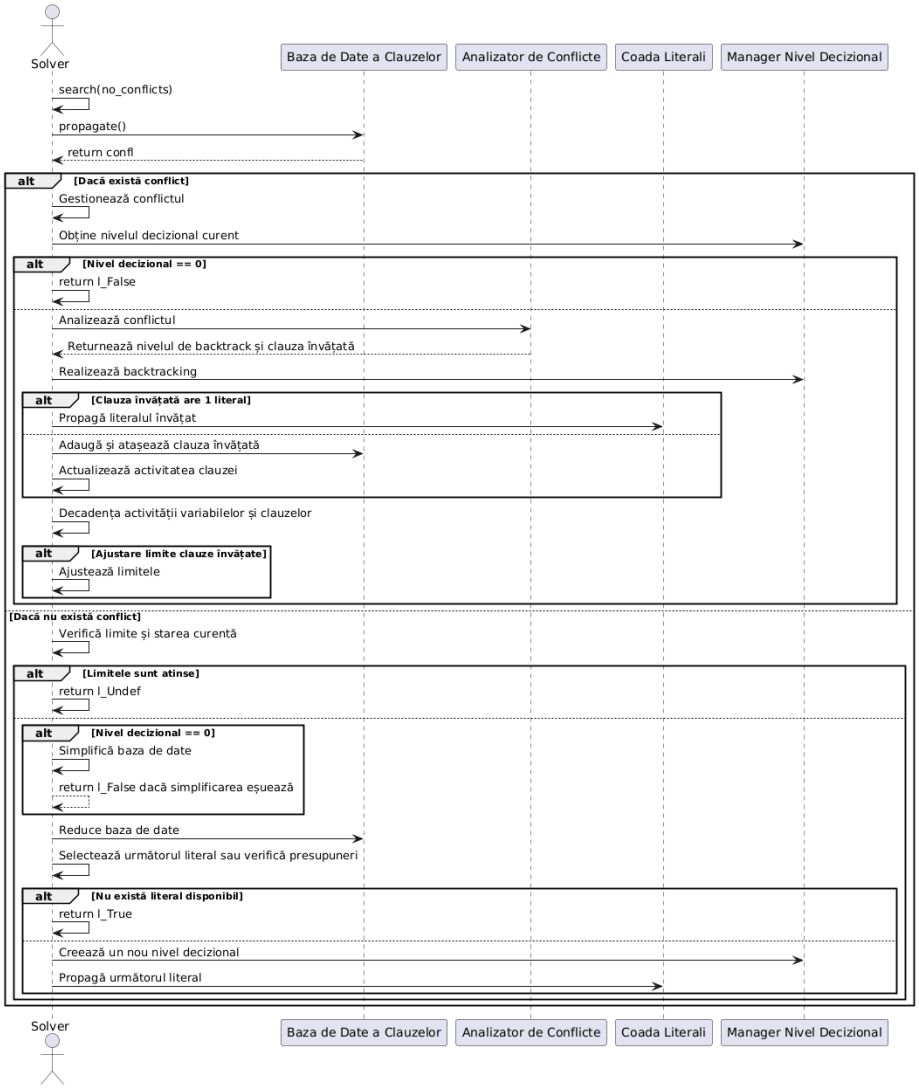
Totusi utilizarea relativ scazuta a memoriei sugereaza o eficienta buna a algoritmului de cautare in utilizarea resurselor. Dar, timpul de procesare total a fost aproape de limita alocata de 8640 de secunde, ceea ce indica necesitatea unor algoritmi de cautare mai rapizi sau a unei optimizari suplimentare pentru a rezolva aceste instante de mari dimensiuni in mod eficient.

5 Analiza Cod

In imaginea de mai jos este prezentata o schita a algoritmului de search din codul MiniSat, prezentata sub forma de pseudo-cod. Imaginea ne va ajuta sa intelegem pasii principali pe care acest algoritm ii urmeaza pentru a incerca, verifica si invata din greseli pana cand se gaseste o solutie sau pana cand se demonstreaza ca nu exista nici o solutie.

```
loop
    propagate()     – propagate unit clauses
    if not conflict then
        if all variables assigned then
            return SATISFIABLE
        else
            decide()     – pick a new variable and assign it
    else
        analyze()     – analyze conflict and add a conflict clause
        if top-level conflict found then
            return UNSATISFIABLE
        else
            backtrack()     – undo assignments until conflict clause is unit
```

Pentru a intelege mai in detaliu flux-ul acestui algoritm am creat diagrama de secvente de mai jos unde avem actorul principal(Solverul) si restul componentelor participante la algoritmul de search: Baza de date a clauzelor, Analizatorul de conflicte, Coada de literali si Managerul de nivel decizional:



Am observat, uitandu-ne la codul proiectului MiniSat, ca acesta se foloseste de algoritmul CDCL, iar in fisierul solver.cc din directorul core, acest algoritm este bine evidentiat prin metoda search. Pentru a intelege mai bine cum functioneaza **search()**, vom descrie fiecare etapa importanta din acest proces.

Aceasta metoda exploreaza diferite combinatii de valori pentru variabilele din formula pentru a determina daca exista o solutie care sa satisfaca toate clauzele sau, in caz contrar, ca formula este insatisfiabila. Algoritmul urmeaza mai multi pasi, iar fiecare componenta joaca un rol esential in procesul de cautare.

5.1 Initializare:

La inceput, **search()** initializeaza mai multe variabile de lucru, precum **conflicts**, care este contorul de conflicte, si **learnt_clause**, folosita pentru stocarea clauzei conflictuale curente in timpul analizei.

Cazul special este daca nu exista clauze de la inceput, algoritmul concluzioneaza ca formula este satisfiabila deoarece o formula fara clauze este implicit satisfacuta. In acest caz, metoda **search()** returneaza o solutie imediat, evitand astfel executia restului algoritmului.

5.2 Propagare (functia propagate()):

Propagarea este urmatorul pas in proces, aplicata pentru a deduce valori pentru variabile pe baza clauzelor existente. **propagate()** incearca sa seteze valori noi pentru variabile, verificand in acelasi timp daca aceste setari satisfac clauzele deja existente. Propagarea unitara este o tehnica esentiala folosita in metoda pentru a determina valoarea variabilelor.

Cazul special este daca **propagate()** detecteaza un conflict (adica o clauza are toate literalele false), metoda returneaza o referinta la acea clauza conflictuala. In acest caz, **search()** trece la analiza conflictului pentru a invata o clauza noua care sa evite conflictele similare in viitor.

5.3 Gestionarea conflictului:

Cand **propagate()** returneaza o clauza conflictuala, **search()** apeleaza **analyze()** pentru a determina cauza conflictului. In acest pas, **analyze()** examineaza variabilele implicate si construieste o noua clauza numita clauza de conflict. Aceasta este adaugata la baza de date de clauze si este destinata sa previna conflictele similare in viitor.

Cazul special este daca conflictul apare la nivelul de decizie 0, adica la nivelul de baza al algoritmului, **search()** concluzioneaza ca formula este insatisfiabila. In aceasta situatie, functia se opreste si returneaza UNSAT, deoarece nu exista nicio configuratie a variabilelor care sa satisfaca formula.

5.4 Invatarea clauzei de conflict si backtracking-ul (functia analyze()):

Functia **analyze()** parcurge clauza conflictuala(**conflict_clause**) si variabilele aferente pentru a construi o clauza noua care sa impiedice conflicte similare. In acest mod, solverul CDCL devine mai eficient pe masura ce isi „aminteste" conflictele trecute.

Cazul special este dupa crearea clauzei de conflict, **search()** stabileste la ce nivel de decizie sa revina(backtracking) pentru a evita conflictul. Aceasta revenire la un nivel anterior de decizie permite algoritmului sa incerce un alt set de decizii, explorind astfel o alta parte a spatiului solutiilor.

5.5 Curatarea bazei de date de clauze (functia reduceDB()):

Pentru a evita acumularea de clauze care incetinesc algoritmul, **search()** apeleaza **reduceDB()** la intervale regulate. Functia **reduceDB()** sterge clauzele cu activitate scazuta, adica acelea care nu contribuie semnificativ la prevenirea conflictelor. **Cazul special** Daca baza de date de clauze devine prea mare, **reduceDB()** elimina clauzele mai putin relevante, asigurand o performanta optima pe termen lung. Acest proces ajuta algoritmul sa fie mai eficient in gestionarea memoriei si a timpului de executie.

5.6 Decizie pentru variabila (functia pickBranchLit()):

Daca nu sunt gasite conflicte in propagare si nu au fost setate toate variabilele, **search()** selecteaza o variabila de decizie folosind **pickBranchLit()**. Aceasta functie se bazeaza pe activitatea variabilelor(cele implicate frecvent in conflicte recente), masurata in functie de frecventa cu care variabilele au fost implicate in conflicte recente. Variabilele cu activitate ridicata sunt prioritizate pentru a spori eficienta cautarii. Setarea acestei variabile contribuie la avansarea in cautare si la explorarea spatiului solutiilor. **Cazul special** daca toate variabilele au fost setate fara a aparea conflicte, **search()** returneaza SAT, indicand ca formula este satisfiabila.

5.7 Restarturi:

Daca numarul de conflicte depaseste un anumit prag, **search()** efectueaza un restart. Restartul consta in revenirea la nivelul de decizie initial (0) si inceperea unei noi explorari a spatiului solutiilor. Toate clauzele de conflict invatate sunt pastrate, astfel incit algoritmul sa foloseasca aceste informatii in cautarea ulterioara. Restarturile sunt o strategie pentru a evita blocarea intr-o regiune specifica a cautarii si pentru a explora noi rute de solutii.

Cazul special daca numarul de conflicte depaseste un prag stabilit, algoritmul executa un restart si revine la nivelul de decizie 0, reluind cautarea, dar pastrand clauzele conflictuale invatate.

5.8 Finalizarea cautarii:

Daca **search()** gaseste o setare a variabilelor care satisface toate clauzele, algoritmul returneaza **l_True**, indicind ca formula este SAT. Pe de alta parte, daca la nivelul de decizie 0 apare un conflict si nu mai exista alte cai de explorat, algoritmul conchide ca formula este UNSAT si returneaza **l_False**.

6 Propuneri de imbunatatiri pentru metodele MiniSat

MiniSat este un solver puternic pentru problema SAT, dar ca orice algoritm, poate fi optimizat. In continuare sunt prezentate cateva directii posibile de imbunatatire.

6.1 Invatarea clauzelor conflictuale

In metoda **analyze()**, MiniSat invata clauze noi din conflictele pe care le intalneste, astfel incat sa evite conflictele similare in viitor. Acest proces functioneaza foarte bine, dar, pe masura ce solverul ruleaza, baza de date de clauze devine foarte mare, ceea ce poate incetini algoritmul.

O posibila imbunatatire ar fi introducerea unei analize mai detaliate a clauzelor invatate. De exemplu, daca o clauza nu a fost utilizata intr-o propagare recent, inseamna ca este probabil inutila si poate fi eliminata. In plus, putem optimiza clauzele invatate reducand numarul de literali din ele. Clauzele mai compacte sunt mai rapide de utilizat in propagare, dar trebuie sa avem grija sa nu pierdem informatii importante.

6.2 Curatarea bazei de date de clauze

Metoda reduceDB() elimina periodic clauzele invatate care nu mai sunt utile, dar procesul actual se bazeaza pe reguli fixe. De exemplu, clauzele cu activitate scazuta sunt sterse automat, indiferent daca mai sunt relevante sau nu.

Propunem o metoda mai dinamica de curatare a bazei de date. In loc sa stergem toate clauzele sub un anumit prag de activitate, putem evalua cat de des au fost utilizate recent in propagare. Clauzele care nu au fost folosite deloc in ultimele iteratii ar putea fi sterse mai intai. De asemenea, putem curata doar o parte din baza de date la fiecare iteratie, pentru a evita incetinirea solverului.

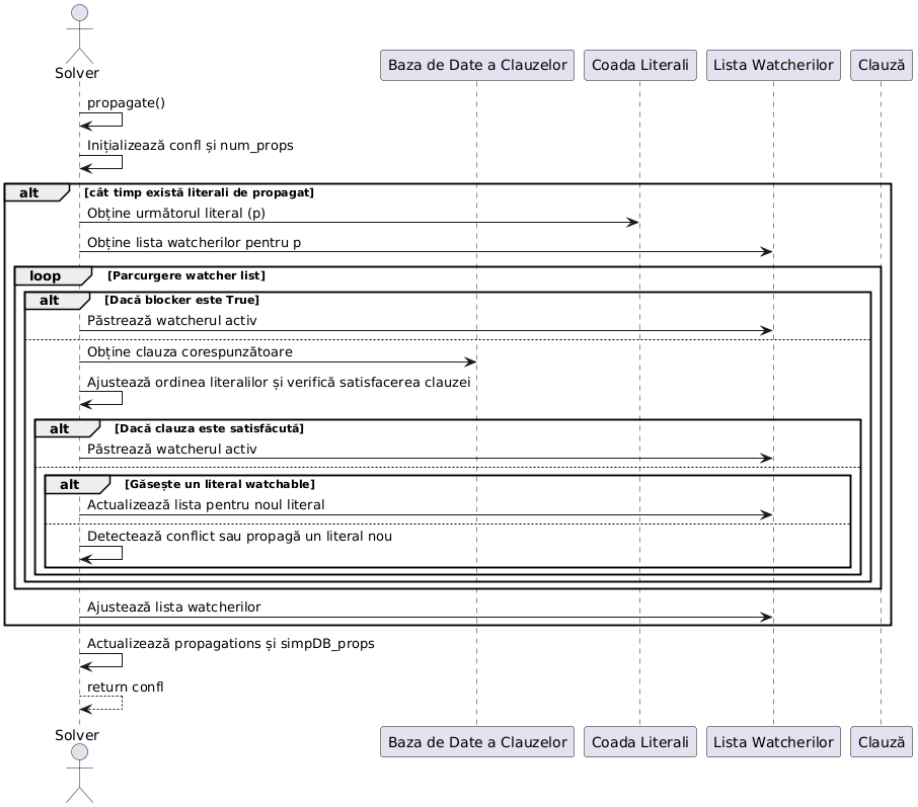
6.3 Optimizarea propagarii

Propagarea unitara este unul dintre cele mai importante procese din algoritmul MiniSat, dar si unul dintre cele mai costisitoare din punct de vedere al timpului de executie. In timpul propagarii, solverul verifica clauzele pentru a vedea daca setarile actuale ale variabilelor determina anumiti literali sa fie adevarati sau falsi. Pentru a face acest lucru eficient, MiniSat foloseste liste „watch” pentru literali, un mecanism care urmareste doar doi literali din fiecare clauza. Totusi, acest proces poate fi optimizat prin imbunatatirea structurii listei „watch”.

In forma actuala, listele „watch” sunt implementate astfel incat fiecare literal sa fie asociat cu un set de clauze in care este urmarit. Acest lucru functioneaza bine pentru probleme mai mici, dar pentru formule mari, cu milioane de clauze si variabile, accesarea si actualizarea acestor liste poate deveni lenta. De exemplu, de fiecare data cand un literal devine fals, solverul trebuie sa parcurga clauzele asociate acestuia pentru a gasi un alt literal valid care sa fie in watch. Acest proces poate fi optimizat prin folosirea unor structuri de date mai compacte. In loc sa stocam aceste liste ca vectori obisnuiti, am putea folosi o structura mai eficienta, cum ar fi un hash map, unde fiecare literal este o cheie, iar valorile sunt clauzele asociate. O astfel de abordare ar reduce timpul necesar pentru accesarea clauzelor, mai ales in problemele mari.

7 Prezentarea metodelor cheie din minisat prin diagrame uml

7.1 propagate()



Funcția propagate() este una dintre cele mai importante componente ale MiniSat, responsabilă pentru realizarea propagării unitare și pentru identificarea conflictelor. Această funcție parcurge coada de propagare, care conține literalii ce trebuie procesați. Pe baza acestor literali, se examinează clauzele asociate pentru a decide dacă există deducții noi ce pot fi făcute sau dacă a apărut un conflict. Mai jos este o descriere detaliată, explicată cu propoziții. Inițializarea procesului

Când funcția propagate() este apelată, se inițializează variabilele necesare pentru proces. Variabila confl este setată inițial la NULL, ceea ce indică faptul că nu există conflicte la începutul propagării. În același timp, num_props este utilizată pentru a contoriza numărul de literali procesați pe parcursul execuției funcției.

Coada de propagare, care conține literalii ce trebuie evaluați, este preluată pentru a începe procesul de propagare. Această coadă reprezintă punctul central al propagării unitare.

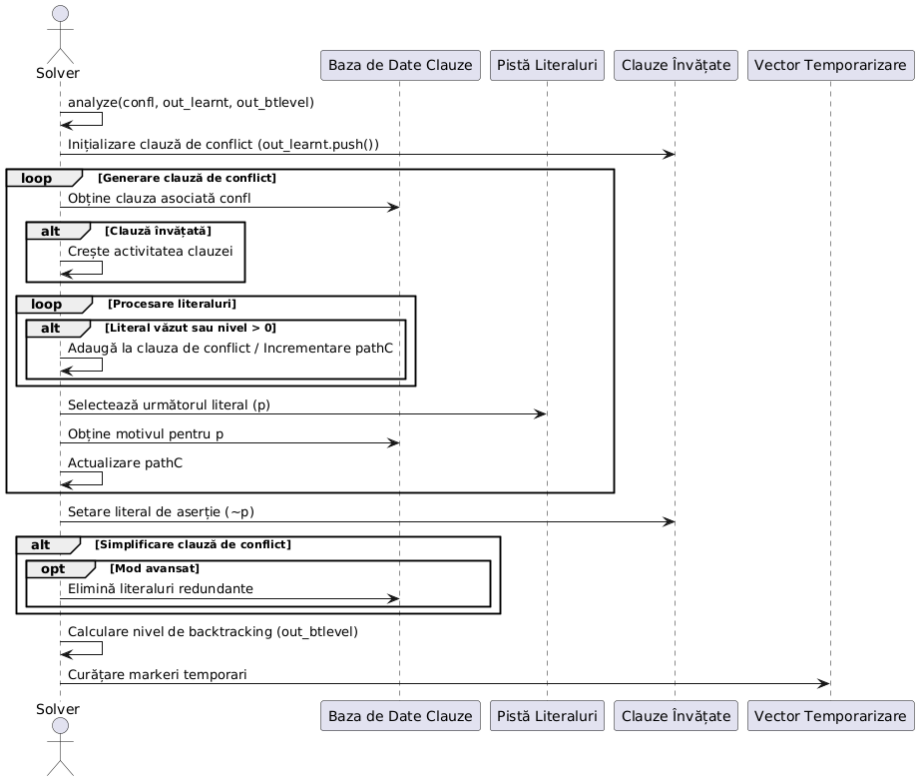
Procesarea cozii de propagare Funcția începe să proceseze coada de propagare iterativ. Pe rând, fiecare literal din coadă este extras și procesat. Pentru fiecare literal, se obține lista de „watchers" (supraveghetori) asociați cu acesta. Watcherii sunt structuri care indică clauzele ce trebuie verificate atunci când un literal este modificat.

Analiza clauzelor asociate Pentru fiecare literal extras, se parcurge lista clauzelor asociate, verificând starea fiecărei clauze:

Dacă clauza este satisfăcută deja, watcherul asociat este păstrat activ. Acest lucru înseamnă că nu este necesar să se facă modificări în această clauză, deoarece condiția sa este îndeplinită. Dacă clauza nu este satisfăcută, se verifică ceilalți literali din clauză pentru a găsi un „literal watchable" (un literal care poate deveni adevărat). Dacă un astfel de literal este găsit, lista watcherilor este actualizată pentru a include noul literal și procesul continuă. Dacă nu există literali watchable în clauză și aceasta devine unitară (toți ceilalți literali sunt falși), funcția propagă literalul unitar, adăugându-l în coada de propagare. Dacă clauza devine complet falsă (niciun literal nu poate fi satisfăcut), se identifică un conflict. În acest caz, variabila confl este actualizată pentru a semnala conflictul și procesul de propagare se încheie. Actualizarea datelor și ieșirea din funcție.

Pe măsură ce funcția procesează fiecare literal, variabila numprops este incrementată pentru a reflecta numărul total de literali procesați. După ce coada de propagare este complet procesată, funcția returnează fie NULL (dacă propagarea s-a încheiat fără conflicte), fie o referință la clauza în care a fost detectat conflictul.

7.2 analyse()



Funcția analyse() din MiniSat se ocupă cu înțelegerea motivului pentru care apare un conflict și ajută solver-ul să învețe din greșeli, astfel încât să nu facă aceleași erori în viitor. Când solver-ul întâlnește un conflict, funcția este chemată pentru a analiza situația și pentru a crea o clauză nouă, numită clauza învățată, care va ajuta la evitarea acelui conflict data viitoare. Totul începe prin a verifica variabilele implicate în conflict pentru a înțelege ce decizii au dus la problemă.

Pe baza acestor informații, funcția creează clauza învățată, care conține doar variabilele importante legate de conflict. Această clauză este adăugată în baza de date a solver-ului, astfel încât să împiedice reapariția aceleiași situații. În același timp, funcția decide și până la ce pas anterior trebuie să se întoarcă solver-ul pentru a corecta problema, fără să ia de la capăt tot procesul. În loc să reia totul de la zero, solver-ul se întoarce doar până la pasul necesar pentru a continua corect.

La final, funcția analyse() returnează două lucruri: clauza învățată, care este salvată în baza de date, și pasul la care solver-ul trebuie să revină. Această funcție este importantă pentru că ajută solver-ul să rezolve problemele mai repede și să evite să repete greșelile.

8 Cum colaborează DPLL cu CDCL în implementare

În implementarea unui solver SAT modern, precum Minisat, metodele tradiționale de rezolvare a problemei satisfiabilității booleene, cum ar fi DPLL (Davis-Putnam-Logemann-Loveland), sunt combinate cu tehnici avansate de învățare a clauzelor bazate pe CDCL (Conflict-Driven Clause Learning). Cele două metode lucrează împreună pentru a accelera căutarea soluției, îmbunătățind eficiența algoritmului și minimizând numărul de conflicte și propagări necesare pentru a rezolva problema SAT.

8.1 DPLL: Algoritmul de bază

Algoritmul DPLL este un algoritm recursiv care extinde metoda de backtracking pentru a rezolva problemele SAT. DPLL are la bază câteva operații esențiale:

Decizii: Algoritmul face alegeri ale variabilelor, atribuie valori (adevărat sau fals) acestora și se asigură că toate clauzele sunt satisfăcute. Dacă o clauză nu este satisfăcută, se face o decizie de backtrack, încercând o altă valoare pentru variabila respectivă. Propagare unitară: Când o variabilă este atribuită, aceasta poate determina alte variabile să aibă un anumit sens în contextul clauzelor. Propagarea unitară se referă la procesul de propagare a acestor decizii prin toate clauzele implicate. Backtracking: Dacă se ajunge la un punct în care nu există niciun mod de a satisface clauzele, algoritmul face backtracking pentru a modifica deciziile anterioare și a încerca soluții alternative. Totuși, DPLL are limitările sale, în special când vine vorba de gestionarea unui număr mare de clauze și de conflictele care apar. De aici apare necesitatea îmbunătățirii aduse de CDCL.

8.2 CDCL: Învățarea de clauze și gestionarea conflictelor

CDCL extinde DPLL prin integrarea tehnicii de învațare a clauzelor conflictuale și utilizarea unui restart dinamic pentru a îmbunătăți performanța căutării soluției.

Principalele elemente ale CDCL includ:

Învățarea clauzelor: Când apare un conflict, CDCL analizează conflictul și învață o nouă clauză care reflectă acea contradicție. Această clauză este adăugată în baza de date a clauzelor și este folosită pentru a preveni conflictele similare în viitor. În esență, CDCL „învață" din conflictele întâlnite, îmbunătățind căutarea. Backtracking bazat pe conflict: În loc de a face backtracking simplu (ca în DPLL), CDCL efectuează un backtracking bazat pe conflict. Acest lucru înseamnă că atunci când apare un conflict, solverul caută să identifice care decizii anterioare au contribuit la conflict și revine doar la acele decizii, eliminând astfel opțiunile care nu ar fi putut duce niciodată la o soluție satisfăcătoare. Restart dinamic: CDCL introduce și restarturi dinamice, care ajută la evitarea impasurilor în care solverul rămâne blocat. Aceste restarturi ajută la eficientizarea căutării prin resetarea anumitor părți ale algoritmului și încercarea unor noi căi de explorare.

8.3 Integrarea DPLL cu CDCL

DPLL formează baza algoritmului SAT, iar CDCL adaugă optimizări semnificative pentru a îmbunătăți performanța generală. Cele două componente sunt interdependente și colaborează în mod continuu pentru a rezolva problema satisfiabilității. Într-o implementare de tip CDCL, cum este Minisat, DPLL este folosit pentru a face decizii și pentru a propaga valorile atribuite variabilelor, dar CDCL adaugă un strat suplimentar de învățare și gestionare a conflictelor.

8.4 Deciziile și Propagările

Într-un solver CDCL, procesul începe printr-o decizie de variabilă, realizată folosind aceleași tehnici ca în DPLL. O variabilă este aleasă și i se atribuie o valoare. Acesta este punctul în care DPLL „intervine", selectând variabilele care urmează a fi atribuite.

După ce decizia este făcută, propagarea este efectuată pentru a determina dacă această decizie determină un conflict sau duce la noi decizii. Dacă nu există conflicte, algoritmul continuă cu noile decizii, în mod similar cu DPLL.

8.5 Învățarea Clauzelor Conflictuale

Dacă un conflict apare în timpul propagării, CDCL adaugă o componentă suplimentară: învățarea clauzelor conflictuale. Algoritmul analizează conflictul, identifică sursa acestuia și creează o clauză de conflict care reflectă contradicția întâlnită. Acesta este un proces complex de analyze care îmbunătățește căutarea prin adăugarea unei clauze care va preveni aceleași conflicte în viitor.

Odată ce clauza de conflict este învățată, solverul efectuează un backtracking bazat pe conflict. Spre deosebire de DPLL, care revine la niveluri de decizie arbitrare, CDCL revine la un nivel de decizie mai înalt (nivelul care a contribuit la conflict), eliminând ramuri ale căutării care nu vor duce niciodată la o soluție validă.

8.6 Restarturi și Performanță

Într-un solver DPLL, algoritmul ar continua să exploreze aceleași ramuri, indiferent de câte ori întâlnește un blocaj. În schimb, CDCL introduce restarturi dinamice, care resetează anumite părți ale algoritmului pentru a evita blocajele pe termen lung. Aceste restarturi sunt ghidate de o secvență de Luby și sunt utilizate pentru a explora căi alternative într-o manieră mai eficientă. Restarterile sunt aplicate doar atunci când solverul ajunge într-un punct în care nu mai există progres.

8.7 Beneficiile Colaborării DPLL cu CDCL

Colaborarea dintre DPLL și CDCL aduce mai multe beneficii:

Îmbunătățirea performanței: Prin utilizarea învățării clauzelor, CDCL permite solverului să evite conflictele care ar fi apărut din nou, reducând semnificativ timpul de calcul. Explorarea mai eficientă a spațiului de soluții: CDCL folosește restarturi dinamice și backtracking bazat pe conflict pentru a evita blocajele și a eficientiza căutarea. Reducerea dimensiunii bazei de date a clauzelor: În CDCL, clauzele conflictuale care nu sunt folosite de mult timp sunt eliminate, ceea ce reduce memoria utilizată și îmbunătățește performanța.

9 Challenges

Datorita optimizarilor deja existente, orice imbunatatire trebuie gandita foarte bine, deoarece modificarile care functioneaza bine pentru un tip de probleme pot sa nu fie la fel de eficiente pentru alte tipuri. De exemplu, daca introducem o noua strategie pentru selectarea variabilelor, aceasta poate accelera foarte mult rezolvarea formulelor cu putine clauze, dar acelasi algoritm ar putea sa incetineasca semnificativ performanta pentru formule mai complexe sau mai mari. Acest tip de compromis face ca procesul de imbunatatire sa fie unul sensibil si care necesita o analiza detaliata. Performanta unui solver SAT nu depinde doar de modul in care sunt implementati algoritmii, ci si de caracteristicile formulelor pe care le rezolva. De exemplu, o formula cu foarte multe clauze scurte ar putea beneficia de o tehnica diferita fata de o formula cu mai putine clauze, dar mai lungi. De aceea, pentru a evalua cu adevarat efectul unei modificari, este nevoie de un set extins de teste, care sa includa formule din diferite categorii.

Pe partea de benchmark, pot aparea provocari in momentul gestionarii unei probleme complexe cu numar mare de variabile si clauze. Se poate ajunge la un spatiu de cautare mare si o rezolvare mai lenta, astfel necesitand mai multe restarturi si decizii. De asemenea, timpul de procesare poate depasii limita alocata, mai ales in cazul problemelor cu dificultate sporita, rezultatul ramanand "indeterminate", ceea ce sugereaza ca algoritmii de cautare nu sunt suficent de rapizi sau eficienti pentru astfel de situatii.

References

1. Decision Procedures Book, https://www.decisionprocedures.org/handouts/MiniSat.pdf
2. Eén, Niklas and Niklas Sörensson. "An Extensible SAT-solver." International Conference on Theory and Applications of Satisfiability Testing (2003).
3. Heule, Marijn J.H., Iser, Markus, Järvisalo, Matti, and Suda, Martin. *Proceedings of SAT Competition 2024: Solver, Benchmark, and Proof Checker Descriptions.* University of Helsinki, 2024. Available at: http://hdl.handle.net/10138/584822. Non peer-reviewed.

10 Contributii

1. Haiosta Michelle si Bosna Marinel: Analiza codului, Propuneri de imbunatatiri, Challenges, Diagrame
2. Popovici Adrian Robert: Rulare Benchmark
3. Cerean Bogdan-Ioan: Introducere, Descrierea problemei, Instalare MiniSat