Scheduling Family Benchmark SAT Competition 2024

Berindeie Adrian, Buzuriu Roxana, Constantinescu Mihai, și Felea Irina

Universitatea de Vest din Timișoara
https://www.uvt.ro

Abstract. This paper focuses on optimizing the MiniSat solver for benchmarks from the SAT 2024 competition, focusing on conflict-driven clause learning (CDCL) algorithm and variable ordering heuristics. We propose enhancements to MiniSat's functionality, focusing on memory allocation improvements. Preliminary results indicate performance gains in execution time and memory usage, tested across two different configurations.

Keywords: SAT, Satisfiability, MiniSat, CDCL, DPLL, Solvers, Heuristics, SAT Competition

1 Introducere

Idei propuse pentru rulare si optimizare:

- Optimizare:
  - Optimizarea ordonării variabilelor: MiniSat folosește o metodă euristică pentru a decide ordinea variabilelor în timpul căutării. Astfel, o îmbunătățire ar fi să prioritizăm variabilele mai importante, adică cele care apar mai frecvent în conflictele recente.
  - Optimizarea memoriei alocate și reducerea dimensiunii alocate pentru clauze neutilizate.

Problemele de satisfiabilitate booleeană (SAT) sunt o componentă esențială a logicii computaționale, având aplicații extinse în domenii precum verificarea formală a sistemelor software si hardware, planificarea automată și optimizarea [2].

Solverele de tip SAT, cum ar fi MiniSat, sunt utilizate pentru a determina dacă o formulă booleană poate fi satisfăcută. Se poate confirma că este satisfăcută atunci când există o soluție, adică dacă s-a găsit o atribuire de valori de adevăr pentru variabile care să facă întreaga formulă adevărată.

MiniSat, este un solver SAT care se bazează pe algoritmul CDCL (Conflict-Driven Clause Learning)[9,8], fiind bazat pe învățarea de conflicte și reordonarea dinamică a variabilelor [5]. Acesta s-a remarcat prin eficiență și simplitate, astfel putem spune că a devenit un reper în dezvoltarea de soluții avansate pentru SAT.

Algoritmul CDCL folosit în MiniSat a adus îmbunătățiri semnificative față de DPLL (Davis-Putnam-Logemann-Loveland) [3,4], introducând mecanisme precum învățarea din conflicte, generarea clauzelor de conflict, deciziile euristice pentru selectarea variabilelor și reluarea adaptivă a căutării prin backtracking cronologic. Aceste îmbunătățiri au condus la o creștere semnificativă a eficienței în rezolvarea instanțelor complexe de SAT.

De-a lungul timpului, solverele CDCL au fost optimizate prin integrarea unor tehnici suplimentare, cum ar fi reordonarea dinamică a clauzelor și metodele de propagare folosind doi literali urmăriți [6].

Competițiile SAT oferă o platformă de testare riguroasă pentru evaluarea și compararea noilor soluții în rezolvarea problemelor SAT. Aceste evenimente adună dezvoltatori pentru a prezenta și testa cele mai recente soluții, contribuind astfel la progresul constant al acestui domeniu și evidențiind rolul esențial al implementărilor optimizate și a algoritmilor eficienți [7].

MiniSat și variațiile sale au fost supuse unor îmbunătățiri constante în cadrul acestor competiții, care includ strategii avansate de propagare și gestionare a clauzelor învățate. Aceste tehnici s-au dovedit esențiale pentru reducerea timpului de rulare și creșterea performanței generale a solverului. În acest context, proiectul nostru își propune să implementeze și să evalueze modificări specifice în MiniSat, axate pe optimizarea algoritmilor, cu scopul de a îmbunătăți eficiența în soluționarea problemelor.

Modificările propuse vor fi testate pe un set de benchmark-uri din familia scheduling, utilizate în competiția SAT 2024 [10], permițând o analiză detaliată a impactului asupra performanței și o comparație cu versiunea inițială a MiniSat. Vom folosi MiniSat simp pentru rularea fișierelor, o variantă îmbunătățită și simplifcată a MiniSat-ului core. Pentru un studiu cât mai elaborios, vom folosi 3 sisteme de operare diferite, Windows, Linux și MacOS. Prin această cercetare vom evidenția atât dificultățile întâmpinate, cât și posibilele direcții de optimizare viitoare pentru solverele de tip SAT.

2 Descrierea Problemei

-Detalii tehnice despre cum abordam problema găsită și variantele de optimizare, soluționarea problemei explicată tehnic.

3 MiniSat

3.1 Diferențe între MiniSat standard și MiniSat simp

MiniSat este un solver SAT open-source, apreciat pentru eficiența și flexibilitatea sa.Versiunea standard a MiniSat implementează algoritmul *Conflict-Driven Clause Learning* (CDCL), acesta utilizând tehnici esențiale precum propagarea unitate și învățarea din conflicte. Pe de altă parte, MiniSat simp extinde funcționalitățile

versiunii standard prin integrarea unor tehnici avansate de simplificare a formulelor SAT, precum eliminarea clauzelor redundante. Aceste tehnici reduc dimensiunea formulei și contribuie la îmbunătățirea performanței solverului, fiind utile în cazul problemelor complexe. MiniSat simp oferă avantaje clare, fiind utilizat frecvent în competițiile SAT și în cercetările care implică scenarii complexe. [11]

3.2 Instalare MiniSat

Instalare MiniSat pe Linux Pentru a instala MiniSat pe sistemul de operare Linux, este nevoie de clonarea proiectului de reporitory-ul de GitHub, folosind comanda prezentată în Secvența de cod 1.1. După clonarea proiectului, din directoriul MiniSat-ului, se rulează comanda prezentată în Secvența de cod 1.2 pentru a rula codul într-un mod mai permisiv. Această comandă setează un flag specific pentru compilatorul de C++, transformând erorile mai puțin critice în advertismente și astfel ajutând la compilarea codului. Comanda prezentată în Secvența de cod 1.3 este o comanda necesară pentru cazul în care sistemul nu are deja comanda "make", comandă necesară pentru a crea un executabil din codul C++. Dupa rularea comenzilor se navighează în locația executabilului prin comanda prezentată în Secvența de cod 1.4, iar din acest directoriu putem folosi programul MiniSat.

Secvența de cod 1.1: Clonare proiect

```
git clone https://github.com/irinafelea/minisat.git
```

Secvența de cod 1.2: Compilare permisivă

```
cd minisat/
make CXXFLAGS="-fpermissive"
```

Secvența de cod 1.3: Instalarea comenzii make

```
sudo apt install make
```

Secvența de cod 1.4: Navigarea către directoriul executabilului

```
cd build/release/bin
```

Instalare MiniSat pe Windows Instalarea MiniSat pe sistemul de operare Windows este precedată de instalarea unor programe și configurarea căilor de acces. Pentru început, este necesară instalarea subsistemului Unix pentru Windows, Cygwin[1], rularea fișierului `setup.exe`, în urma căruia Cygwin va crea directoarele în directoriul `../cygwin/`. Apoi este necesară descărcarea softwareului CMake[2] și extragerea fișierelor într-un director cu denumirea `C:\CMake`.După

[1] Cygwin `https://www.cygwin.com` [accesat în 02.11.2024]
[2] CMake `https://cmake.org/download/` [accesat în 03.11.2024]

realizarea instalărilor necesare, se configurează căile de acces în PATH. Pentru a configura căile de acces este necesară deschiderea setărilor sistemlui, iar în **"Environment Variables"**, la variabila `Path` sunt adăugate căi de acces către `MinGW\bin` și `CMake\bin`. Pentru a instala Minisat, acesta trebuie descărcat de pe pagina oficială a programului[3]. Se deschide subsistemul Cygwin și se navigează în directoriul Minisat. Din acest directoriu se rulează comanda prezentată în Secvența de cod 1.5, comandă care specifică generatorul de fișiere Makefiles pentru Minimalist GNU for Windows (MinGW) și creează un fișier Makefile compatibil cu `mingw32-make`. În Secvența de cod 1.6 este prezentată comanda care compilează fișierele generate anterior și creează executabilul `minisat.exe`.

Secvența de cod 1.5: Generare fișiere Makefile

```
cmake -G "MinGW Makefiles" ...
```

Secvența de cod 1.6: Generare fișiere Makefile

```
cmake --build
```

3.3 Detalii de implementare

Structuri de date folosite În această secțiune vom prezenta principalele structuri de date utilizate in cadrul aplicației MiniSat. Definirea acestor structuri de date a fost realizată în cadrul fișierului `minisat/mtl/SolverTypes.h`.

- `struct Lit` - Reprezentarea unui literal logic utilizând o variabilă de tip `int`, astfel făcând posibilă și stocarea semnului literalului.
- `class lbool` - Reprezentarea unei valori logice. Această clasă este creată pentru a optimiza comparația între o variabilă și o constantă, precum și pentru a se asigura că `gcc` realizează suficientă propagare a constantelor. Aceste aspecte sunt precizate pe linia 89. Această clasă mai conține în plus suprascrierea unor operatori pentru a ușura lucrul cu acest tip de variabilă în contextul rezolvării unei probleme de satisfiabilitate în logica computațională, cum ar fi: `==`, `!=`, `^`, `&&`, `||`. Aceste variabile au fost create cu scopul de a reprezenta 3 valori, chiar dacă acestea sunt definite de tipul `uint8_t`: `true`, `false`, `undefined`. Astfel, pe liniile 123–131, putem observa definirea următoarelor constante:
  - `l_True (lbool((uint8_t)0))`
  - `l_False (lbool((uint8_t)1))`
  - `l_Undef (lbool((uint8_t)2))`
- `class Clause` - Reprezentarea unei clauze logice. Acestă clasă reține informațiile de bază precum lista de literali și lungimea acesteia dar și alte informații mai avansate precum daca această clauză a fost învățată deja(proprietatea `learned`). Totuși este precizat că nu se poate folosi în mod direct contructorului acestei clase, ci se va folosi clasa prieten `ClauseAllocator` care se ocupă de alocarea corectă și eficientă a memoriei pentru acestă structură de date.

[3] The MiniSat Page `http://minisat.se/MiniSat.html` [accesat în 05.11.2024]

Pașii parcurși de algoritm În figura 1 este reprezentată diagrama de secvențe a programului MiniSAT. Analiza pașilor principali executați de programul MiniSat începe cu fișierul `minisat/core/Main.cc`. La începutul funcției `main()` putem observa cum programul analizează argumentele primite din linia de comandă și extrage calea către fișierul de intrare (dacă există) și opțiunile selectate cum ar fi: verbozitatea, limita de timp de execuție pe procesor permisă (în secunde), limita de memorie utilizată și alegerea de a valida sau nu antetul DIMACS în timpul parcurgerii (liniile 63-68). După acest pas programul inițializează solver-ul S, extrage timpul inițial, limitează resursele bazat pe argumentele primite și citește datele de intrare de la tastatură sau dintr-un fișier de intrare după caz. Pe linia 95 programul începe să parcurgă datele de intrare și să adauge clauze în interiorul lui S în cadrul funcției `parse_DIMACS`. Se încearcă simplificarea clauzelor. Daca acesta eșuează, se declară că formula este nesatisfiabilă iar execuția este oprită.

Rezolvarea propriu-zisă a problemei începe pe linia 124 prin apelarea metodei `S.solveLimited(dummy)`. Implementarea acestei metode se poate găsi în fișierul `minisat/core/Solver.cc`. Acestă metodă începe prin resetarea atributelor `model` și `conflict` urmată de verificarea dacă solver-ul este într-o stare validă. Căutarea unei soluții se realizează în interiorul buclei dintre liniile 865-870 care se execută atât timp cât valoarea variabilei `status` este `l_Undef`. Principala acțiune de căutare a unei soluții se realizează în cadrul metodei `search()` care va fi detaliată în subsecțiunea 3.3. După găsirea unui status acesta este procesat și afișat utilizatorului alături de restul statisticilor adunate în timpul execuției.

Detalierea funcțiilor principale

- `simplify()` - În implementarea curentă, acestă funcție doar elimină clauzele care sunt deja satisfăcute, reducând dimensiunea problemei, ajutând astfel la accelerarea procesului de căutare a unei soluții. Aceasta parcurge clauzele care sunt satifăcute cu valorile deja atribuite unor literali, până în acel moment al execuției.

- `pickBranchLit()` - Metoda selectează următoarea decizie, atât literalul cât și polaritatea acestuia. Se alege o valoare aleatoare întreagă între 0 și 1. Dacă acesta este mai mică decât varibila `random_var_freq` și heap-ul în care se reține variabilele neasignate nu este gol, atunci se alege o variabilă la întamplare din heap.
  Dacă acestă condiție nu este adevărată cât timp variabila `next` este egală cu `var_Undef` sau valoarea pentru `next` este diferită de `l_Undef` sau `decision[next]` înca este fals se verifică heap-ul. Dacă acesta este gol, lui `next` i se asignează valoarea `var_Undef` și se iese din buclă. Dacă nu, lui `next` i se dă valoarea minimă din heap.
  Pentru a se alege polaritatea, prima dată se verifică dacă `next` este `var_Undef`. În acest caz se returnează `lit_Undef`. Dacă nu, dacă există o polaritate aleasă de utilizator, acesta este aignată. În caz contrar, dacă variabila `rnd_pol`

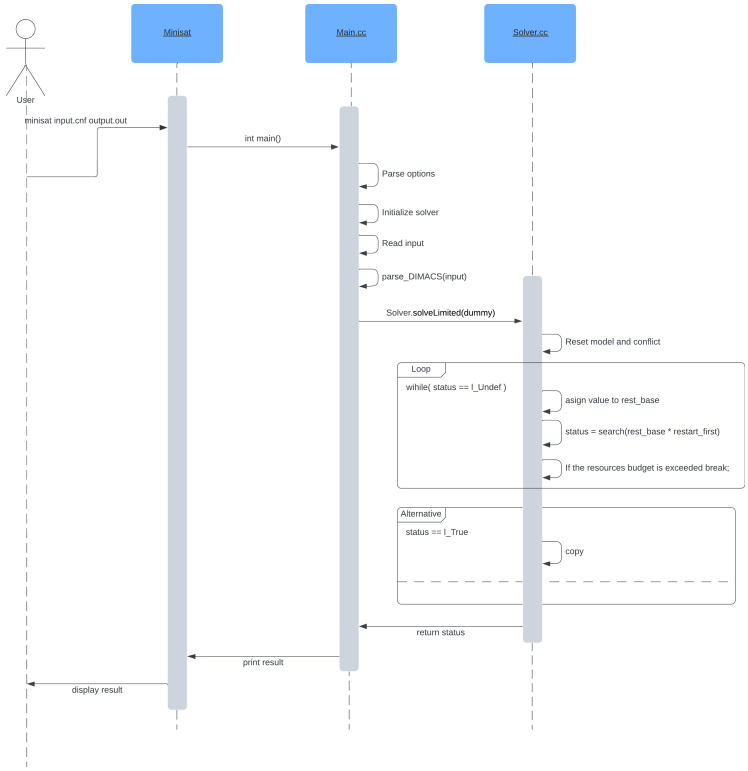

Fig. 1: Diagrama de secvență a programului

este asignată se alege o polaritate aleatoarea. Dacă nici una dintre condițiile de mai sus nu sunt împlinite se asignează o polaritate predefinită `polarity[next]`. Algoritmul folosit de această metodă este reprezentat în cadrul diagramei de activitate prezentată în figura 2

– `propagate()` - Metoda este responsabilă pentru propagarea unitate, atribuirea de valori bazată pe clauzele învățate pentru a reduce domeniul de căutare a unui model. În variabila `Lit p` se reține literalul curent de la care se pleacă cu propagare. Acesta este extras din coada `trail` care este parcusă cu ajutorul variabilei `qhead`. Pentru fiecare literal, funcția `propagate()` parcurge fiecare clauză. Dacă aceasta este deja satisfăcută, atunci se trece peste ea. Dacă nu este, se caută un literal care nu are o valoare atribuită pentru a încerca să satisfacă clauza. Dacă nu se găsește nici un literal care să satisfacă clauza se aplică din nou propagarea unitate (linia 554). În cadrul acestei funcții se mai execută și verificarea unor noi conflicte, iar dacă unul este găsit se va asigna în variabila `confl` referința către acest conflict(liniile 545-552).

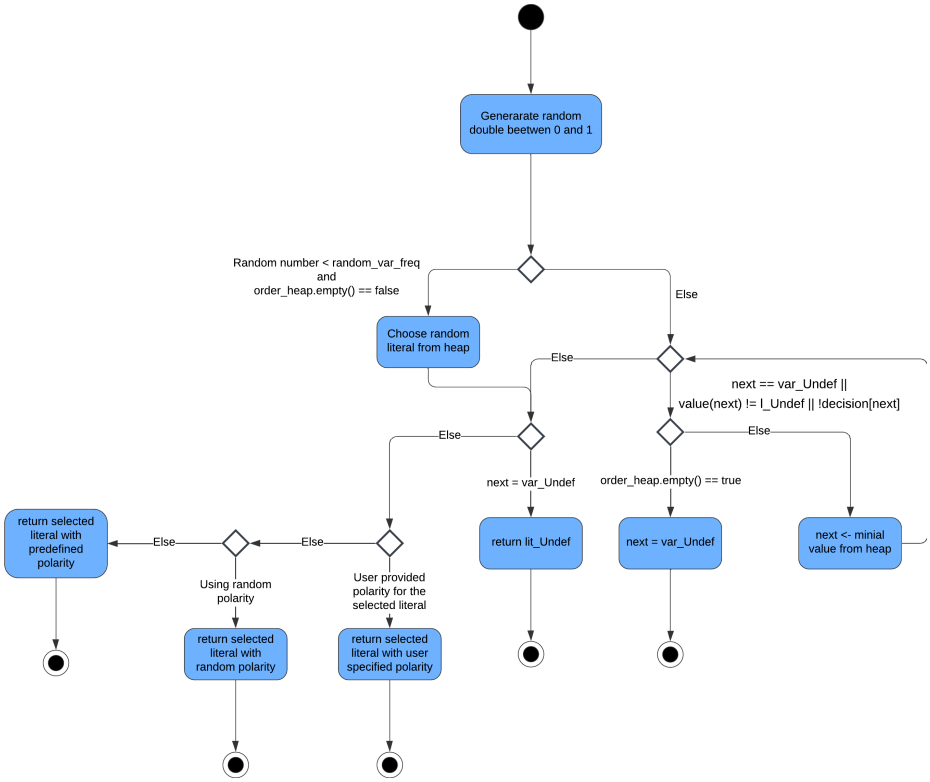

Fig. 2: Diagrama de secvență a programului

– `analyze(CRef confl, vec<Lit>& out_learnt, int& out_btlevel)` - Analizează conflictul transmis ca parametru pentru a produce claza care a produs acest conflict. Clauza găsită este adăugată în mulțimea de clauze învățate pentru a optimiza procesul de căutare a unui model, aplicându-se propagarea unitate peste acestă mulțime pentru a asigna valori unor literali bazate pe implicațiile rezultate din aceste clauze, ajutând la evitarea conflictelor întalinite deja în procesul căutării.
Metoda începe prin inițializarea variabilelor `pathC` și `p`(literalul curent analizat). În cadrul buclei `do-while` se analizează clauza de conflict curentă, iar dacă aceasta este deja învățată, i se crește activitatea. Pentru fiecare literal al clauzei dacă nu mai fost văzut până în acel moment și se găsește pe un nivel de decizie mai mare decât 0, atunci este marcat ca văzut. Dacă nivelul pe care se găsește acest literal este mai mare sau egal cu nivelul de decizie, se incrementează variabila `pathC`, iar dacă nu literalul este adăugat în variabila `out_learned`. După acest pas se caută urmatoarea clauză care va trebui analizată pe baza următorului literal din coada `trail`. Pentru acesta se va găsi clauza care l-a produs și va fi asignată către `conf`, iar apoi se va decrementa varibila `pathC`. Această buclă se oprește atunci când ajunge

varibila `pathC` 0. După terminarea buclei, se realizează simplificarea clauzei, urmată de găsirea nivelului corect de backtracking.

– `search(int nof_conflicts)` - Metoda începe prin declararea și inițializarea variabilelor pentru nivelul de backtrack, variabilei `conflictC` și clauzelor învățate, apoi se începe un ciclu infinit în care se încearcă găsirea unui model. Ciclul începe prin căutarea unei clauze de conflict cu ajutorul metodei `propagate()`. În cazul găsirii unui conflict se verifică dacă nivelul de decizie este 0, iar dacă această condiție este adevărată se returnează `l_False`, deducându-se că formula este satisfiabilă. Dacă acesta este diferit de 0, se resetează lista de clauze învățate și folosind metoda `analyze()` pentru a o repopula cu noile claze învățate care iau în considerare conflictul găsit. După analiză se revine la ultimul nivelul găsit aplicând metoda `cancelUntil()`.

  Dacă nu este găsit nici un conflict verificăm dacă ne aflăm în limitele setate, adică limita de conflicte, dacă acesta există și limitele de resurse(timp, memorie). Dacă una oricare dintre limită este depășită se va return `l_Undef`, îndicând că nu s-a putut studia complet formula în limitările setate. Dacă se trece de acestă verificare se încearcă simplificarea clauzelor. Dacă ne aflăm pe nivelul 0 și nu se poate realiza nici o simplificare atunci se va returna `l_False`. Următoarea acțiune care se realizează este reducerea clauzelor învățate daca numărul acestora depășește limita setata.

  După realizarea verificărilor menționate mai sus, se realizează o nouă decizie. Inițial se încearcă asignarea unei valori indicate de utilizator asupra unui literal. Dacă aceste presupuneri nu există se incearcă asignarea folosind metoda `pickBranchLit()`. Dacă și după această încercare de asignare nu a putut fi aleasă nici o valoare înseamnă că am găsit un model, deci vom returna `l_True`. Dacă s-a ales o valoare pentru un literal, incrementăm nivelul de decizie și adăugăm noul literal în coadă.

Identificarea algoritmilor de enumerare, DPLL sau CDCL În urma analizei realizatea asupra codului am putut deduce că aplicația implementează alogirtmul CDCL pentru a optimiza căutarea unui model. Acest algoritm are la bază algoritmul DPLL, care presupune la fiecare pas alegerea unei decizii, un literal și o polaritate pentru acesta, și se încearcă să se determine noi literali analizând restul clauzelor. Dacă se găsește un conflict, se urmăresc deciziile care au dus la acest conflict și se reiau luarea deciizilor de la cel mai mic nivel. Optimizările aduse de CDCL presupun învățarea literalilor care au provocat conflicte pentru a nu mai repeta din nou acele conflicte. Putem obesrva acest comportament în cadrul programului MiniSAT putem observa cum în metoda `search()` în cazul găsirii unui conflict se apelează metoda `analyze`, care actualizează clauzele învățate prin propagarea fiecărui literal din lista de conflicte. Aceste metode sunt prezentate mai detaliat în cadrul subsecțiunii 3.3

4 Descrierea familiei *scheduling*

În această familie de probleme, sunt abordate mai multe probleme de planificare. În fișierul *2a15a30186afdad41a49c5c5366d01be-Timetable_C_392_E_62_Cl_26_S_28.cnf* se abordează o problemă de **planificare a unui orar școlar**. În [12], autorii propun o abordare de realizare unui orar supus doar unor constrângeri tari care trebuie satisfăcute. Un curs este definit de o clasă (un grup de studenți) și o disciplină, iar fiecare curs trebuie să fie predat de un profesor, să aibă o sală și un anumit număr de locuri în sală. În plus, două cursuri predate de același profesor nu se pot desfășura în același timp, nu pot avea loc două cursuri în aceeași sală în același timp, profesorii pot avea intervale în care nu sunt disponibili, trebuie respectat numărul de ore alocate unui curs, iar orele cursului trebuie să fie apropiate, profesorii pot să nu fie calificați să predea acest acel curs și orele unui profesor trebuie să se întindă pe un număr limitat de zile. Autorii doresc să găsească pentru fiecare curs al fiecărei clase un profesor care să predea cursul, o sală în care să aibă loc acel curs și intervalele orare ale acestuia. Aceștia au denumit variabilele, seturile și constantele necesare rezolvării problemei, dar și constrângerile prezentate mai sus și le-au scris sub formă de expresie logică. Pentru această problemă au fost generate 20 de instanțe, rulate utilizând solverul **MapleLCMDistChronoBT** cu o limită de timp de 5000 de secunde și limită de memorie de 24 GB. Unul dintre cazuri a fost rulat folosind 272 de cursuri, 52 de profesori, 18 clase și 32 de săli, și s-a obținut un rezultat **satisifiabil** într-un timp de 80.65 secunde.

Un alt subiect abordat în această familie de probleme este **programarea optimă a sarcinilor paralele**. Autorii lucrării [1] definesc problema de planificare $P||C_{max}$ drept un set de sarcini n cu timpi de execuție cunoscuți w distribuite la un set de mașini identice *(m)* și propun o abordare neconstrânsă a acestei probleme. Aceșia au ordonat în ordine descrescătoare sarcinile *(j)*, în funcție de timpul de execuție și s-a căutat o modalitate de a asigna fiecărui procesor *(p)* o sarcină. S-a denumit câte o variabilă $a_{j,p}$ și s-au scris clauzele sub formă de propoziții logice, astfel încât unui procesor să îi fie asignată o singură sarcină. La final s-a verificat ca suma ponderilor atribuite fiecărui procesor nu depășește limita superioară considerată pentru fiecare procesor, care este echivalent cu constrângerea pseudo-booleană (PBC). Aceșia au redus această PBC la diagrama de decizie binară ordonată redusă (BDD). Diagrama reprezintă combinațiile de sarcini ce pot fi atrbuite unui proces, iar datorită faptului că este binară, aceasta poate produce doar răspunsuri *true* sau *false*. În plus a fost forțată unei sarcini primului procesor care are încărcare identică, apoi a fost forțată alocarea unei sarcini nealocate unui procesor căruia i-a rămas spațiu nealocat. Au fost considerate 11 instanțe cu un număr de sarcini cuprins între 20 și 220 și $n/m \in [2, 3]$, care au fost rulate utilizând solver-ul **KISSAT** cu o limită de timp de 500 de secunde, obținând pentru fiecare dintre acestea un rezultat nesatistifiabil. În urma acestei rulări au fost alese alte 9 instanțe care au produs rezultatul satisfiabil într-un timp cuprins între 13 și 54 de secunde.

5 Rezultatele experimentale

Pentru familia ***scheduling***, au fost rulatele fișierele specifice acestei familii.

Pentru rularea testelor, a fost setat un timp CPU limită de *1728 de secunde*. Testele au fost rulate pe diferite configurații:

1. Laptop **Macbook Air**:
   - Procesor **Apple M1**
   - Memorie RAM **8GB**
   - Sistem de operare **MacOS Sonoma 14.4.1**
2. Laptop **Lenovo V310-15IKB**:
   - Procesor **Intel Core i7-7500U**
   - Memorie RAM **8GB**
   - Sistem de operare **Windows 10 Home**

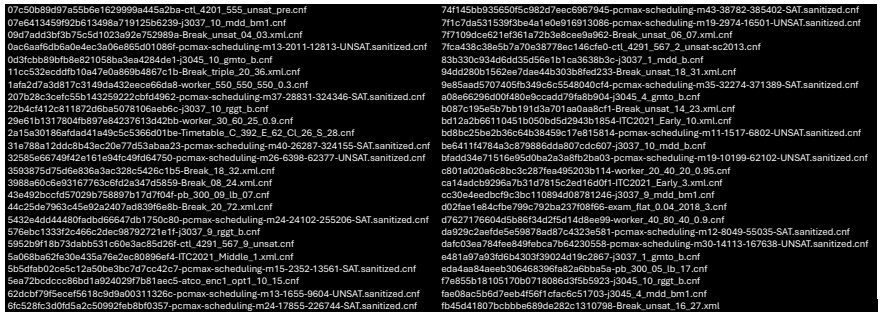

Fig. 3: Fișiere din familia *scheduling*.

5.1 Crearea de programe ajutătoare

Pentru a avea o mai bună întelegere a familiilor am realizat un program Python ce se află în folderul *pythonprogram* din proiectul prezent pe GitHub. Acest program utilizează Web Scraping pentru a colecta și număra fișierele pentru fiecare familie, stocând într-un folder cate un fișier cu numele familiei și o listă cu hash-urile pentru descărcarea fișierelor din acea familie.

Pe lângă programul Python, am avut o serie de programe scrise folosind Bash pentru Linux. Fisierul *downloadedfile.sh* preia toate fișierele pentru fiecare familie și le descarcă intr-o locație dată. La rularea programului se creează o altă locație cu fișierele descărcate, dezarhivate și redenumite. Pentru rularea unei familii am utilizat un timp de rulare prestabilit (de 24 de ore împarțit la numărul de fișiere) pentru fiecare fișier. Am utilizat un program simplu Bash care primește ca argument familia pentru care urmează să se execute programul MiniSat iar datele de ieșire se păstrau intr-un fișier de output pentru fiecare fișier denumit *runfamilyfiles.sh*

Table 1: Rezultatele testelor experimentale.

Scheduling								
			Configuratie 1			Configuratie 2		
Test	Variabile	Clauze	Timp CPU	Memorie	Răspuns	Timp CPU	Memorie	Răspuns
1	14756	141683	1109.72 s	200.94 MB	UNSAT	1826.18 s	226.49 MB	INDET
2	38781	362841	1659.87 s	381.07 MB	INDET	1873.77 s	1082.81 MB	INDET
3	14400	63874	17.3344 s	33.80 MB	UNSAT	38.1849 s	33.24 MB	UNSAT
4	2973	15516	1648.76 s	146.58 MB	INDET	1880.09 s	68.54 MB	INDET
5	221	1084	0.009421 s	4.85 MB	UNSAT	0.006469 s	5.97 MB	UNSAT
6	1101	5036	0.422917 s	9.49 MB	UNSAT	0.678463 s	6.23 MB	UNSAT
7	2010	11953	1465.29 s	141.27 MB	INDET	1818.18 s	53.30 MB	INDET
8	17850	149132	946.04 s	195.01 MB	UNSAT	1881 s	202.65 MB	INDET
9	14560	48075	1543.39 s	179.64 MB	INDET	1843.72 s	126.55 MB	INDET
10	21288	87245	694.386 s	161.01 MB	SAT	1885.32 s	134.71 MB	INDET
11	49869	264805	1686.52 s	463.18 MB	INDET	1852.69 s	1035.09 MB	INDET
12	37005	187726	1677.85 s	806.57 MB	INDET	1789.65 s	475.77 MB	INDET
13	93713	10295409	58.5334 s	1996.39 MB	SAT	146.975 s	1833.28 MB	SAT
14	32273	361039	1330.87 s	798.20 MB	INDET	1784.44 s	1630.84 MB	INDET
15	28830	320272	1621.04 s	1697.41 MB	INDET	1856.52 s	1659.16 MB	INDET
16	14128	46571	1717.94 s	161.27 MB	INDET	1784.44 s	98.07 MB	INDET
17	14174	66704	13.9012 s	43.11 MB	SAT	33.8056 s	29.14 MB	SAT
18	22365	83896	1715.57 s	298.08 MB	INDET	1795.28 s	252.23 MB	INDET
19	1015	21642	1706.28 s	327.14 MB	INDET	1829.38 s	271.76 MB	INDET
20	249117	764929	1314.89 s	2797.59 MB	INDET	1804.97 s	1710.56 MB	INDET
21	282525	1743751	449.656 s	2999.46 MB	INDET	1830.74 s	4562.01 MB	INDET
22	1516	6635	1122.38 s	121.11 MB	UNSAT	1811.21 s	40.80 MB	INDET
23	26286	319289	1691.84 s	823.04 MB	INDET	1834.96 s	1640.22 MB	INDET
24	14945	66518	14.4803 s	38.26 MB	SAT	34.4257 s	33.75 MB	SAT
25	6397	60575	246.293 s	96.22 MB	UNSAT	569.099 s	96.45 MB	UNSAT
26	10198	61395	1673.77 s	198.71 MB	INDET	1814.36 s	153.00 MB	INDET
27	37028	187771	1705.58 s	1323.73 MB	INDET	1841.8 s	716.64 MB	INDET
28	656	11934	0.011122 s	5.35 MB	SAT	0.007044 s	6.55 MB	SAT
29	3890	14187	0.068435 s	6.82 MB	SAT	0.064183 s	7.41 MB	SAT
30	29478	125712	0.106474 s	12.52 MB	SAT	0.170763 s	16.52 MB	SAT
31	145943	625454	20.1524 s	181.53 MB	SAT	73.6448 s	182.39 MB	SAT
32	19502	82348	1709.56 s	191.89 MB	INDET	1815.89 s	153.42 MB	INDET
33	53844	270141	1708.94 s	1572.19 MB	INDET	1832.5 s	1247.98 MB	INDET
34	1275631	1974386	1707.63 s	1204.72 MB	INDET	1821.6 s	686.98 MB	INDET
35	24101	254904	1684.74 s	829.19 MB	INDET	1820.72 s	1059.04 MB	INDET
36	2508	87544	1702.79 s	834.93 MB	INDET	1820.79 s	1702.12 MB	INDET
37	18913	85586	1705.65 s	183.61 MB	INDET	1837.52 s	149.83 MB	INDET
38	8048	54850	1707.95 s	281.02 MB	INDET	1837.36 s	180.82 MB	INDET
39	17850	154655	223.214 s	131.86 MB	UNSAT	544.334 s	158.11 MB	UNSAT
40	14112	166497	1381.68 s	259.28 MB	INDET	1829.94 s	229.40 MB	INDET
41	58907	193418	1675.12 s	400.32 MB	INDET	1851.66 s	690.25 MB	INDET
42	21015	69505	1138.31 s	152.57 MB	SAT	1841.53 s	156.52 MB	INDET
43	2351	13401	1415.36 s	180.14 MB	INDET	1842.87 s	123.18 MB	INDET
44	145862	624664	379.05 s	194.04 MB	SAT	1235.91 s	1593.70 MB	SAT
45	46759	250518	86.6242 s	99.90 MB	SAT	265.449 s	210.96 MB	SAT
46	14751	63112	1707.47 s	174.44 MB	INDET	1842.64 s	119.05 MB	INDET
47	1654	8801	1678.18 s	156.80 MB	INDET	1869.49 s	43.13 MB	INDET
48	13043	51999	1380.45 s	148.19 MB	INDET	1841.23 s	97.95 MB	INDET
49	17854	226553	1694.31 s	800.26 MB	INDET	1871.49 s	697.38 MB	INDET
50	31113	127844	1638.65 s	1227.83 MB	INDET	1851.76 s	380.50 MB	INDET

6 Provocări

7 Concluzie

Bibliografie

1. Matthew Akram and Dominik Schreiber. Optimal parallel task scheduling via sat. In *Proceedings of SAT Competition 2024 : Solver, Benchmark and Proof Checker Descriptions*, pages 30–31. Helda University of Helsinki, 2024.
2. Franc Brglez, XY Li, and MF Stallmann. On sat instance classes and a method for reliable performance experiments with sat solvers. *Annals of Mathematics and Artificial Intelligence*, 43:1–34, 2005.
3. Martin Davis, George Logemann, and Donald Loveland. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.
4. Martin Davis and Hilary Putnam. A machine program for theorem-proving. In *Communications of the ACM*, volume 5, pages 394–397. ACM New York, NY, USA, 1962.
5. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.
6. Tobias Fuchs, Jakob Bach, and Markus Iser. Active learning for sat solver benchmarking. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–425, Cham, 2023. Springer Nature Switzerland.
7. Marijn JH Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of sat competition 2024: Solver, benchmark and proof checker descriptions. 2024.
8. Jinbo Huang et al. The effect of restarts on the efficiency of clause learning. In *IJCAI*, volume 7, pages 2318–2323, 2007.
9. Lawrence Ryan. Efficient algorithms for clause-learning sat solvers. 2004.
10. SAT Competition Committee. SAT Competition 2024 Rules, 2024.
11. Niklas Sörensson and Niklas Een. Minisat v1.13-a sat solver with conflict-clause minimization. In *Proc. SAT*, volume 8, page 53, 2005.
12. Rodrigue Konan Tchinda and Clémentin Tayou Djamegni. A sat encoding of school timetabling. In *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, pages 97–99. Helda University of Helsinki, 2020.

8 Împărțirea responsabilităților în echipă

1. **Berindeie Adrian**
   - script pentru descărcare fisiere pe familii
   - script pentru pregătirea și rularea benchmark cu diferite configurații de fișiere
   - prelucrarea și rulare benchmark
   - analiza date obtinunte
2. **Buzuriu Roxana**
   - studiu lucrari științifice
   - împărțirea pe familii

   - structurare referat
   - introducere
   - instalare pe windows
3. **Constantinescu Mihai**
   - studiu cod - analiza structurii generale a codului, structurilor de date folosite, functiilor principale folosite in program
   - realizare diagrame
4. **Felea Irina**
   - studiu cod - analiza parametrilor utilizați pentru rularea fișierelor
   - împărțirea pe familii
   - studiu familii
   - studiu al lucrarilor științifice în care sunt descrise problemele abordate în familia aleasă
   - rulare benchmark
   - generare licență