

Studiul Algoritmilor de Rezolvare SAT și Implementarea lor în MiniSat

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

West University of Timișoara

Abstract

Lucrarea prezintă o introducere scurtă în probleme de tip SAT și analizează SAT Solverul MiniSat, prezentând modul de instalare a programului, o analiză asupra codului și posibile modalități de a îmbunătăți acest program.

Keywords: MiniSat · SAT Solver · Logică propozițională.

1 Introducere

În această lucrare este prezentat SAT Solverul MiniSat , un program folosit pentru a rezolva probleme de verificare formală cât și în competiții de rezolvare a problemelor SAT. Lucrarea este împărțită în 6 secțiuni care adresează aspectele acestei aplicații:

1. Primul capitol este o prezentare a MiniSat-ului și o introducere a problemelor de tip SAT.
2. Al doilea capitol este o explicație a procesului de instalare și pregătire a MiniSat pentru a-l putea folosi în practică.
3. Al treilea capitol descrie modul de folosire a aplicației pentru a rula un benchmark, un set de probleme folosite în competiții SAT pentru a testa eficiența unui SAT Solver.
4. Al patrulea capitol conferă o analiză asupra codului și logici programului MiniSat, incluzând o diagramă care arată pas cu pas ordinea apelării funcțiilor în program și o descriere a fiecărei metode în parte.
5. Al cincilea capitol discută posibile metode de a îmbunătăți SAT Solver-ul, tema competițiilor anuale de SAT unde mai multe echipe prezintă algoritmi modificați sau programe originale de rezolvare a problemelor de tip SAT.
6. Ultimul capitol, capitolul 6, este o încheiere în care sunt discutate dificultăți întâlnite în timpul compunerii acestei lucrări.

2 Descrierea problemei

Problemele de satisfiabilitate , cunoscute și ca probleme SAT, sunt probleme în care analizăm o propoziție logică, formată din una sau mai multe clauze aflate în conjuncție. Fiecare clauză conține literalii aflați în disjuncție. Un literal reprezintă fie o variabilă, fie o negare de variabilă. Acest format, în care avem doar conjuncții, disjuncții și negări este cunoscut ca Forma Normală Conjunctivă sau CNF. (1)

În urma analizei , care implică acordarea de valori True sau False variabilelor propoziției în diferite combinații, ne așteptăm să găsim o combinație de variabile care satisface fiecare clauză a propoziției. Dacă fiecare clauză este satisfăcută, propoziția finală este adevărată, ceea ce înseamnă că această problemă este satisfiabilă, sau SAT. Dacă nu există o combinație de variabile în care propoziția să fie adevărată, adică dacă nu avem o soluție a problemei, ea este considerată nesatisfiabilă sau UNSAT. (2)

Problemele de tip SAT sunt folosite în domenii cum ar fi inteligența artificială, proiectarea de circuite și Automated Theory Proving (ATP). În aceste domenii pot apărea probleme care au zeci de mii de variabile și sute de mii de simboluri. În rezolvarea manuală apar dificultăți chiar și când o problemă are doar zece variabile, iar complexitatea și timpul necesar pentru a rezolva o problemă SAT

ar fi enorme, chiar și când aplicăm algoritmi cum ar fi CDCL sau DPLL. Din acest motiv, programe de tip SAT Solver, care pot analiza probleme SAT în mod automat și rapid, au fost dezvoltate și încă sunt folosite în prezent. Domeniul problemelor SAT este foarte studiat, existând concursuri anuale în care SAT Solvere sunt puse în competiție pentru a vedea care rulează mai eficient pe seturi de date enorme numite benchmark-uri.

MiniSat este un astfel de SAT Solver, gratuit și open-source, dezvoltat de Niklas Een și Nikita Sorensson în 2003 și publicat în 2004, folosit atât academic, cât și în industrie. Acest program a fost dezvoltat ca fiind un SAT Solver extensibil, care să poată fi modificat și adaptat la diverse domenii mult mai rapid, comparabil cu alte Solvere contemporane sau alternativa de a crea un Solver nou de la zero. MiniSat rulează în Linux, iar instalarea sa este prezentată în capitolul următor.

3 Instalarea MiniSat

Pentru a rula *MiniSat*, este necesar să folosim o distribuție Linux. Putem folosi o mașinărie virtuală pentru a rula MiniSat, dar o metodă mult mai simplă este „Windows Subsystem for Linux”, abreviat WSL, care ne permite să rulăm un environment Linux fără a folosi o mașinărie virtuală sau paralel booting. (5)

Instalarea WSL se poate executa direct din PowerShell sau Windows Command Prompt, folosind comanda „WSL –install” care, după repornirea sistemului, va instala pe calculator o distribuție Ubuntu. Dacă dorim să folosim altă distribuție, este suficient să folosim comanda: `WSL -install <Numele distribuției>`

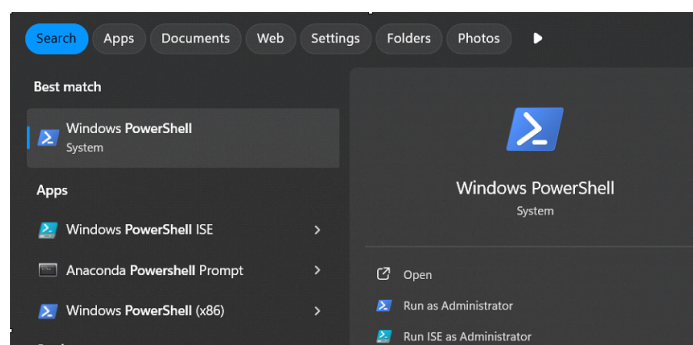


Fig. 1. Fereastra selectare WindowsPowerShell

Odată ce am instalat WSL și am repornit calculatorul, putem folosi aplicația WSL pentru a accesa distribuția Ubuntu pe care o avem. Odată ce accesăm linia de comandă, fie folosind WSL, fie o mașină virtuală, MiniSat poate fi instalat

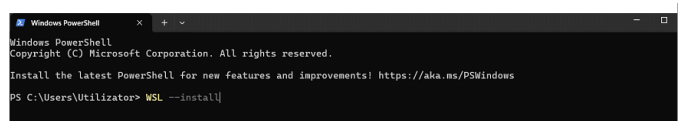
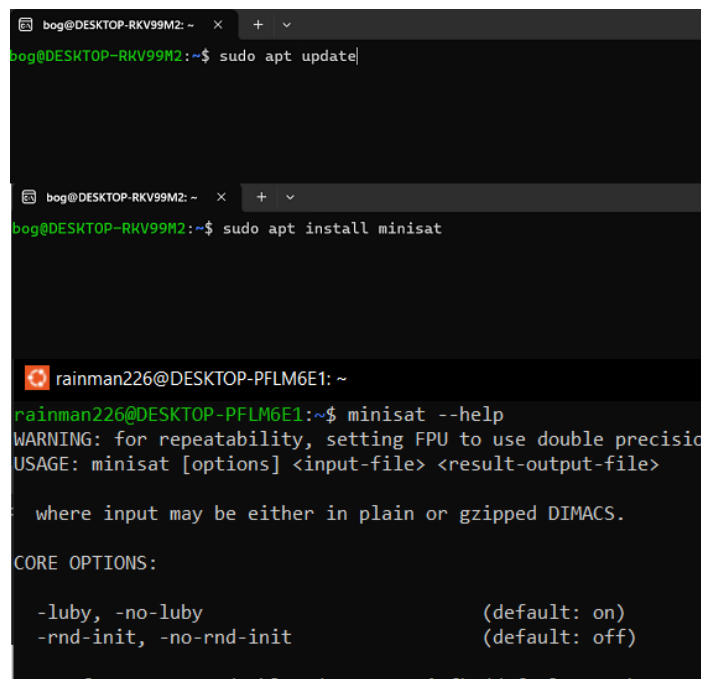


Fig. 2. Instalare WSL

folosind managerul de pachete Linux. Putem actualiza managerul de pachete folosind comanda `sudo apt update`, după care instalarea MiniSat necesită o singură comandă: `sudo apt install minisat`



4 Rularea benchmark

MiniSat este un SAT solver simplu, dar foarte eficient. Pentru a înțelege mai bine cum funcționează și cât de bine face față problemelor complexe, am rulat un benchmark din competiția SAT pentru a analiza performanța și comportamentul său.

Competiția SAT (6) este un concurs internațional în care cei mai buni solvers SAT sunt evaluați pe un set de benchmark-uri standardizate, pentru a testa eficiența și corectitudinea algoritmilor. Aceste benchmark-uri sunt gândite pentru a acoperi o varietate de probleme complexe și pentru a stimula dezvoltarea unor soluții mai rapide și mai eficiente.

4.1 Descriere benchmark folosit

În cadrul acestui test, am folosit un benchmark bazat pe "Simon Cipher" — o schemă criptografică de tip block cipher, care utilizează o structură echilibrată de tip Feistel și operații pe blocuri de biți prin porți logice AND și XOR. Acest tip de schemă este cunoscut pentru simplitatea implementării sale și eficiența sa pe dispozitive de tip Internet of Things, menținând un nivel de securitate acceptabil.

Pentru a adapta problema la formatul SAT și a genera instanțe cu dificultate moderată, s-a simplificat schema originală a algoritmului. Această simplificare a implicat reducerea numărului de runde, eliminarea procesului de programare a cheilor (key scheduling) și utilizarea unui set de stări inițiale prestabilite. Cheia utilizată pentru toate rundele a fost compusă exclusiv din biți de valoare 1 (all-1 key), iar numărul de runde a fost redus la un interval de 16 până la 25. Aceste ajustări asigură că instanțele pot fi rezolvate de către majoritatea solverelor SAT moderne într-un timp rezonabil, păstrând totodată caracteristicile esențiale ale problemei.

Benchmark Names	R	V	C
simon-r16-0.cnf, simon-r16-1.cnf	16	2688	8896
simon-r17-0.cnf, simon-r17-1.cnf	17	2848	9440
simon-r18-0.cnf, simon-r18-1.cnf	18	3008	9984
simon-r19-0.cnf, simon-r19-1.cnf	19	3168	10528
simon-r20-0.cnf, simon-r20-1.cnf	20	3328	11072
simon-r21-0.cnf, simon-r21-1.cnf	21	3488	11616
simon-r22-0.cnf, simon-r22-1.cnf	22	3648	12160
simon-r23-0.cnf, simon-r23-1.cnf	23	3808	12704
simon-r24-0.cnf, simon-r24-1.cnf	24	3968	13248
simon-r25-0.cnf, simon-r25-1.cnf	25	4128	13792

Fig. 3. Detalii instanțe

Benchmarks-urile prezentate sunt afișate în Fig. 3, unde sunt indicate numărul de runde (R), numărul de variabile (V) și numărul de clauze (C). Pentru fiecare rundă, creăm aleatoriu două instanțe folosind două semințe diferite. Aceste instanțe au fost concepute pentru a evidenția performanțele solverelor SAT în fața problemelor criptografice cu o structură bine definită, dar suficient de simplificate pentru a fi analizate eficient.

4.2 Metodologia de rulare extinsă

Pentru a evalua performanța MiniSat pe benchmark-urile selectate, am efectuat experimentele într-un interval total de 24 de ore. Am avut la dispoziție 10 fișiere de benchmark, fiecare fiind rulat cu o limită de timp de 8.460 secunde folosind comanda:

```
minisat -cpu-lim=8640 "$benchmark" "$output".
```

Testele au fost efectuate pe un calculator echipat cu procesor AMD Ryzen 5 4600H și 8 GB RAM, resurse suficiente pentru monitorizarea eficientă a performanței MiniSat.

Pentru a optimiza rulările și a obține rezultate mai eficiente, am utilizat câțiva parametri recomandați dintr-un articol de cercetare (7) care analizează impactul setărilor asupra performanței solverului. Acești parametri ajustează euristici cheie ale MiniSat, având o influență semnificativă asupra timpului de rulare. Iată explicațiile acestora:

- **-var-decay** (factorul de decădere al variabilelor):
Acest parametru controlează frecvența cu care activitatea variabilelor scade după conflicte. Activitatea variabilelor reflectă frecvența cu care acestea apar în conflicte recente. Un factor de decădere mai mic face ca variabilele recente să rămână relevante pentru mai mult timp.
Valoare optimă recomandată: `-var-decay=0.95`.
- **-cla-decay** (factorul de decădere al clauzelor):
Similar cu variabilele, activitatea clauzelor crește atunci când acestea sunt implicate în conflicte și scade periodic în funcție de acest factor de decădere. Un factor de decădere bine ajustat ajută la eliminarea clauzelor inactive, optimizând astfel performanța solverului.
Valoare optimă recomandată: `-cla-decay=0.93`.
- **-rfirst** (intervalul de restart inițial):
Determină numărul de conflicte care trebuie să apară înainte de prima reluare a căutării (restart). Restart-urile frecvente pot ajuta solverul să evite blocarea într-o regiune a spațiului de căutare dificilă de rezolvat.
Valoare optimă recomandată: `-rfirst=200`.

Pe baza acestor recomandări, am configurat rulările benchmark-urilor folosind comanda:

```
minisat -var-decay=0.95 -cla-decay=0.93 -rfirst=200 "$benchmark" "$output".
```

Aceste setări au fost alese deoarece optimizările propuse permit MiniSat să exploateze mai bine structura instanțelor, îmbunătățind atât numărul de conflicte rezolvate, cât și timpul total de rulare.

4.3 Interpretarea valorilor rezultate din rulare

Pentru a evalua performanța MiniSat pe setul de benchmark-uri, am analizat log-urile generate de solver, concentrându-ne pe unele statistici. Datele colectate pot fi împărțite în două părți: statisticile problemei și statisticile rezolvării. Statisticile problemei oferă informații despre configurația inițială a fiecărei instanțe SAT, în timp ce statisticile rezolvării urmăresc progresul solverului în timpul procesului de rezolvare.

Următorii parametri reprezintă caracteristicile de bază ale instanței SAT analizate:

- **Variables:** numărul total de variabile din formula SAT, care indică complexitatea problemei și dimensiunea spațiului de căutare a soluției.
- **Clauses:** Reprezintă numărul total de clauze din formula SAT, ceea ce determină câte relații logice există între variabile. Cu cât sunt mai multe clauze, cu atât problema devine mai complexă și dificilă.

Iar statisticile relevante rezolvării:

- **Restarts:** Restarturile sunt strategii de reînnoire a procesului de căutare pentru a evita blocajele în soluționare și pentru a explora mai eficient spațiul soluțiilor.
- **Conflicts:** Numărul total de conflicte întâmpinate. Conflictele apar atunci când solverul identifică o contradicție în setul curent de alocări, ceea ce forțează revenirea (backtracking) și ajustarea traseului de căutare.
- **Decisions:** Numărul total de decizii făcute. Reprezintă alegerile pe care solverul le face pentru a atribui valori variabilelor. Numărul ridicat de decizii reflectă o căutare extinsă a soluției.
- **Propagations:** Numărul de propagări efectuate. Pași intermediari prin care o decizie influențează alte variabile, reducând spațiul posibil al soluțiilor și avansând soluționarea.
- **Conflict literals:** Numărul total de literalii conflictuali. Sunt literalii care au condus la conflicte și care pot fi eliminați pentru a simplifica problema.
- **Memory used:** Memoria utilizată de solver în timpul execuției.
- **CPU time:** Timpul total de procesare alocat execuției.
- **Result:** Rezultatul final al solverului.

4.4 Rezultate rulare benchmark

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

Tabel cu rezultate

Rezultatele obținute arată că MiniSat a reușit să proceseze benchmark-urile complexe în limitele de timp și memorie specificate, dar pentru toate instanțele testate soluția nu a fost determinată în intervalul de timp alocat, fiind returnat rezultatul INDETERMINATE. Acest lucru evidențiază dificultatea extremă a problemelor abordate, fiecare necesitând un număr foarte mare de conflicte, decizii și propagări pentru a explora spațiul soluțiilor.

Se observă o corelație clară între dimensiunea instanțelor și dificultatea lor. De exemplu, instanțele cu mai multe variabile și clauze, cum ar fi simon-r25

cu 4128 variabile și 13664 clauze, au necesitat un volum mult mai mare de propagări și conflicte în comparație cu instanțe mai mici, precum simon-r16-1 cu 2688 variabile și 8768 clauze. Această creștere a dificultății este confirmată de numărul ridicat de conflicte și decizii pentru instanțele mari, cum sunt simon-r24 (71501545 conflicte și 78398268 decizii) sau simon-r25 (69626160 conflicte și 76191213 decizii). De asemenea, numărul mare de restarturi, cum ar fi 167930 pentru simon-r16-1 și 131071 pentru simon-r17, reflectă provocările întâlnite de solver în evitarea blocării în regiuni dificile ale spațiului de căutare.

În ceea ce privește utilizarea resurselor, memoria folosită a fost relativ scăzută, variind între 90 MB și 134 MB, ceea ce indică o bună eficiență în gestionarea resurselor hardware. Totuși, timpul de procesare s-a apropiat de limita alocată pentru fiecare instanță, de exemplu 8639,73 secunde pentru simon-r25, sugerând că algoritmul folosit de MiniSat nu este suficient de rapid pentru a rezolva astfel de probleme complexe într-un interval de timp rezonabil.

Aceste rezultate subliniază necesitatea unor optimizări suplimentare ale euristiciilor folosite de MiniSat sau chiar a utilizării unor metode mai avansate. Spre exemplu, integrarea unor algoritmi mai sofisticati sau utilizarea unor tehnici bazate pe învățare automată ar putea contribui la reducerea spațiului de căutare și la îmbunătățirea performanței.

5 Analiza Cod

În imaginea de mai jos este prezentată o schiță a algoritmului de search din codul MiniSat (8) , prezentată sub forma de pseudo-cod. Imaginea ne va ajuta să înțelegem pașii principali pe care acest algoritm îi urmează pentru a încerca, verifica și învăța din greșeli până când se găsește o soluție sau până când se demonstrează că nu există nicio soluție.

```

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 înțelege mai în detaliu fluxul acestui algoritm am creat diagrama de secvențe de mai jos, unde avem actorul principal (Solverul) și restul componentelor participante la algoritmul de search: Baza de date a clauzelor, Analizatorul de conflicte, Coadă de literal și Managerul de nivel decizional:

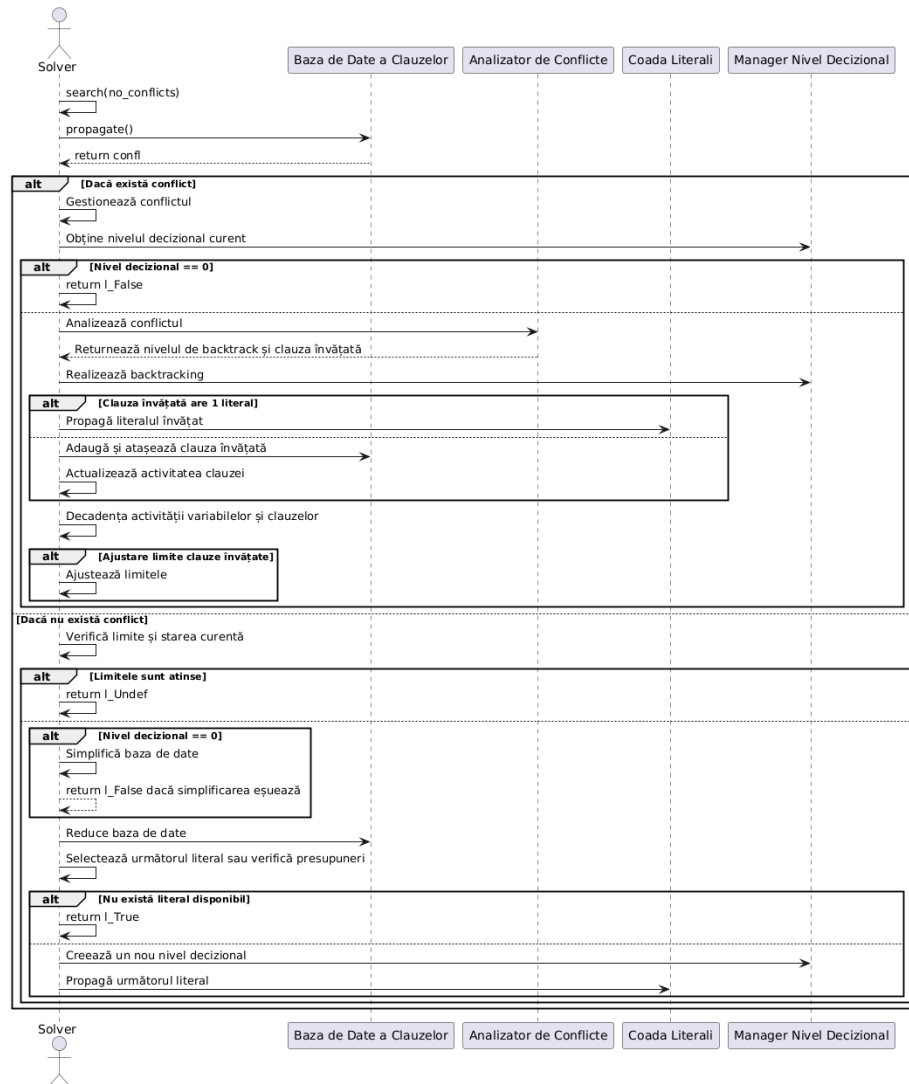


Diagrama de secvențe pentru metoda search

Am observat, uitându-ne la codul proiectului MiniSat, că acesta se folosește de algoritmul CDCL, iar în fișierul solver.cc din directorul core, acest algoritm este bine evidențiat prin metoda search. Pentru a înțelege mai bine cum funcționează **search()**, vom descrie fiecare etapă importantă din acest proces. (9)

Această metodă explorează diferite combinații de valori pentru variabilele din formulă pentru a determina dacă există o soluție care să satisfacă toate clauzele sau, în caz contrar, că formula este nesatisfiabilă. Algoritmul urmează mai mulți pași, iar fiecare componentă joacă un rol esențial în procesul de căutare.

5.1 Inițializare

La început, **search()** inițializează mai multe variabile de lucru, precum **conflicts**, care este contorul de conflicte, și **learnt_clause**, folosită pentru stocarea clauzei conflictuale curente în timpul analizei.

Cazul special este dacă nu există clauze de la început, algoritmul concluzionează că formula este satisfiabilă deoarece o formulă fără clauze este implicit satisfăcută. În acest caz, metoda **search()** returnează o soluție imediat, evitând astfel execuția restului algoritmului.

5.2 Propagare (funcția **propagate()**)

Propagarea este următorul pas în proces, aplicată pentru a deduce valori pentru variabile pe baza clauzelor existente. **propagate()** încearcă să seteze valori noi pentru variabile, verificând în același timp dacă aceste setări satisfac clauzele deja existente. Propagarea unitară este o tehnică esențială folosită în metodă pentru a determina valoarea variabilelor.

Cazul special este dacă **propagate()** detectează un conflict (adică o clauză are toți literalii falși), metoda returnează o referință la acea clauză conflictuală. În acest caz, **search()** trece la analiza conflictului pentru a învăța o clauză nouă care să evite conflictele similare în viitor.

5.3 Gestionarea conflictului

Când **propagate()** returnează o clauză conflictuală, **search()** apelează **analyze()** pentru a determina cauza conflictului. În acest pas, **analyze()** examinează variabilele implicate și construiește o nouă clauză numită clauză de conflict. Aceasta este adăugată la baza de date de clauze și este destinată să prevină conflictele similare în viitor.

Cazul special este dacă conflictul apare la nivelul de decizie 0, adică la nivelul de bază al algoritmului, **search()** concluzionează că formula este nesatisfiabilă. În această situație, funcția se oprește și returnează UNSAT, deoarece nu există nicio configurație a variabilelor care să satisfacă formula.

5.4 Învățarea clauzei de conflict și backtracking-ul (funcția **analyze()**)

Funcția **analyze()** parcurge clauza conflictuală (**conflict_clause**) și variabilele aferente pentru a construi o clauză nouă care să împiedice conflicte similare. În acest mod, solverul CDCL devine mai eficient pe măsură ce își „amintește” conflictele trecute.

Cazul special este după crearea clauzei de conflict, **search()** stabilește la ce nivel de decizie să revină (backtracking) pentru a evita conflictul. Această revenire la un nivel anterior de decizie permite algoritmului să încerce un alt set de decizii, explorând astfel o altă parte a spațiului soluțiilor.

5.5 Curățarea bazei de date de clauze (funcția `reduceDB()`)

Pentru a evita acumularea de clauze care încetinesc algoritmul, `search()` apelează `reduceDB()` la intervale regulate. Funcția `reduceDB()` șterge clauzele cu activitate scăzută, adică acelea care nu contribuie semnificativ la prevenirea conflictelor.

Cazul special: Dacă baza de date de clauze devine prea mare, `reduceDB()` elimină clauzele mai puțin relevante, asigurând o performanță optimă pe termen lung. Acest proces ajută algoritmul să fie mai eficient în gestionarea memoriei și a timpului de execuție.

5.6 Decizie pentru variabilă (funcția `pickBranchLit()`)

Dacă nu sunt găsite conflicte în propagare și nu au fost setate toate variabilele, `search()` selectează o variabilă de decizie folosind `pickBranchLit()`. Această funcție se bazează pe activitatea variabilelor (cele implicate frecvent în conflicte recente), măsurată în funcție de frecvența cu care variabilele au fost implicate în conflicte recente. Variabilele cu activitate ridicată sunt prioritizate pentru a spori eficiența căutării. Setarea acestei variabile contribuie la avansarea în căutare și la explorarea spațiului soluțiilor.

Cazul special Dacă toate variabilele au fost setate fără a apărea conflicte, `search()` returnează SAT, indicând că formula este satisfiabilă.

5.7 Restarturi

Restarturile se declanșează atunci când numărul de conflicte întâlnite de solver depășește un prag predefinit. Acest prag poate fi stabilit, sau fix, sau ajustat dinamic, în funcție de progresul solverului. Acesta constă în revenirea la nivelul de decizie inițial (0) și începerea unei noi explorări a spațiului soluțiilor. Toate clauzele de conflict învățate sunt păstrate, astfel încât algoritmul să folosească aceste informații în căutarea ulterioară. Restarturile sunt o strategie pentru a evita blocarea într-o regiune specifică a căutării și pentru a explora noi rute de soluții.

Cazul special dacă numărul de conflicte depășește un prag stabilit, algoritmul execută un restart și revine la nivelul de decizie 0, reluând căutarea, dar păstrând clauzele conflictuale învățate.

5.8 Finalizarea căutării

Dacă `search()` găsește o setare a variabilelor care satisface toate clauzele, algoritmul returnează `l_True`, indicând că formula este SAT. Pe de altă parte, dacă la nivelul de decizie 0 apare un conflict și nu mai există alte căi de explorat, algoritmul conchide că formula este UNSAT și returnează `l_False`.

6 Doxygen

6.1 Ce este Doxygen?

Doxygen (10) este un generator de documentație care funcționează pe mai multe limbaje de programare printre care și `c++` sau `c`, limbaje cu care a fost scrisa implementarea MiniSat. Acest generator de documentații extrage comentariile formate din codul sursă pe care dezvoltatorul software le adauga și salvează toate aceste informații într-un format acceptat, în cazul nostru formatul în care Doxygen salvează informațiile este `.html` și `latex`, de asemenea se consideră faptul că Doxygen acceptă analiza statică a unei baze de cod.

6.2 De ce am ales să documentăm codul cu Doxygen?

Doxygen este unul dintre cele mai populare instrumente de documentare a codului folosit în dezvoltarea software, însă motivația principală în alegerea folosirii Doxygen a fost dată de faptul că MiniSat nu are nici un fel de documentație a codului în nici un fel de format, de asemenea în formatul generat de Doxygen cu documentația codului se poate include și fișierul `read.me` ceea ce oferă niște informații clare despre proiect, benchmarkuri și niște pași clari în instalarea MiniSat pe computerul personal.

6.3 Cum am instalat Doxygen?

Pentru instalarea Doxygen pe Windows trebuie să accesăm site-ul lor oficial: <https://www.doxygen.nl/download.html> și să descărcăm executorul. După rularea executorului trebuie să urmărim pașii pentru instalarea lui pe computerul personal. După finalizarea pașilor de instalare trebuie să adăugăm în `environment variables` path-ul către directoriul Doxygen. Însă pentru instalarea Doxygen pe Linux putem să utilizăm managerul de pachete folosind comanda: `sudo apt install doxygen`, iar pe macOS putem să instalăm Doxygen folosind Homebrew ruland: `brew install doxygen`. După instalarea Doxygen trebuie să parcurgem niște pași de configurare:

1. Navigare în directorul proiectului nostru.
2. Generarea fișierului de configurare `Doxyfile` folosind: `doxygen -g`. `Doxyfile` este fișierul de configurare care conține toate setările proiectului nostru.
3. Generarea setărilor documentației. După ce ne-am configurat fișierul `Doxyfile` pentru generarea documentației cum ar fi definirea fișierelor pe care dorim să le includem în documentația noastră, tipul documentației pe care dorim să îl generăm (`latex`, `html`, `xml`, etc.), sau putem configura setări personalizabile pentru cum vrem să arate documentația noastră (putem genera documentația doar pentru metode sau attribute cu un anumit nivel de vizibilitate).

4. Generarea fișierelor de documentație folosind comanda: `doxygen Doxyfile`. După rularea comenzii se vor genera fișierele `html` și `latex`. Pentru vizualizarea documentației în format `html` trebuie să intrăm în directorul `html` și să accesăm fișierul `index.html`, dacă am optat pentru configurarea documentației în `latex` în fișierul de configurare `Doxyfile` atunci se va genera și un fișier `latex` pe care îl putem accesa pentru vizualizarea documentației codului în `latex`. Fișierul `latex` se poate converti în format `zip` și încarca în orice fel de editor (ex: Overleaf).
5. După fiecare actualizare a codului pentru actualizarea documentației trebuie să rulăm din nou comanda: `doxygen Doxyfile` în fișierul sursă al codului.

6.4 Cum am comentat codul folosind Doxygen

Comenzi utile pentru utilizarea Doxygen:

- **doxygen -g**: aceasta comandă ne crează fișierul de configurație `Doxyfile` în directorul unde rulăm comanda. Fișier ce conține toate setările de generare a documentației.
- **doxygen Doxyfile**: această comandă rulează fișierul de configurație conform setărilor din interiorul său și generează documentația în formatul dorit.
- **GENERATE_LATEX = YES**: această comandă convertește fișierele `latex` în fișiere `pdf` care pot fi vizualizate fără un editor precum Overleaf.
- **HAVE_DOT = YES**: această comandă ne permite generarea automată a diagramelor de clasă folosind Graphviz.

Tag-uri utile pentru gestionarea fișierelor de cod:

- **@file**: acest tag se ocupă de descrierea unui fișier sursă. Exemplu:

```
/**
 * @file Solver.cc
 * @brief Implementarea MiniSat SAT solver.
 *
 * Acest fișier conține cele mai importante metode pentru rezolvarea problemelor SAT
 */
```

- **@class**: acest tag descrie rolul și funcționalitatea unei clase. Exemplu:

```
/**
 * @class Solver
 * @brief Clasa principală pentru rezolvarea problemelor SAT.
 *
 */
```

- **@brief**, **@param**, **@return**: **@brief** se ocupă cu descrierea unei funcții sau clase, **@param** se ocupă cu descrierea parametrilor unei funcții, **@return** se ocupă cu precizarea tipului de return a unei funcții.

```

/**
 * @brief Realizează propagarea tuturor literali puși în coadă.
 *
 * @param level Nivelul decizional curent.
 * @return CRef Referință la clauza conflictuală dacă există un conflict; altfel, CR
 */
CRef propagate(int level);

```

- **@see**, **@ref**: **@see** sau **@ref** creează legături între funcții, clase sau orice alt tip de fișier din cod. Exemplu:

```

/**
 * @brief Algoritmul principal de căutare.
 *
 * @see propagate()
 * @see analyze()
 */
lbool search(int nof_conflicts);

```

- **@note**, **@warning**: acest tag se ocupă cu evidențierea lucrurilor mai importante sau cu atenționarea la diferite riscuri. Exemplu:

```

/**
 * @note Această funcție este optimizată pentru probleme mici.
 * @warning Utilizarea pentru probleme mari poate duce la performanțe scăzute.
 */
void optimizeDatabase();

```

7 Procesul de Compilare, Rulare și Interpretare a Rezultatelor MiniSat

Am configurat și rulat în detaliu proiectul MiniSat în CLion(11) . Am documentat toate etapele pentru a putea explica clar ce am făcut, cum am procedat și de ce am luat anumite decizii. În această explicație, mă voi concentra pe fiecare detaliu pentru a crea un ghid care să reflecte procesul meu pas cu pas.

7.1 Crearea fișierului de input pentru MiniSat

Primul pas a fost să pregătesc un fișier de input care să definească o problemă logică pe care solverul MiniSat să o poată procesa. Am creat un fișier text simplu pe care l-am numit **input.cnf**. Acesta conține problema exprimată în formatul DIMACS standard, folosit pentru problemele de satisfacibilitate booleane. Formatul este bine cunoscut și ușor de interpretat de către MiniSat.

În fișierul meu **input.cnf**, am inclus următorul conținut:

```
p cnf 2 1
1 -2 0
```

Prima linie, `p cnf 2 1`, indică faptul că problema este de tip CNF și specifică că avem 2 variabile și 1 clauză.

A doua linie, `1 -2 0`, reprezintă o singură clauză. Aceasta spune că variabila 1 este adevărată, iar variabila 2 este falsă. 0 marchează sfârșitul fiecărei clauze, conform specificației DIMACS.

Am plasat acest fișier în directorul **cmake-build-debug**. Acest director este generat automat de CLion atunci când proiectul este compilat. Dacă fișierul nu este în acest director, rularea nu va funcționa.

7.2 Configurarea proiectului MiniSat în CLion

Pentru a putea rula corect MiniSat cu fișierul de input, am făcut câteva configurări în CLion. Acest pas este esențial, deoarece fără el, programul nu ar ști unde să găsească fișierul `input.cnf`.

Primul pas a fost să accesez secțiunea de configurare a rulării în CLion. Pentru aceasta, am mers în meniul principal al IDE-ului și am selectat opțiunea **Edit Configurations**. Aici, se află toate configurațiile proiectului pentru rularea aplicației. Am căutat și am selectat configurația specifică numită `minisatsimp`, care este responsabilă pentru rularea principală a MiniSat.

În continuare, am setat argumentele programului. Acestea sunt necesare pentru a specifica fișierul de input pe care MiniSat trebuie să-l proceseze. În câmpul `Program Arguments`, am introdus numele fișierului meu de input, `input.cnf`. Aceasta este denumirea fișierului care conține problema SAT, exprimată în format DIMACS. Este esențial ca acest câmp să fie completat corect, deoarece fără acest argument, programul MiniSat nu va ști ce fișier să proceseze și va genera erori.

După setarea argumentelor, am configurat și directorul de lucru. Programul MiniSat trebuie să știe unde să caute fișierul `input.cnf`. În câmpul `Working Directory`, am introdus calea către directorul `cmake-build-debug`. Directorul `cmake-build-debug` este generat automat de CLion în timpul procesului de compilare, iar fișierul `input.cnf` trebuie plasat în acest director pentru a fi accesibil programului în timpul rulării.

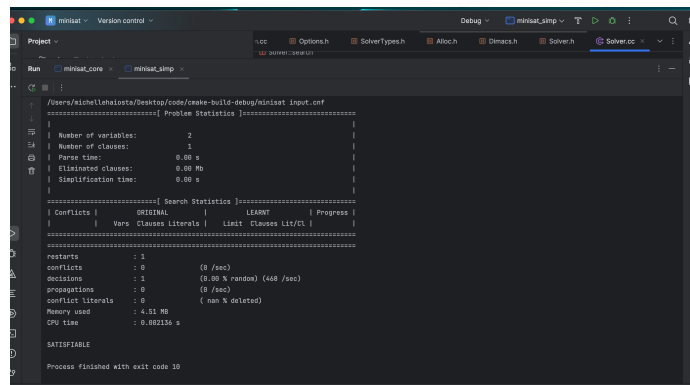
7.3 Compilarea și rularea proiectului MiniSat

După ce am configurat proiectul în CLion, am trecut prin procesul de compilare.

Procesul de compilare a fost inițiat prin accesarea opțiunii `Build` din CLion. Am apăsat pe butonul `Build` pentru a porni procesul de compilare. Această acțiune a permis IDE-ului să analizeze codul sursă al MiniSat, să rezolve eventualele dependențe și să genereze fișierele binare necesare rulării programului. Compilarea s-a desfășurat fără erori, confirmând faptul că toate configurațiile anterioare au fost realizate corect și că codul sursă al MiniSat este valid.

După compilare, am trecut la etapa de rulare a proiectului. Pentru a executa programul, am apăsât pe butonul Run din CLion. Aceasta a declanșat procesul de rulare a MiniSat folosind fișierul input.cnf specificat în configurarea proiectului. Programul a citit conținutul fișierului input.cnf din directorul de lucru setat în configurare (cmake-build-debug) și a procesat problema de satisfiabilitate definită acolo. Rularea s-a desfășurat cu succes, iar rezultatul a fost afișat în consola CLion.

Rezultatul execuției a fost afișat sub forma unui raport detaliat generat de MiniSat. Acesta include informații despre statistici ale problemei și despre performanța solverului în timpul rulării. Mai jos este o interpretare detaliată a datelor afișate în consolă:



```

/Users/nichellehoasta/Desktop/code/cmake-build-debug/minisat input.cnf
===== Problem Statistics =====
|
| Number of variables: 2 |
| Number of clauses: 1 |
| Parse time: 0.00 s |
| Eliminated clauses: 0.00 Mb |
| Simplification time: 0.00 s |
|
===== Search Statistics =====
| Conflicts | ORIGINAL | LEARNED | Progress |
| | New Clauses Literals | Limit | Clauses Lb/Lt | |
=====
restarts : 1
conflicts : 0 (0 /sec)
decisions : 1 (0.00 % random) (408 /sec)
propagations : 0 (0 /sec)
conflict literals : 0 (non % related)
Memory used : 4.51 MB
CPU time : 0.002136 s

SATISFIABLE
Process finished with exit code 10

```

Fig. 4. Rezultatul execuției generat de MiniSat

MiniSat oferă două categorii principale de informații: statistici despre problemă și detalii privind execuția programului.

Statistici despre problemă: Solverul a identificat două variabile și o singură clauză logică în problema specificată. Fișierul input.cnf a fost citit și procesat aproape instantaneu (Parse time: 0.00 s), fără a elimina clauze redundante sau a necesita timp suplimentar pentru simplificare.

Detalii despre execuție: Solverul a efectuat un singur restart și nu a întâmpinat conflicte, ceea ce indică faptul că problema a fost simplă și bine definită. Cu o singură decizie luată și fără propagări suplimentare, algoritmul a găsit rapid soluția. Consumul de memorie a fost minim (4.51 MB), iar timpul de procesare a fost extrem de scurt (0.001 s).

Rezultatul final: Solverul a determinat că problema este SATISFIABILĂ, ceea ce înseamnă că o configurație validă a variabilelor satisface condițiile specificate în fișierul de input.

7.4 Analiza etapei de debug – Funcția Solver::search

În această etapă, am analizat comportamentul funcției Solver::search din cadrul proiectului MiniSat utilizând breakpoint pentru a înțelege fluxul de execuție și starea variabilelor. Această funcție este responsabilă de găsirea unei soluții satisfiabile sau de confirmarea nesatisfiabilității problemei.

Funcția Solver::search se ocupă de procesarea clauzelor, luarea deciziilor și propagarea acestora în cadrul algoritmului SAT. La momentul analizei, am configurat inputul ca fișierul input.cnf, plasând fișierul în directorul cmake-build-debug, și am setat acest director ca Working Directory în configurația de rulare. Fișierul CNF utilizat conținea o problemă simplă pentru a facilita înțelegerea procesului.

Breakpoint la linia 788

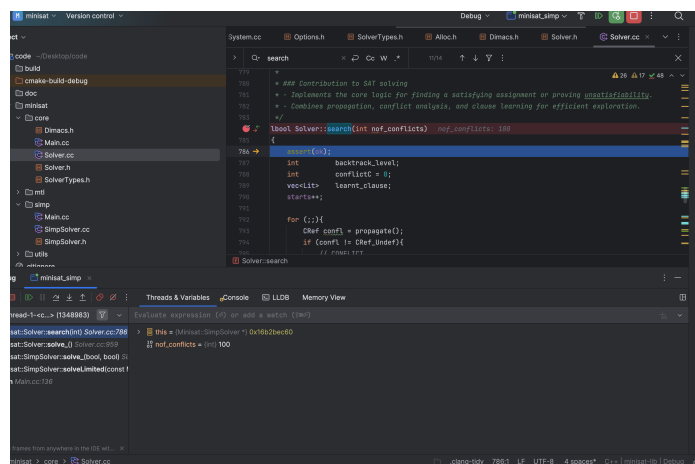


Fig. 5. debug Search

La începutul funcției Solver::search, solverul inițializează variabilele necesare procesării problemei.

nofconflicts este setată la 100. Aceasta definește limita maximă de conflicte acceptate înainte de a efectua un restart. Este o măsură de control care ajută solverul să gestioneze eficient explorarea.

backtracklevel are o valoare mare (16082616 în cazul nostru). Acest lucru indică faptul că, la momentul inițial, solverul nu se află într-o stare de conflict și nu are nevoie să revină la niveluri anterioare pentru a reevalua deciziile. Concluzie: Această etapă confirmă că solverul este pregătit să proceseze problema fără conflicte inițiale, iar toate variabilele relevante sunt corect setate pentru procesarea ulterioară.

Breakpoint la linia 837

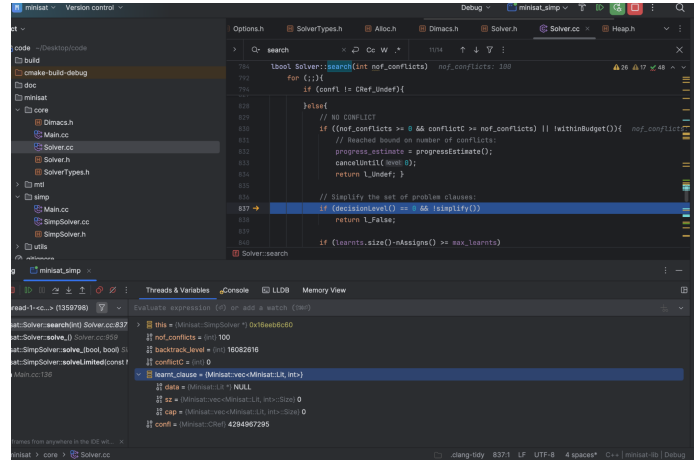


Fig. 6. debug Solver

La acest punct, solverul verifică dacă este necesară simplificarea clauzelor. Procesul de simplificare reduce redundanțele și optimizările clauzelor pentru a accelera rezolvarea:

Condiția evaluată este `decisionLevel() == 0 and !simplify()`. Aceasta verifică dacă solverul se află la nivelul de decizie 0 (fără decizii recente) și dacă există clauze care pot fi simplificate.

În cazul nostru:

Simplificarea nu este necesară, deoarece problema este simplă, iar clauzele sunt deja optimizate.

Solverul continuă procesarea fără a efectua modificări suplimentare. Variabile observate la acest punct:

`learnclause`: Aceasta este goală (NULL), ceea ce indică faptul că solverul nu a întâlnit conflicte și, astfel, nu a fost nevoie să învețe sau să deducă clauze noi.

`conflict`: Valoarea rămâne 0, confirmând că solverul nu a identificat contradicții în procesul de propagare sau decizie.

8 Propuneri de îmbunătățiri pentru metodele MiniSat

MiniSat este un solver puternic pentru problema SAT, dar, ca orice algoritm, poate fi optimizat. În continuare, sunt prezentate câteva direcții posibile de îmbunătățire. (12) (13)

8.1 Învățarea clauzelor conflictuale

În metoda **analyze()**, MiniSat învață clauze noi din conflictele pe care le întâlnește, astfel încât să evite conflictele similare în viitor. Acest proces funcționează foarte

bine, dar, pe măsură ce solverul rulează, baza de date de clauze devine foarte mare, ceea ce poate încetini algoritmul.

O posibilă îmbunătățire ar fi introducerea unei analize mai detaliate a clauzelor învățate. De exemplu, dacă o clauză nu a fost utilizată într-o propagare recentă, aceasta este probabil inutilă și poate fi eliminată. În plus, putem optimiza clauzele învățate reducând numărul de literali din ele. Clauzele mai compacte sunt mai rapide de utilizat în propagare, dar trebuie să fim atenți să nu pierdem informații importante.

8.2 Curățarea bazei de date de clauze

Metoda **reduceDB()** elimină periodic clauzele învățate care nu mai sunt utile, dar procesul actual se bazează pe reguli fixe. De exemplu, clauzele cu activitate scăzută sunt șterse automat, indiferent dacă mai sunt relevante sau nu.

Propunem o metodă mai dinamică de curățare a bazei de date. În loc să ștergem toate clauzele sub un anumit prag de activitate, putem evalua cât de des au fost utilizate recent în propagare. Clauzele care nu au fost folosite deloc în ultimele iterații ar putea fi șterse mai întâi. De asemenea, putem curăța doar o parte din baza de date la fiecare iterație, pentru a evita încetinirea solverului.

8.3 Optimizarea propagării

Propagarea unitară este unul dintre cele mai importante procese din algoritmul MiniSat, dar și unul dintre cele mai costisitoare din punct de vedere al timpului de execuție. În timpul propagării, solverul verifică clauzele pentru a vedea dacă setările actuale ale variabilelor determină anumiți literali să fie adevărați sau falși. Pentru a face acest lucru eficient, MiniSat folosește liste „watch” pentru literali, un mecanism care urmărește doar doi literali din fiecare clauză. Totuși, acest proces poate fi optimizat prin îmbunătățirea structurii listei „watch”.

În forma actuală, listele „watch” sunt implementate astfel încât fiecare literal să fie asociat cu un set de clauze în care este urmărit. Acest lucru funcționează bine pentru probleme mai mici, dar pentru formule mari, cu milioane de clauze și variabile, accesarea și actualizarea acestor liste poate deveni lentă. De exemplu, de fiecare dată când un literal devine fals, solverul trebuie să parcurgă clauzele asociate acestuia pentru a găsi un alt literal valid care să fie urmărit.

Să considerăm următoarea formulă SAT simplă în forma normală conjunctivă (CNF):

$$C_1 : (x_1 \vee \neg x_2 \vee x_3)$$

$$C_2 : (\neg x_1 \vee x_4)$$

$$C_3 : (x_2 \vee \neg x_4 \vee x_5)$$

În această formulă, fiecare clauză conține literali care pot fi fie o variabilă (x_1, x_2, \dots) fie negarea unei variabile ($\neg x_1, \neg x_2, \dots$). La începutul algoritmului, MiniSat selectează doi literali din fiecare clauză pentru a-i urmări. De exemplu, în C_1 , ar putea fi selectați x_1 și $\neg x_2$ ca literali „watch”. Acest lucru permite

solverului să monitorizeze doar o parte din clauză, reducând astfel numărul de verificări necesare pentru propagare.

În timpul execuției, dacă solverul decide că $x_1 = \text{False}$, clauzele care urmăresc x_1 trebuie reevaluate. Pentru C_1 , x_1 fiind fals, solverul trebuie să găsească un alt literal valid care să fie urmărit, cum ar fi x_3 . Acest lucru asigură că C_1 rămâne satisfăcută. În mod similar, pentru C_2 , dacă $x_1 = \text{False}$, x_4 devine singurul literal „watch” valid. Dacă x_4 este ulterior setat la **False**, clauza C_2 devine conflictuală, iar solverul trebuie să gestioneze acest conflict.

Deși aceste liste „watch” sunt eficiente în reducerea numărului de verificări, accesarea și actualizarea lor poate fi un proces costisitor în cazul problemelor de mari dimensiuni. Fiecare literal are asociată o listă de clauze, iar atunci când un literal devine fals, solverul trebuie să parcurgă toate clauzele asociate pentru a găsi un alt literal valid. Acest proces poate consuma mult timp, mai ales dacă numărul de clauze este mare.

Pentru a aborda această problemă, o posibilă optimizare este utilizarea unei structuri de date mai eficiente, cum ar fi un **hash map**. În loc ca listele „watch” să fie stocate ca vectori, fiecare literal poate fi o cheie într-un hash map, iar valorile asociate să fie listele de clauze. Această structură permite accesarea și actualizarea mai rapidă a clauzelor asociate unui literal. De exemplu, pentru formula de mai sus, un hash map ar putea fi organizat astfel:

Literal $x_1 : \{C_1, C_2\}$

Literal $x_2 : \{C_1, C_3\}$

Literal $x_3 : \{C_1\}$

Literal $x_4 : \{C_2, C_3\}$

Literal $x_5 : \{C_3\}$

Această organizare reduce timpul necesar pentru accesarea clauzelor și face propagarea mai eficientă. Mai mult, pentru formule mari, unde fiecare literal poate fi asociat cu mii de clauze, utilizarea unui hash map asigură o scalabilitate mai bună și minimizează impactul creșterii dimensiunii formulei asupra performanței solverului.

9 Prezentarea metodelor cheie din MiniSat prin diagrame UML

9.1 propagate()

[H]

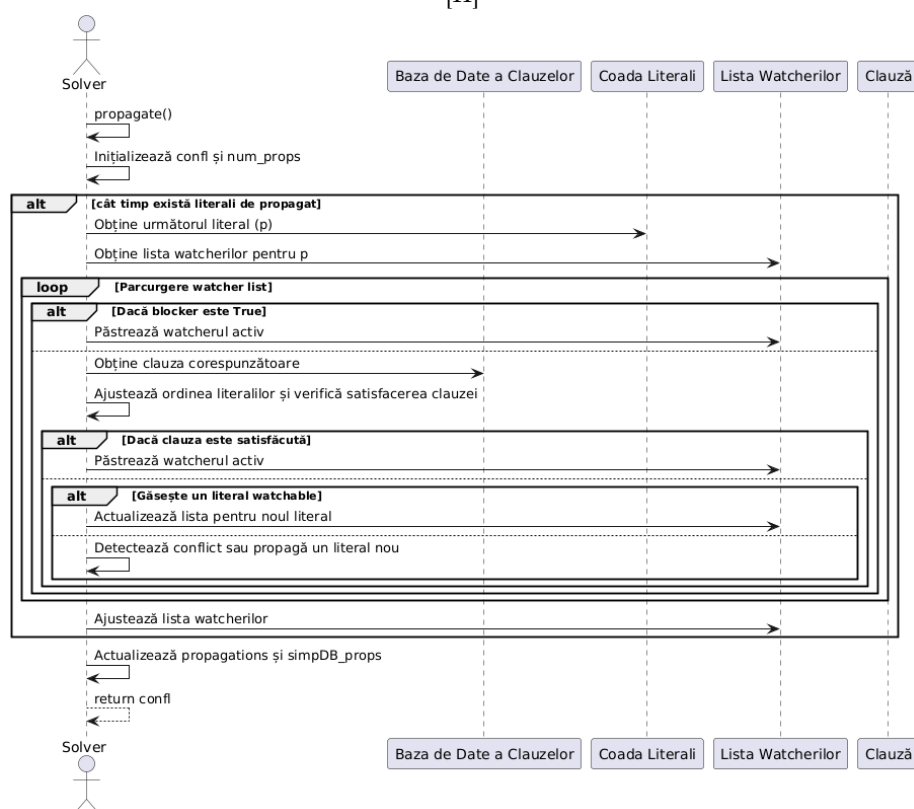


Diagrama de secvențe pentru metoda propagate

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 literalii, se examinează clauzele asociate pentru a decide dacă există deducții noi ce pot fi făcute sau dacă a apărut un conflict.

Descriere detaliată a funcției propagate(): La început, funcția verifică dacă coada de propagare conține literalii neprocesați. Dacă coada este goală, funcția se încheie, indicând că nu mai există deducții de făcut. În caz contrar, se selectează un literal din coadă și se examinează clauzele asociate acestuia.

Pentru fiecare clauză, funcția verifică dacă starea variabilelor permite deducerea unui nou literal. Dacă o clauză are toți literalii evaluați ca fiind falși,

se generează un conflict, iar funcția returnează referința la această clauză. În schimb, dacă un literal al clauzei rămâne nesatisfăcut, acesta este dedus ca fiind adevărat, iar propagarea continuă.

Acest proces se repetă până când fie coada de propagare devine goală, fie un conflict este detectat. Funcția **propagate()** joacă un rol crucial în eficiența algoritmului, deoarece permite reducerea spațiului de căutare prin deducții automate și identificarea rapidă a contradicțiilor.

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 literal procesați pe parcursul execuției funcției.

Coadă 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 literal 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 literal watchable în clauză și aceasta devine unitară (toți ceilalți literal 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 semnaliza 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 literal 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.

9.2 analyse()

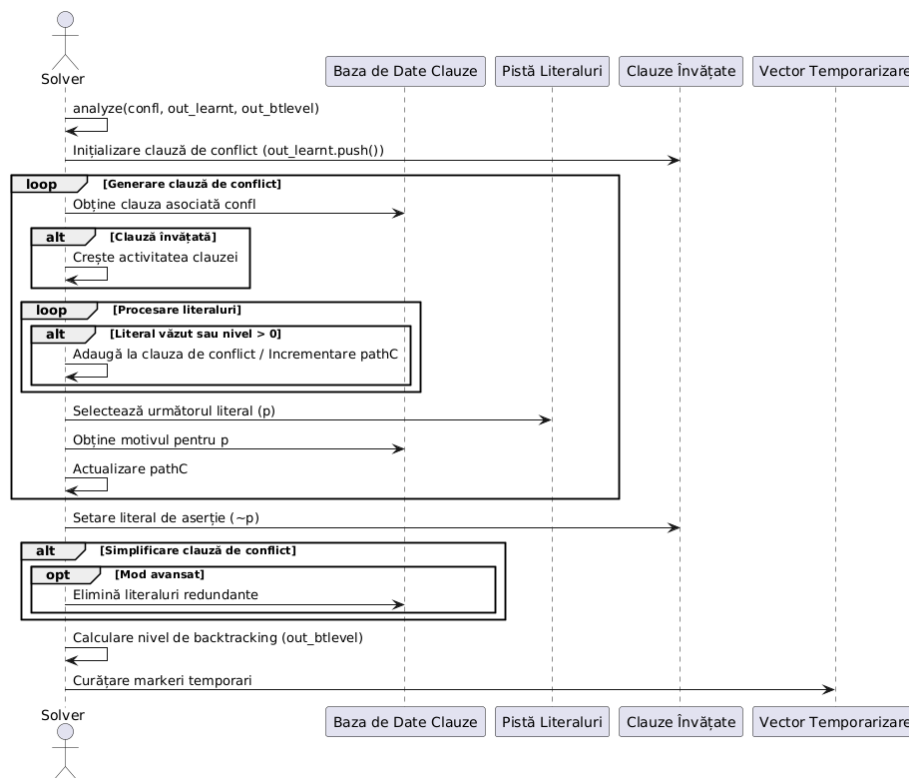


Diagrama de secvențe pentru metoda analyse

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.

9.3 reduceDb()

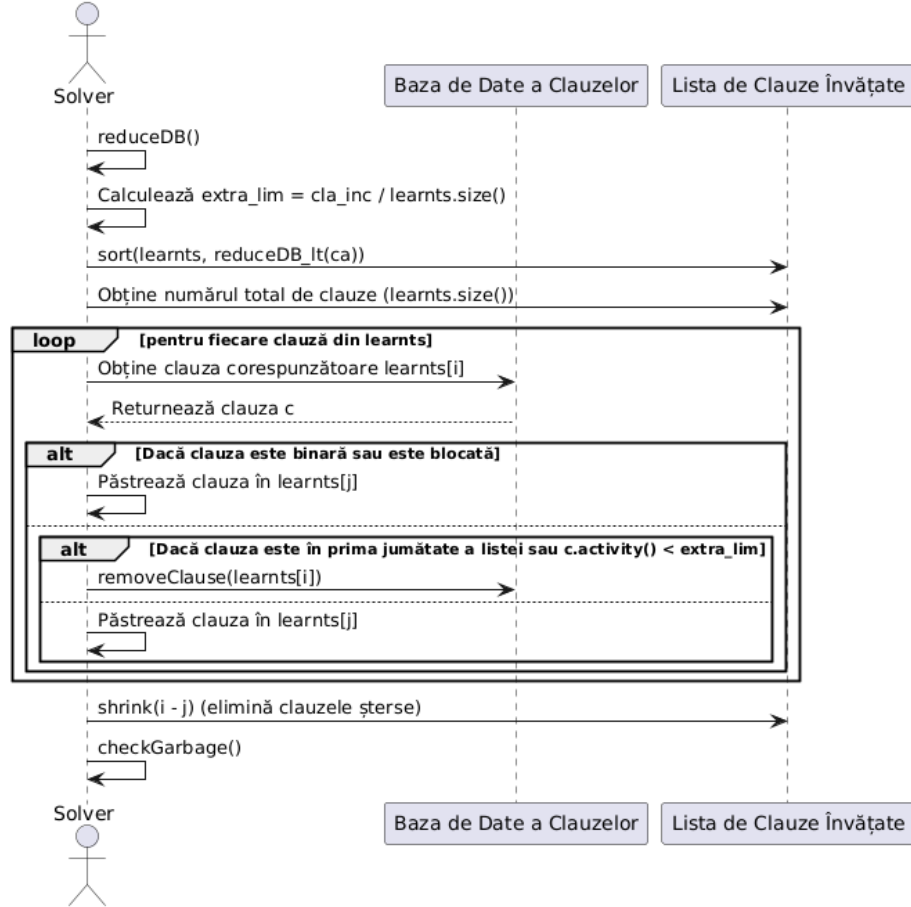


Diagrama de secvențe pentru metoda reduceDb

Funcția `reduceDB()` are rolul de a menține solverul eficient și rapid prin eliminarea clauzelor învățate care nu mai sunt utile. La început, calculează un prag de activitate (`extralim`) care ajută la identificarea clauzelor relevante. Apoi, ordonează clauzele învățate astfel încât cele mai puțin importante să fie analizate primele. Clauzele care sunt binare sau blocate sunt păstrate deoarece ajută foarte mult în procesul de deducere. În schimb, clauzele mai puțin active sau cele aflate în prima jumătate a listei sunt eliminate pentru a nu aglomera solverul. La final, funcția actualizează lista de clauze și curăță memoria, astfel încât MiniSat să funcționeze mai eficient și să evite supraîncărcarea cu informații inutile.

10 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 (14)

10.1 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. Propagare 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.

10.2 CDCL: Învățarea de clauze și gestionarea conflictelor

CDCL extinde DPLL prin integrarea tehnicii de învățare 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 „învăță” 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. (15)

10.3 Integrarea DPLL cu CDCL

DPLL formează baza algoritmului SAT, iar CDCL adaugă optimizări semnificative pentru a îmbunătăți performanța generală. Cele două componente sunt in-

terdependente 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.

10.4 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.

10.5 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 analize 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.

10.6 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. Restarturile sunt aplicate doar atunci când solverul ajunge într-un punct în care nu mai există progres.

10.7 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.

11 Challenges

Datorită optimizărilor deja existente, orice îmbunătățire trebuie gândită foarte bine, deoarece modificările care funcționează bine pentru un tip de probleme pot să nu fie la fel de eficiente pentru alte tipuri. De exemplu, dacă introducem o nouă strategie pentru selectarea variabilelor, aceasta poate accelera foarte mult rezolvarea formulelor cu puține clauze, dar același algoritm ar putea să încetinească semnificativ performanța pentru formule mai complexe sau mai mari. Acest tip de compromis face ca procesul de îmbunătățire să fie unul sensibil și care necesită o analiză detaliată. Performanța unui solver SAT nu depinde doar de modul în care sunt implementați algoritmii, ci și de caracteristicile formulelor pe care le rezolvă. De exemplu, o formulă cu foarte multe clauze scurte ar putea beneficia de o tehnică diferită față de o formulă cu mai puține clauze, dar mai lungi. De aceea, pentru a evalua cu adevărat efectul unei modificări, este nevoie de un set extins de teste, care să includă formule din diferite categorii.

Pe partea de benchmark, pot apărea provocări în momentul gestionării unei probleme complexe cu număr mare de variabile și clauze. Se poate ajunge la un spațiu de căutare mare și o rezolvare mai lentă, necesitând mai multe restarturi și decizii. De asemenea, timpul de procesare poate depăși limita alocată, mai ales în cazul problemelor cu dificultate sporită, rezultatul rămânând "indeterminate", ceea ce sugerează că algoritmii de căutare nu sunt suficient de rapizi sau eficienți pentru astfel de situații.

Bibliography

- [1] A. Biere, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability*. Amsterdam, Netherlands: IOS Press, 2009.
- [2] L. Zhang and S. Malik, *Boolean Satisfiability: Theory and Engineering*. New York, NY: Springer, 2013.
- [3] N. Eén and N. Sörensson, “Minisat: A sat solver with conflict-clause minimization,” in *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, (St. Andrews, UK), pp. 502–518, 2005.
- [4] N. Eén and N. Sörensson, “An extensible sat-solver,” in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, (Santa Margherita Ligure, Italy), pp. 502–518, 2003.
- [5] N. Eén and N. Sörensson, *MiniSat User Guide*, 2005.
- [6] “Sat competition,” 2002–2023. Annual competition proceedings and solver descriptions.
- [7] F. Beskyd and P. Surynek, “Parameter setting in sat solver using machine learning techniques,” (Praha 6, Czech Republic), Faculty of Information Technology, Czech Technical University, 2023. Faculty of Information Technology, Czech Technical University.
- [8] N. Eén and N. Sörensson, “Minisat github repository,” 2005. Source code and documentation for MiniSat.
- [9] N. Eén and N. Sörensson, “Minisat v1.14: A sat solver with conflict-clause minimization,” tech. rep., Chalmers University of Technology, 2005.
- [10] D. van Heesch, *Doxygen Manual*. Dimitri van Heesch and contributors, 1997–2023. Official documentation for Doxygen, a documentation generator for C++, C, Java, and other programming languages.
- [11] JetBrains, *CLion Documentation*. JetBrains, 2014–2023. Official documentation for CLion, a cross-platform IDE for C and C++ development by JetBrains.
- [12] N. Eén, *SAT Solving in Practice: Algorithms and Applications*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2006.
- [13] M. Heule, *Efficient SAT Solving: Theory and Practice*. PhD thesis, Delft University of Technology, Delft, Netherlands, 2015.
- [14] A. Biere, *Conflict-Driven Clause Learning SAT Solvers*. PhD thesis, Johannes Kepler University, Linz, Austria, 2012.
- [15] G. Audemard and L. Simon, “Glucose: A sat solver based on conflict analysis,” in *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, (Swansea, UK), pp. 21–25, 2009.

12 Contributii

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

Link-ul spre repository-ul de GitHub: <https://github.com/rainman226/minisat>