Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Eina educativa de suport per l'estudi de resoledors SAT

Document: Memòria

Alumne: Marc Cané Salamià

Tutors: Mateu Villaret i Jordi Coll
Departament: Informàtica, Matemàtica Aplicada i Estadística(IMAE)
Àrea: Llenguatges i Sistemes Informàtics (LSI)

Convocatòria (mes/any): Setembre 2019

Index

1. Introducció....................5
 1.1 - Motivacions....................5
 1.2 - Objectius del projecte....................6
2 - Estudi de viabilitat....................7
 2.1 Coneixements necessaris....................7
 2.2 Costos de desenvolupament....................8
 2.3 Costos d'equipament....................8
3 - Metodologia....................9
4 - Planificació....................10
5 - Marc de treball i conceptes previs....................11
 5.1 Conceptes....................11
  5.1.1 El problema de satisfactibilitat booleana....................11
  5.1.2 Forma normal conjuntiva....................12
  5.1.3 CNFSAT....................13
  5.1.4 Propagació unitària....................13
  5.1.5 La regla de resolució....................14
  5.1.6 Aprenentatge de clàusules....................14
  5.1.7 Backjumping....................15
  5.1.7 Two-watched-literals....................16
 5.2 Algoritmes....................18
  5.2.1 Backtracking....................*18*
  5.2.2 DPLL....................18
  5.2.3 CDCL....................19
 5.3 Marc de treball....................20
6 - Requisits del sistema....................21
 6.1 Requisits funcionals....................21
 6.2 Requisits no funcionals....................21
  6.2.1 Requisits del maquinari....................21
  6.2.2 Requisits de programari....................21
7 - Estudis i decisions....................22
 7.1 Llenguatge de programació....................22
 7.2 Entorn de desenvolupament....................22
 7.3 API per la interfície gràfica....................23
  7.3.1 Dissenyadors gràfics....................24
 7.4 Draw.io....................24
 7.5 Tècniques incloses en els solucionadors....................24
8 - Anàlisi, disseny i implementació del sistema....................25
  8.1 Solucionadors....................25
   8.1.1 Solucionador inicial Backtracking....................25
   8.1.2 Solucionador inicial DPLL....................26
   8.1.3 Solucionador DPLL iteratiu....................26

8.1.4 Solucionador CDCL....28
8.1.4.1 Clause Learning....29
8.1.4.2 Backjumping....30
8.1.4.3 Two-watched-literals....31
8.1.4.4 Propagació unitària amb 2WL....33
8.1.5 Solucionador Backtracking final....34
8.2 Estructures dels solucionadors....35
8.2.1 Solver....36
8.2.2 BackjumpDelay....36
8.2.3 Event....36
8.2.4 Instance....37
8.3 Problemes i solucions durant la implementació....37
9 - Proves....39
10 - Resultats....40
10.1 Interfície de l'aplicació....40
10.1.1 Visualitzador del trail....41
10.1.2 Visualitzador de clàusules....42
10.1.3 El visualitzador d'events....43
10.1.4 Finestra d'anàlisi de conflicte....44
10.1.5 Botons interactius....45
10.2 Interfície de terminal....46
10.3 Normativa i legislació....47
11 - Conclusions....48
12 - Treball futur....49
13 - Bibliografia....50
14 - Annexos....51
14.1 Implementacions solucionadors inicials....51
15 – Manual d’usuari i/o instal·lació....55
15.1 Instruccions per executar l'aplicació....55

1. Introducció

A la Universitat de Girona s'ofereix el Grau en Enginyeria Informàtica on al quart curs els alumnes han d'escollir un dels 4 itineraris proposats.

En l'itinerari *Enginyeria de la Computació* es cursa l'assignatura *Programació declarativa. Aplicacions*, on s'estudien varies tècniques per resoldre problemes combinatoris.

Una d'aquestes tècniques és la reducció al *problema de satisfactibilitat booleana* o *SAT*, explicat més endavant.

A part d'aprendre com reduir els problemes a *SAT* també s'estudia el funcionament dels solucionadors *SAT* i les tècniques que aquests solucionadors utilitzen.

Entendre el funcionament dels solucionadors *Conflict-Driven Clause-Learning (CDCL)*, especialment les tècniques de *Clause learning* i *Backjumping,* pot ser complicat al principi.

L'objectiu del projecte és crear una eina de caràcter educatiu, interactiva i visual a disposició dels estudiants i el professorat per tal de facilitar l'estudi dels solucionadors *SAT*.

S'han ajuntat els apartats d'anàlisi, disseny i implementació en un de sol perquè han estat molt relacionats en el desenvolupament i ha permès facilitar-ne la lectura.

1.1 - Motivacions

M'interessa el món dels problemes combinatoris perquè són generalment problemes desafiants on quasi mai hi ha una solució perfecte.

Com que són problemes que porten estudiant-se des de fa temps existeixen una gran varietat d'eines i utilitats per atacar-los, que estan en continuu procés de millora i refinament. *SAT* és una bona opció per resoldre molts d'aquests problemes.

Estudiar les estratègies que s’utilitzen per fer-hi front és interessant per entendre l’estat de l’art, els punts forts i febles i idear possibles millores.

Vaig veure aquest projecte com una oportunitat d'aprofundir en el camp dels problemes combinatoris i aprendre sobre les tècniques (algoritmes, estructures d'acceleració, heurístics,...) que s'utilitzen per resoldre'n un dels problemes centrals.

1.2 - Objectius del projecte

La idea era fer una eina per facilitar l’estudi dels algoritmes que s’usen en els solucionadors SAT moderns, en concret, per als estudiants de l’assignatura de *Programació declarativa. Aplicacions.*

Aquesta eina de suport havia d’ajudar a l’alumnat a poder traçar visualment les passes de resolució que feia l’algoritme per arribar a una solució, de manera que la pogués usar per entendre el seu funcionament, validar els resultats dels exercicis proposats a classe i trobar l’errada en cas d’error.

A l’assignatura s’expliquen 3 algoritmes per resoldre SAT, que augmenten en complexitat i potència. El primer és usar l’esquema de *backtracking*, molt lent i ineficient, el segon és l’algoritme de *DPLL* (*Davis–Putnam–Logemann–Loveland*), la base de gairebé tots els solucionadors SAT i finalment el *CDCL* (*Conflict Driven Clause Learning*), una evolució que incorpora les tècniques *aprenentatge de clàusules* i *backtracking no cronològic.*

L'objectiu principal del projecte és el desenvolupament d'aquesta eina per estudiar els solucionadors *SAT.* L'aplicació, que s'anomena *SAT-IT* (de SAT *Interactive Tracer*), tindrà els següents elements*:*

- Tres solucionadors que implementaran els diferents algoritmes vistos a classe
- Una interfície gràfica interactiva per poder seguir les passes dels solucionadors
- Una interfície en mode terminal per poder obtenir solucions al problema *CNFSAT*

2 - Estudi de viabilitat

En aquest apartat s'estudien els requisits i possibles costos de dur a terme aquest projecte.

2.1 Coneixements necessaris

Hagués sigut quasi impossible dur a terme aquest projecte sense la formació rebuda al llarg del grau, especialment per els continguts apresos en les següents assignatures:

- Estructures de dades i algorítmica
- Paradigmes i llenguatges de programació
- Enginyeria del software II
- Programació declarativa. Aplicacions
- Estades en entorn laboral

Els continguts imprescindibles apresos en aquestes assignatures són: *Java* i *backtracking*, paradigma funcional, disseny de classes, *SAT* i *Scala*, desenvolupament d'aplicacions amb interfície gràfica.

2.2 Costos de desenvolupament

Els costos de desenvolupament s'han comptat a posteriori. El temps reflectits als apartats són una aproximació. S'ha fet servir la xifra de 15 euros la hora per calcular el cost.

Tasca	Hores	Preu (€)
Backtracking i DPLL inicials	20	300
Estudiar estat de l'art SAT	15	225
Reimplementació DPLL	20	300
Implementació CDCL	30	450
Implementació 2WL	30	450
Depurar CDCL	80	1200
Estudiar opcions per la interfície	20	300
Implementació interfície	100	1500
Total	305	4725

2.3 Costos d'equipament

Aquests costos inclouen el maquinari i programari utilitzats al llarg del projecte.

Tot i no ser un requisit indispensable, per treballar amb més agilitat es recomana disposar d'un ordinador modern amb 8 GB de RAM o més.

En aquest cas ja es disposava del maquinari necessari.

Concepte	Preu (€)
Ordinador	0
GNU/Linux	0
IntelliJ Idea Community	0
IntelliJ Idea Scala Plugin	0
Firefox	0
Notepadqq	0
LibreOffice	0
Draw.io	0
Total	0

3 – Metodologia

La metodologia escollida per realitzar aquest projecte s'anomena *extreme programming (XP).*

Té diverses característiques però la més important és que prioritza l'adaptació per sobre la previsió. Aquesta característica la fan molt atractiva perquè quan es treballa amb interfícies gràfiques acostumen a aparèixer desviacions al pla original. La poca rigidesa d'algunes decisions que es prenen fan necessària una metodologia que s'adapti progressivament.

Degut a la gran quantitat d'opcions que es tenen quan es treballa amb llibreries grans mai pots estar segur d'encertar la bona decisió a la primera i és necessària la ràpida iteració per perdre com el menor temps possible. Aquesta metodologia permet aprofitar les prestacions que ofereixen les llibreries sense veure frustrada tota la previsió inicial degut a alguna limitació que no es podia conèixer abans de començar.

Extreme programming es regeix pels següents valors:

- Comunicació: el codi ha de ser net perquè tots els integrants del projecte el puguin entendre. El codi es modifica sovint i progressivament, per tant els comentaris han de ser justos i necessaris, sobretot en les parts que difícilment es canviaran. El codi però ha d'estar ben autodocumentat: els noms de classes i variables han de ser útils i significatius i que han de donar informació de la seva funcionalitat. Una pràctica comuna d'aquesta metodologia és la programació en parelles. La comunicació amb el client ha de ser freqüent i àgil perquè ha de ser aquest que ha d'anar aprovant les parts desenvolupades.
- Simplicitat: els dissenys complexos fan que el cost de manteniment augmenti en relació a la complexitat. Per això una de les idees de la metodologia és fer els dissenys el màxim de senzills possible.
- Retroalimentació: el desenvolupament de l'aplicació es fa per cicles. En cada cicle s'implementen unes funcionalitats i després de cada cicle el client aprova la part implementada o en proposa modificacions. La filosofia d'extreme programming és integrar el client en el desenvolupament del projecte, per mantenir el contacte constant i avançar en cicles curts, que minimitzi el cost que pot comportar una rectificació per part del client.

- Valentia: és necessària quan cal refer codi que ja estava fet, encara que s'hagi de perdre feina feta que ha comportat hores de treball.

El rol del client l'han exercit els tutors, que alhora són els professors de l'assignatura que utilitzarà aquesta aplicació. Tot i així no se m'han limitat les opcions ni les decisions.

De tots els valors de la metodologia aquest projecte s'ha basat en els de simplicitat, retroalimentació i valentia.

Simplicitat perquè sempre s'han prioritzat les solucions simples per davant les complexes, ja que acostumen a donar millors resultats .

A diferència del que marquen els valors de retroalimentació no sempre s'han pogut fer cicles de curta durada i la comunicació amb el client ha sigut inferior a la optima.

La valentia ha sigut necessària per refer el codi després de cada cicle. Sovint el codi refet no s'ha pogut reaprofitar però si que s'ha aprofitat l'experiència adquirida durant el seu desenvolupament.

El projecte ha diferit de la metodologia pel que fa a la comunicació. Com que el treball és individual no s'ha pogut fer treball per parelles ni s'han tingut en compte algunes pràctiques necessàries quan es treballa en equip. El que si s'ha intentat sempre és que el codi sigui autodocumentat, no només per facilitar futures ampliacions si no perquè també pugui ser entès per altres desenvolupadors.

4 - Planificació

Aquest diagrama està fet *a posteriori.*

Les franges són aproximades i no reflecteixen amb amb precisió la realitat, ja que moltes tasques no tenien un inici i un fi clarament definit i això les fa difícils de plasmar en el diagrama.

	Abril				Maig				Juny				Juliol				Agost			
Nom de la tasca	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
Backtracking i DPLL inicials																				
Estudiar estat de l'art SAT																				
Reimplementació DPLL																				
Implementació CDCL																				
Implementació 2WL																				
Depurar CDCL																				
Estudiar opcions per la interfície																				
Implementació interfície																				
Documentació																				

5 - Marc de treball i conceptes previs

En aquest apartat s'expliquen els coneixements essencials per poder entendre el projecte.

5.1 Conceptes

Aquests algorismes de cerca consisteixen en anar construint la solució incrementalment mitjançant l'assignació de valors de veritat a les variables Booleanes (*decisions*) i reculant (*backtraking*) quan es detecta que la solució parcial no podrà ser estesa a una solució total (*conflicte*). Aquesta cerca típicament es pot representar mitjançant el que es coneix com a l'arbre de cerca. Donat un node de l'arbre, direm que el seu *nivell de decisió* és el del número de decisions des de l'arrel fins al node considerat.

5.1.1 El problema de satisfactibilitat booleana

El problema de satisfactibilitat booleana o *SAT* és el problema de determinar si existeix una interpretació que satisfà una fórmula booleana donada. Dit d'altre manera, aquest problema respon a la pregunta de si les variables d'una fórmula donada poden ser substituïdes consistentment pels valors *cert* o *fals* de tal manera que la fórmula s'avaluï a *cert.*

Aquest problema va ser el primer que es va demostrar pertànyer a la classe de complexitat *NP-Complet.* Tot i ser un problema intractable en el cas general, moltes instàncies de problemes reals poden ser resoltes en un temps raonable.

Això és degut a que la seva gran popularitat i senzillesa l'han convertit en un dels problemes més estudiats. Fruit d'aquesta recerca han aparegut molts solucionadors potents que són capaços de resoldre instàncies amb milers de variables i milions de símbols.

Els solucionadors SAT tenen aplicacions pràctiques en problemes d'intel·ligència artificial, disseny de circuits i demostracions automàtiques de teoremes.

5.1.2 Forma normal conjuntiva

Diem que una fórmula proposicional està en *forma normal conjuntiva* (en anglès c*onjuntive normal form* o *CNF*) si la fórmula és una conjunció d'una o més clàusules que només contenen disjuncions; dit d'altre manera, si és una *AND* de *ORs*.

Una fórmula proposicional és una combinació de variables booleanes i operadors lògics:

$$x1 \rightarrow (\overline{x2} \vee x3) \wedge (x3 \rightarrow x1)$$

Sempre té representació equivalent en CNF (conjunció de AND).

Una clàusula és una disjunció (OR) de literals positius $(x1, x3)$ o negatius $(\overline{x1}, \overline{x2}, \overline{x3})$:

$$(\overline{x1} \vee \overline{x2} \vee x3)$$

Una fórmula en CNF és una conjunció (AND) de clàusules:

$$(\overline{x1} \vee \overline{x2} \vee x3) \wedge (\overline{x3} \vee x1)$$

Una interpretació és una assignació de variables booleanes:

$$\{\ x1 \rightarrow 0, x2 \rightarrow 0, x3 \rightarrow 1\ \} \text{ o } \overline{x1}\,\overline{x2}\,x3$$

Una clàusula està satisfeta sota una interpretació si almenys un dels seus literals pertany a la interpretació. Quan una interpretació satisfà una clàusula, direm que és un *model*.

$$\overline{x1}\,\overline{x2}\,x3 \text{ és un model per } (x1 \vee \overline{x2} \vee x3)$$

$$\overline{x1}\,x2\,\overline{x3} \text{ no és un model per } (x1 \vee \overline{x2} \vee x3)$$

Qualsevol fórmula proposicional es pot transformar a un equivalent en *CNF*. Algunes fórmules proposicionals però, poden expandir-se exponencialment al ser transformades directament a *CNF*. Afortunadament es coneixen mètodes com la *transformació de Tseytin*, que donada una fórmula proposicional qualsevol, permeten obtenir una fórmula equisatisfactible en *CNF* en

temps polinòmic a canvi d'introduir un nombre lineal de variables auxiliars. Així doncs, sempre es pot obtenir una fórmula en *CNF* sense augmentar la complexitat del problema.

5.1.3 CNFSAT

CNFSAT és el mateix problema que *SAT* però les fórmules proposicionals estan en CNF

Per comoditat i senzillesa la gran majoria de solucionadors treballen amb les formules *CNF*.

La transformació en si no redueix la complexitat del problema però facilita la implementació dels solucionadors, que poden fer servir estructures de dades més simples directament. També permet aplicar regles d'inferència durant el procés de solució com ara la *propagació unitària* o la *resolució*.

5.1.4 Propagació unitària

La propagació unitària (en anglès *Unit Propagation* o *UP*) és un procediment que pot simplificar un conjunt de clàusules proposicionals.

El procediment es basa en les *clàusules unitàries*, és a dir, clàusules compostes per un sol literal. Quan una clàusula té un sol literal, o tots els literals menys un estan assignats a fals, el literal no assignat ha de ser necessàriament cert.

Si un conjunt de clàusules contenen la clàusula unitària amb el literal *l*, les altres clàusules del conjunt es poden simplificar aplicant les següents dos regles:

1. Cada clàusula que contingui *l* , es pot eliminar (la clàusula està satisfeta si *l* ho està).
2. Es pot eliminar el literal *-l* de cada clàusula on aparegui (ja que no pot contribuir a satisfer la clàusula).

Per exemple, podem aplicar propagació unitària sobre el següent conjunt de clàusules:

$\{ a \vee b, \neg a \vee c, \neg c \vee d, a \}$

La primera clàusula és unitària i conté a , així que la podem eliminar perquè queda satisfeta. La segona conté el literal $\neg a$, que es pot eliminar perquè no pot contribuir a satisfer la clàusula. La tercera no conté a ni $\neg a$ així que es queda igual.

Degut a la clàusula unitària a podem inferir que a ha de ser cert. Això desencadena que c ha de ser cert degut a la $\neg a \vee c$. També utilitzant propagació unitària, d ha de ser cert degut a la clàusula $\neg c \vee d$.

El conjunt de clàusules resultant és el següent, on verd vol dir cert i vermell vol dir fals.

$\{ a \vee b, \neg a \vee c, \neg c \vee d, a \}$

5.1.5 La regla de resolució

La *regla de resolució* o simplement *resolució* és una regla d'inferència que, aplicada iterativament, permet determinar la insatisfactibilitat d'una fórmula proposicional. Dit d'altre manera, és *refutacionalment completa* en lògica proposicional.

Aquesta regla només es pot aplicar sobre clàusules. És a dir, que si volem resoldre una fórmula proposicional usant *resolució*, aquesta haurà d'estar en CNF.

La regla es pot entendre amb el següent exemple:

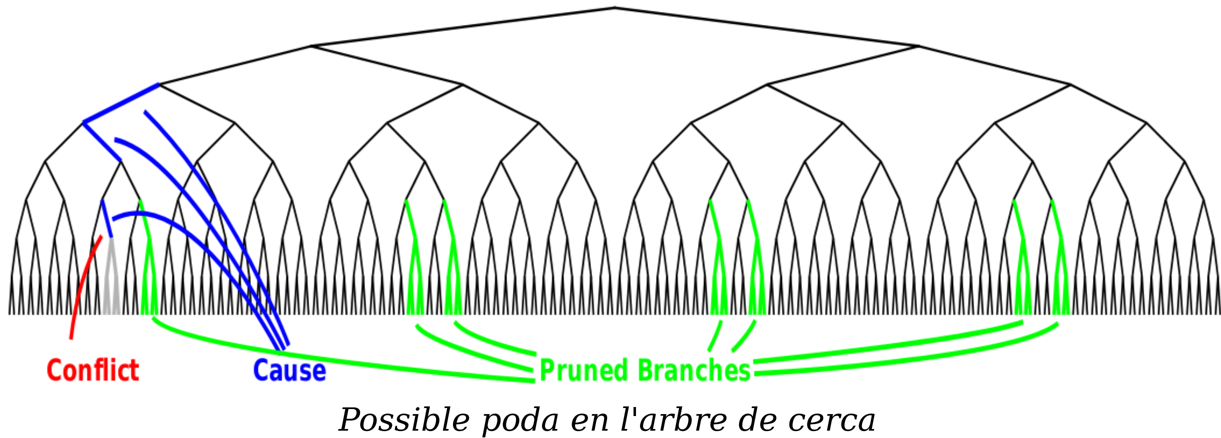
Tenim la fórmula { $a \vee b$, $\neg a \vee c$ }. D'aquesta fórmula en podem treure la conclusió $b \vee c$ ja que la variable a no pot ser *certa* i *falsa* alhora. Perquè la fórmula sigui certa s'ha de satisfer b , c o ambdues alhora.

5.1.6 Aprenentatge de clàusules

L'*aprenentatge de clàusules* o *clause learning (CL)* és un procediment que afegeix clàusules durant la resolució per dirigir la cerca i evitar branques de l'arbre de cerca que no ens poden portar a un resultat. Com que les clàusules apreses són conseqüència lògica de les clàusules originals del problema no es modificaran les solucions del mateix.

L'aprenentatge de clàusules es un component crucial en l'algorisme CDCL (explicat més endavant), que es produeix al detectar un conflicte durant la cerca.

Aquestes clàusules apreses ens poden ajudar a evitar fer fallades en la cerca pels mateixos motius, reduint així l'espai de cerca.



Possible poda en l'arbre de cerca

Hi ha varis mètodes per decidir quina clàusula aprendre. Un dels més usats és el *first unique implication point* (*1-UIP*).

Aquest mètode consisteix en aplicar resolució iterativament fins a aconseguir una clàusula amb un sol literal de l'últim nivell de decisió, és a dir, tota la resta de literals han estat assignats abans d'alguna decisió.

Començarem amb la clàusula que ens ha fet fallar, buscarem la clàusula que ha propagat la última variable propagada, aplicarem resolució entre les dues clàusules, això ens eliminarà la variable. Agafarem la clàusula resultant i repetirem el procés fins que només quedi un sol literal de l'ultim nivell de decisió.

La clàusula resultant ens permetrà evitar repetir l'assignació de variables que ens ha causat el conflicte.

5.1.7 Backjumping

El *backtracking no cronològic* o *backjumping* és una tècnica que ens permet recular nivells de decisió que ja sabem que no seran útils d'explorar, reduint així l'espai de cerca i millorant l'eficiència de l'algoritme.

Un cop fet l'*anàlisi de conflicte* i apresa la nova clàusula *c*, toca desfer les assignacions que ens han portat al conflicte. El nivell de decisió fins on hem de recular és el nivell més profund en el qual tots els literals menys un de la clàusula apresa han estat assignats.

Dit d'altre manera, saltarem al nivell de decisió més baix que podem però assegurant que *c* força una propagació unitària.

En el pitjor dels casos només saltarem un nivell de decisió enrere, tal com faria un *backtrack,* en el millor dels casos eliminarem tot l'arbre de cerca i saltarem al nivell de decisió zero.

Això és conegut com un *top level assignment* (*TLA)* i passa si la clàusula apresa *c* conté només un literal. La recerca suggereix que la quantitat de *TLA*s és un indicador de la dificultat del problema. [**TFG02**]

5.1.7 Two-watched-literals

El *two-watched-literals* (*2WL*) és una tècnica que permet accelerar la propagació unitària.

L'estratègia ingènua per detectar propagacions unitàries és recórrer totes les clàusules cercant aquelles que no estan satisfetes i contenen només un literal indefinit. L'estratègia 2WL permet només recórrer un subconjunt de les clàusules on apareix un literal recentment assignat sense perdre completesa.

Aquesta tècnica necessita estructures de dades per marcar 2 literals de cada clàusula i per saber, per cada literal, a quines clàusules apareix observat. Aquesta última estructura es coneix com a *llistes d'ocurrències.*

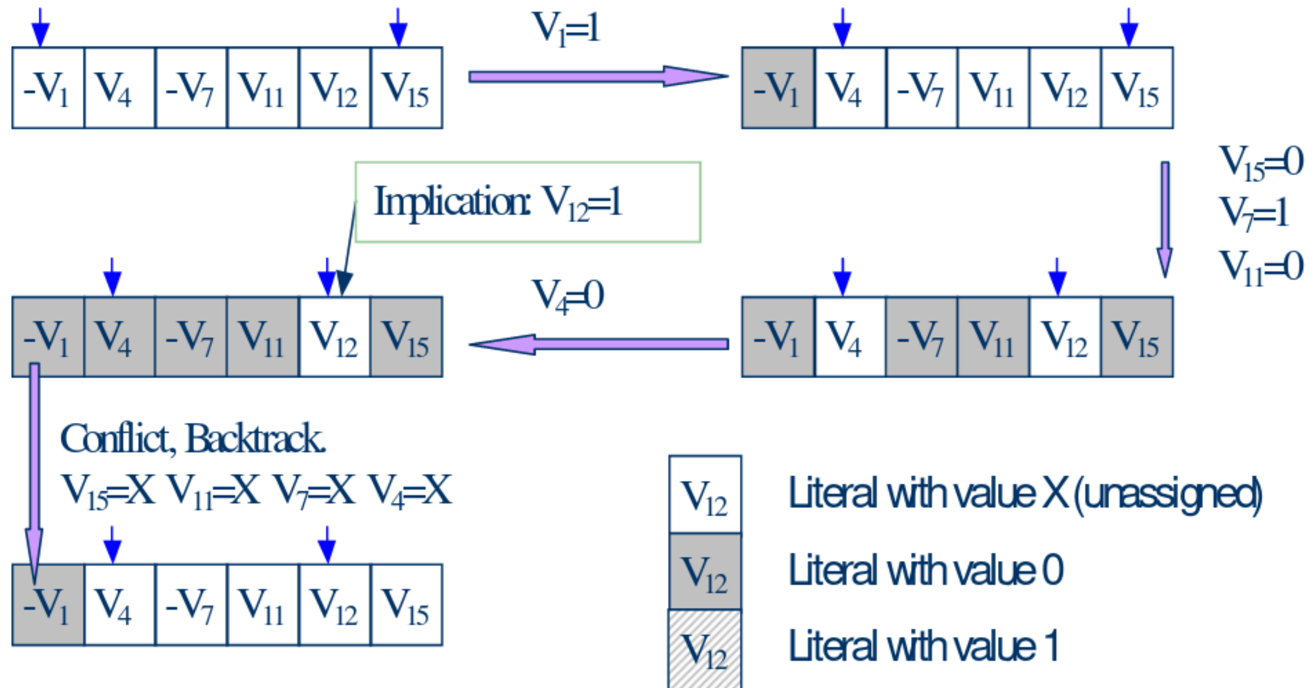
La idea en la que es basa tota la tècnica és que una clàusula amb dos literals no falsos (certs o indefinits), no pot ser unitària ni conflictiva.

Quan propaguem un literal *l* només ens caldrà visitar les clàusules on *-l* apareix observat*,* que estan guardades a les llistes d'ocurrències.

Si no existeixen 2 literals no-falsos vol dir que:

- Tots són falsos, i per tant hem trobat un conflicte.
- Són tots falsos menys un que està indefinit, i per tant hem de fer propagació unitària
- Són tots falsos menys un que és cert, per tant la clàusula està satisfeta

Una característica important que té aquesta tècnica és que les llistes d'ocurrències són estructures *lazy,* és a dir, que no cal actualitzar-les quan fem *backtracking* o *backjumping*, ja que si eliminem assignacions fetes en ordre cronològic els literals que teníem observat continuaran complint les condicions del *2WL.*



Exemple de funcionament 2WL [TFG06]

En la figura es veu un exemple de l'evolució d'una clàusula que usa la tècnica dels 2WL. Les fletxes blaves indiquen que el literal està observat. Comencem amb els literals -V1 i V15 observats. Assignem la variable V1 a cert i el literal -V1 que tenim observat passa a ser fals, per el que hem de buscar un altre literal no definit per observar, trobem V4 i passem a observar-lo. Es fan 3 assignacions més. V11 i V7 no ens importen perquè no estan observats. L'assignació V15=0 ens falsifica el nostre literal observat, busquem un literal no definit, trobem V12 i passem a observar-lo. Es fa l'assignació V4=0 que ens falsifica un dels literal observats i ja no trobem cap literal no definit. Això implica que el literal V12 que continua observat és unitari i s'ha de propagar. La propagació ens crea un conflicte, fem backtrack i eliminem vàries assignacions però els literals observats no s'han de moure perquè el backtrack segur que ens ha desfet almenys l'última assignació V4.

5.2 Algoritmes

Aquí s'expliquen els 3 algoritmes implementats a l'aplicació.

5.2.1 Backtracking

El *backtracking* és un algoritme genèric per trobar una, vàries o totes les solucions a alguns problemes combinatoris, usualment problemes de satisfacció de restriccions.

L'algoritme va generant les solucions incrementalment i torna enrere (fa *backtracking*) quan veu que la solució parcial no pot formar una solució vàlida. Quan es fa *backtracking* es torna fins a l'ultima decisió feta i s'elegeix el següent candidat, en cas de que en quedin més.

El fet de provar tots els candidats el converteix en un algoritme complet.

El *backtracking* només és aplicable per problemes on es puguin descartar solucions parcials i normalment és més ràpid que buscar la solució per força bruta generant tots els possibles estats finals.

Conceptualment es poden imaginar els candidats parcials del problema com a nodes d'un arbre, l'arbre de cerca. L'algoritme recorre aquest arbre de cerca recursivament des de l'arrel fent una cerca per profunditat.

A cada node n l'algoritme comprova si n és una solució parcial vàlida. Si no ho és tot el subarbre que penjaria d'n es salta (o poda). Altrament es comprova si n és una solució completa vàlida i en cas afirmatiu es mostra o es guarda.

L'algoritme de *backtracking* es pot utilitzar per resoldre SAT però és molt ineficient perquè s'exploren molts camins que es poden anticipar com a falsos.

5.2.2 DPLL

El *DPLL* (*Davis–Putnam–Logemann–Loveland*) és un algoritme complet amb una cerca basada en la del *backtracking* per decidir la satisfactibilitat d'una formula de lògica proposicional en *CNF*.

Aquest algoritme millora l'eficiència del *backtracking* fent ús de la regla de propagació unitària.

Si una clàusula no satisfeta és unitària, és a dir, que només conté un literal no assignat, només es podrà satisfer si aquest literal s'avalua a cert, per tant no cal decidir sobre el valor d'aquella variable. Això ens redueix l'espai de cerca de manera determinista, sense perdre solucions i mantinguent la completesa.

Sovint les propagacions unitàries creen una cascada de propagacions.

El problema del DPLL és que podem fallar múltiples vegades per culpa del mateix error (una assignació concreta de variables que generen conflicte) i hauríem de poder evitar-ho.

5.2.3 CDCL

El *CDCL* (*Conflict-Driven Clause Learning)* és una millora del *DPLL* que inclou les tècniques de *clause learning* i *backjumping* explicades anteriorment.

Tot i fer servir tècniques per podar l'espai de cerca l'algoritme manté la completesa.

El *CDCL* usa la *regla de resolució* per inferir clàusules que ajuden a dirigir la cerca i a saltar parts de l'arbre de cerca.

La diferència principal respecte l'algorisme DPLL rau en com s'actua quan es detecta un conflicte. Mentre que en el DPLL s'inverteix l'última decisió presa, en el CDCL s'inicia l'*anàlisi de conflicte.* El resultat de l'anàlisi de conflicte serà una clàusula nova que s'aprendrà, i que ens permetrà identificar el nivell al que ha de recular el backjumping.

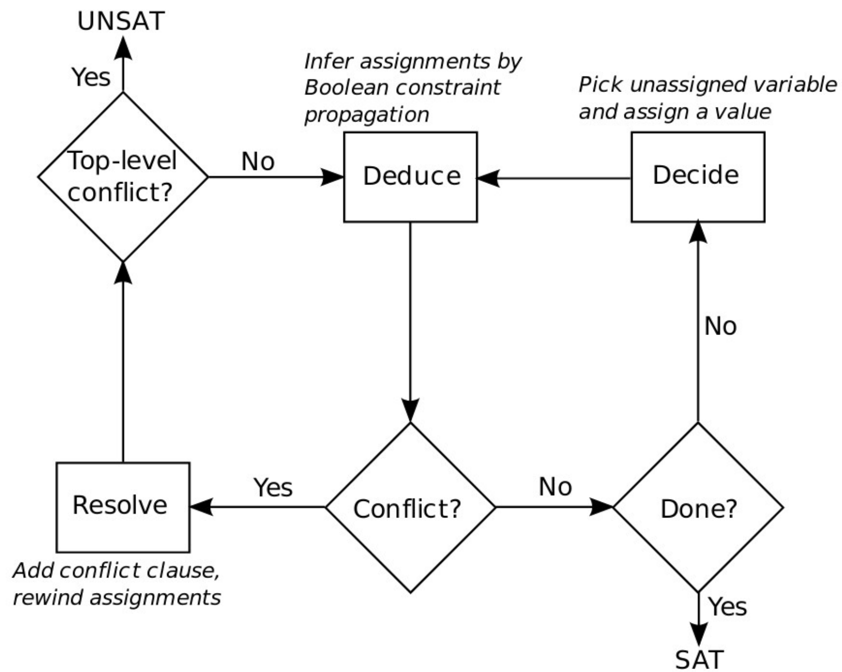


Diagrama d'activitat d'un solucionador CDCL [TFG05]

5.3 Marc de treball

Aquest projecte s'ha realitzat per al grup de recèrca de Lògica i Programació, i no hagués sigut possible sense l'ajuda dels tutors Dr. Mateu Villaret i Dr. Jordi Coll que m'han donat suport amb propostes, ajudat en les decisions, resolt dubtes i col · laborat a abordar els problemes que han anat sortint al llarg del projecte.

6 - Requisits del sistema

6.1 Requisits funcionals

Els requisits funcionals de l'aplicació són els següents:

- L'usuari ha de poder resoldre instàncies del problema de satisfactibilitat booleana
- L'usuari ha de poder triar entre els 3 mètodes de resolució que s'estudien a classe (Backtracking, DPLL, CDCL)
- L'usuari ha de poder avançar per les passes de resolució interactivament
- L'usuari ha de poder estudiar les tècniques de *lemma learning, backjumping* i *Two-watched-literals*
- L'usuari ha de poder veure les diferents tècniques en funcionament

6.2 Requisits no funcionals

Per executar l'aplicació s'han de complir els següents requisits:

6.2.1 Requisits del maquinari

Sistema operatiu: Qualsevol distribució de *GNU/Linux* amb entorn gràfic. Windows i Mac OS també estan suportats però no s'han testejat.

Processador: Qualsevol processador on es puguin satisfer els requisits de programari.

Memòria: Mínim 4GB. Recomanat 8GB

6.2.2 Requisits de programari

- *Java SE Runtime Enviroment 8*

7 - Estudis i decisions

7.1 Llenguatge de programació

Als inicis del projecte s'havia considerat la opció de treballar en *Haskell*, però es va descartar per el caire imperatiu que suposa una aplicació interactiva i per la limitada compatibilitat multiplataforma de les interfícies gràfiques del llenguatge.

La següent opció considerada va ser *Java* per els dos motius anteriorment mencionats, però finalment es va triar Scala gracies a la proposta del tutor d'usar aquest llenguatge, en el que ja havia treballat, i que era un punt intermedi de les anteriors opcions considerades.

Crec que va ser una decisió encertada usar aquest llenguatge per la gran compatibilitat multiplataforma, la intercompatibilitat amb *Java*, la gran quantitat de llibreries disponibles i la disciplina multiparadigma.

7.2 Entorn de desenvolupament

Idea Intellij és un *IDE* (*Integrated Development Enviroment)* inicialment creat per desenvolupar software en *Java.* Disposa d'una versió gratuïta (*community*) i una de pagament (*ultimate*).

Tal com ve aquest *IDE* només permet desenvolupar en Java, Kotlin i Groovy, però gràcies al magnífic suport de *plugins* es poden usar desenes de llenguatges més, inclòs *Scala,* amb el *plugin* mantingut per la comunitat oficial d'aquest llenguatge.

També es va tenir en compte que l'*IDE* disposava d'un dissenyador d'interfícies gràfiques i disposava d'un *plugin* que donava suport per JavaFX.

He triat aquest entorn de desenvolupament perquè és segurament el millor *IDE* que hi ha per treballar en *Scala.* Al haver-lo usat en diverses assignatures i per projectes extraescolars ja estic familiaritzat amb l'entorn i tinc per mà les diferents eines que ofereix com ara el depurador.

El depurador de per si ja és bo, però quan a més treballes en *Scala* es converteix en una de les eines per depurar més potents que he usat. El fet de que *Scala* suporti *lambdes* en les expressions permet usar-les en l'apartat de *watches* en temps d'execució, quan s'està encallat en un breakpoint. Això junt amb la flexibilitat de les col·leccions del llenguatge i les funcions d'ordre superior permeten computar expressions útils per depurar errors difícils.

Tot i disposar de llicència gratuïta com a estudiant i ja haver usat la versió *ultimate* anteriorment, aquest projecte s'ha fet amb la versió *community*.

7.3 API per la interfície gràfica

La elecció de la API de l'interfície gràfica no va ser senzilla. La primera decisió que va aparèixer va ser el Swing vs. JavaFX. Els pros de JavaFX eren API amb millor disseny (*Model-Vista-Controlador*) i més moderna, més opcions per crear camps de text personalitzables i un dissenyador gràfic extern i concís.

Mentre que els pros de Swing eren no requerir dependències externes, llibreria madura i testejada, gran comunitat, informació i dubtes resolts, familiaritat amb la llibreria, dissenyador d'interfícies integrat en l'*IDE* (a Intellij IDEA i Netbeans).

Es va fer una prova amb les dues llibreries i Swing va ser la llibreria que va semblar més adequada pel projecte per tenir els *widgets* necessaris per tal de crear els camps de text personalitzable que es necessitaven. La experiència prèvia adquirida durant l'estada a l'empresa, on es va fer ús d'aquesta llibreria també van influir en la decisió.

Un cop triada la llibreria s'havia de decidir si usar la versió d'*Scala* o *Java*.

Scala Swing és una llibreria creada pels desenvolupadors d'*Scala* que embolcalla Java Swing. Per instal·lar-la es va passar el projecte a SBT i es va afegir la dependència necessària. En aquest moment va aparèixer el primer problema i és que per usar la llibreria s'havia d'usar una versió d'*Scala* antiga, i el pitjor era que no semblava que això s'anés a arreglar degut a que la llibreria està parcialment obsoleta i poc mantinguda.

Apart de no incloure totes les funcionalitats de *Java Swing*, *Scala Swing* tampoc era compatible amb el dissenyador gràfic que incorporava l'*IDE.*

Tot això junt amb la menor comunitat de la llibreria van fer la decisió d'usar *Java Swing* evident.

La part gràfica de l'aplicació s'ha fet en *Java* mentre que els solucionadors s'ha fet en *Scala.* Això no ha suposat un problema gràcies a la compatibilitat que hi ha entre els dos llenguatges

7.3.1 Dissenyadors gràfics

En un primer moment es va veure una opció molt interessant poder treballar amb un dissenyador gràfic per facilitar la programació i disseny de la GUI, ja que era una eina similar a la que es troba a *Netbeans,* amb la que ja s'havia tingut experiència prèvia.

El dissenyador gràfic s'havia usat en un projecte anterior per crear una aplicació estàtica que comptava amb varies entrades, botons i finestres de resultats.

La bona experiència obtinguda suggeria que seria una bona opció per aquest projecte però va resultar no ser així.

La interfície gràfica del projecte requeria un *layout* poc freqüent i dinàmic, que era difícil d'aconseguir amb un dissenyador gràfic pensat principalment per construir formularis.

Finalment es va optar per fer la interfície amb codi, que dificulta el disseny, augmenta el temps de desenvolupament i en limita la modularitat.

7.4 Draw.io

Draw.io és una eina en linea per fer dibuixos i diagrames varis. Disposa de molts elements d'UML que permeten crear tot tipus de diagrames dinàmics de forma ràpida i senzilla. S'ha usat per fer els diagrames d'aquest l'informe.

7.5 Tècniques incloses en els solucionadors

Es va determinar que era essencial que s'incloguessin les tècniques que s'avaluaven a l'assignatura i es va decidir incloure el *2WL* tot i no ser contingut avaluable perquè era una tècnica interessant i podia ser visualitzada fàcilment en l'interfície.

8 - Anàlisi, disseny i implementació del sistema

SAT-IT o *SAT-Interactive Tracer* és l'eina resultant d'aquest projecte que permet resoldre i traçar interactivament la resolució d'una instància *CNFSAT* donada.

A continuació es detallen les parts més interessants.

8.1 Solucionadors

Aquí es detallen els diferents solucionadors que s'han desenvolupat al llarg del projecte.

8.1.1 Solucionador inicial Backtracking

La implementació inicial d'aquest algoritme va ser basada en l'esquema de backtracking recursiu.

En aquesta implementació les clàusules inicials no es conservaven si no que s'anaven traient les clàusules resoltes i s'eliminaven els literals insatisfactibles de les no satisfetes.

Per guardar les clàusules es va crear una estructura de dades *ClauseSet,* que era un conjunt de conjunts d'enters. Es va triar aquesta estructura perquè permetia eliminar clàusules duplicades, no només en la fórmula inicial, si no al llarg de l'execució del solucionador. Aquesta estructura també garantia que no hi havien literals duplicats en una mateixa clàusula que va simplificar molt aquesta implementació inicial

Un dels molts problemes que tenia aquesta implementació era que usava la col·lecció *Set* d'*Scala*, una estructura immutable crea una copia nova cada cop que es modifica. Per resoldre instàncies grans amb aquesta estructura de dades la quantitat de recursos que es gastarien fent copies superarien els necessaris per fer la resolució en si.

En la implementació final les clàusules inicials no es modifiquen i es fa servir un vector mutable per guardar l'estat de cada variable i un altre per guardar el motiu d'assignació de les variables.

8.1.2 Solucionador inicial DPLL

El solucionador *DPLL* inicial es basava en l'anterior però afegia el procediment de la propagació unitària. Això va permetre resoldre instàncies més grans mantinguent una implementació molt senzilla.

Una curiositat d'aquesta implementació era que els literals decidits s'afegien com una clàusula unitària en el conjunt de clàusules. Això funcionava perquè en el nivell anterior la propagació unitària ja havia eliminat totes les clàusules unitàries i quan es feia la propagació unitària del següent nivell, es propagava la variable decidida, que desencadenava la resta de propagacions unitàries que poguessin aparèixer.

La implementació dels dos solucionadors inicials i la seva estructura de dades es pot trobar a l'annex d'aquest document perquè va acabar siguent descartada, com s'explica a continuació.

8.1.3 Solucionador DPLL iteratiu

Quan va arribar el moment d'implementar el solucionador *CDCL* es va veure que la implementació recursiva dificultaria molt la implementació de la tècnica del *backjump*.

Per això, un cop discutit amb els tutors es va prendre la decisió d'implementar el solucionador en versió iterativa i descartar els solucionadors recursius.

Abans de fer el solucionador *CDCL* es va implementar el solucionador *DPLL* iteratiu, partint des de zero.

Els requisits que tenia aquest solucionador eren els següents:

- Saber en quin ordre s'han assignat les variables
- Saber, per cada variable, en quin estat es troba (no assignat o indefinit, cert, fals)
- Per cada variable, saber el motiu d'assignació (decisió, propagació unitària, backtrack)

Aquestes consultes es faran milers o milions de vegades en cada resolució així que han de ser el més rapides possibles, és a dir, que si volem que el solucionador sigui àgil s'han de fer en temps constant.

Les estructures de dades que s'han usat per satisfer els requisits han sigut les següents:

- Un vector dinàmic on es guardaran les assignacions de variables que anem fent al llarg de la resolució, anomenada **trail**
- Un vector de longitud *numVariables,* de tipus *State,* anomenada **varValue**
- Un vector, de longitud *numVariables*, de tipus *Reason*, anomenada **assignmentReason**

En les estructures de dades, un literal es representa com un enter amb signe.

State és una enumeració que té com a possibles valors *UNDEF, TRUE* i *FALSE.*

Reason és una enumeració que té com a possibles valors *DECISION, UNITPROP i BACKTRACK.*

```
override protected def i_solve(instance: Instance): Boolean = {
  val UPresult = init(instance)

  if(UPresult==SolvingState.SAT)
    return true
  else if(UPresult==SolvingState.UNSAT)
    return false

  while(trail.length<numVars){
    val conflClause = unitPropagation()

    if(conflClause != -1) { //si hi ha hagut conflicte...
      val decisionLevel = calculateBacktrackLevel()
      if (decisionLevel < 0)
        return false
      else
        backtrack(decisionLevel)
    }
    else if (trail.length<numVars){ //fer decisió
      val lit = makeDecision
      assign(lit, Reason.DECISION) //assignar i processar lit
    }
  }

  true
}
```

Mètode i_solve de la classe DPLL basat en [TJC00]

8.1.4 Solucionador CDCL

El *CDCL* es basa en el *DPLL* iteratiu, de fet la funció i_solve, que conté el bucle principal, només es diferencia per dues crides de funcions: *calculateBacktrackLevel*() passa a ser *conflictAnalysisAndLearning(confClause)* i *backtrack(decisionLevel)* passa a ser *backjump(decisionLevel).*

Aquest algoritme afegeix nous requisits:

- Saber, per cada variable propagada, quina clàusula l'ha propagat
- Poder afegir noves clàusules durant l'execució

Que s'han satisfet amb les següents estructures:

- Un vector, de longitud *numVariables*, que indiqui, en cas de que hagi estat propagada, quina clàusula l'ha propagat, anomenada **whoPropagated**
- El *ClauseWrapper*, una estructura que embolcalla les clàusules inicials i les clàusules apreses. Es diu senzillament **clauses**

Per indicar quina clàusula ha propagat un literal es pot fer guardant una referència a aquella clàusula o guardant l'index de la clàusula. Es va optar per la segona opció perquè va semblar més pràctic per depurar.

8.1.4.1 Clause Learning

El *CDCL* tal com està explicat a l'apartat 5, es caracteritza per incloure les tècniques de *clause learning* i *backjumping*.

```
private def conflictAnalysisAndLearning(failClauseIdx: Int): Int = {

  //si al trail només queden propagacions pleguem perquè no podem fer backjump enlloc
  if(trail.forall(b => assignmentReason(math.abs(b)) == Reason.UNITPROP))
    return -1

  val failClause = clauses.getClause(failClauseIdx)

  val lastDecisionLevelLits = calcLastDecisionLevelLits
  val lastDecisionLevelLitsSet = lastDecisionLevelLits.map(-_).toSet

  var resolvent = failClause.getClause.toSet
  var lastLevelLit = getLastLevelLit(resolvent, lastDecisionLevelLits)

  while(resolvent.intersect(lastDecisionLevelLitsSet).size>1){ //mentre resolvent tingui
mes d'un literal de lultim nivell de decisio...
    val indexClauseToSolveWith = whoPropagated(math.abs(lastLevelLit))
    var clauseToSolveWith = clauses.getClause(indexClauseToSolveWith).getClause.toSet

    //aplicar resolucio
    clauseToSolveWith -= lastLevelLit
    resolvent -= -lastLevelLit
    resolvent = resolvent.union(clauseToSolveWith)

    lastLevelLit = getLastLevelLit(resolvent, lastDecisionLevelLits)
  }

  val newClauseAB = resolvent.to[ArrayBuffer]
  val newClause = new Clause(newClauseAB)

  //afegir nova clausula
  val newClauseIndex = clauses.addClause(newClause)

  calculateBackjumpLevel(resolvent, newClause, newClauseIndex)
}
```

Versió simplificada de la funció conflictAnalysisAndLearning de CDCL.scala

Aquesta funció duu a terme l'*anàlisi del conflicte* (tal com està explicat a l'apartat 5), l'aprenentatge i retorna la posició del *trail* a la que s'ha de fer el *backjump*.

8.1.4.2 Backjumping

Un cop fet l'anàlisi del conflicte i l'aprenentatge hem de fer *backjump* al nivell de decisió calculat. La següent funció ens calcula directament la posició del *trail* on hem de fer *backjump*.

```
private def calculateBackjumpLevel(learntClause: Set[Int], newClause: Clause,
  newClauseIndex: Int): Int ={

  var trailIdx = 0

  //mentre la nova clausula tingui mes dun literal sense valor...
  while (learntClause.size > 1) { //si es TLA no entrarem aqui
    val lit = trail(trailIdx)
    if (learntClause.contains(-lit)) {
      learntClause -= -lit
    }
    trailIdx += 1
  }

  //apuntem al primer literal del proxim nivell de decisio
  while(trailIdx<trail.length && assignmentReason(math.abs(trail(trailIdx))) ==
Reason.UNITPROP)
    trailIdx += 1

  val litToForceProp = learntClause.head

  backjumpDelay = new BackjumpDelay(litToForceProp,newClauseIndex)

  trailIdx
}
```

Versió simplificada de la funció calculateBackjumpLevel de CDCL.scala

Per determinar on hem de saltar hem de trobar el nivell de decisió que ens forci la propagació unitària de la clàusula apresa.

L'algoritme primer recorre el *trail* des de l'inici i va eliminant els literals falsificats de la clàusula apresa, és a dir, els que apareixen al *trail* en signe contrari, fins que només queda un literal a la clàusula (la clàusula apresa està falsificada amb la configuració del *trail* en el moment del conflicte). El literal que queda a la clàusula al final d'aquest procés és el literal del qual forçarem la propagació.

Per no haver de tornar a fer les propagacions del nivell de decisió al qual hem retrocedit, la nova propagació unitària es farà just abans del primer literal del pròxim nivell de decisió.

8.1.4.3 Two-watched-literals

Per implementar-la es necessita el següent:

- Una manera per marcar 2 literals de cada clàusula
- Poder saber, per cada literal, a quines clàusules apareix observat
- Una estructura LIFO on es guardaran els literals pendents de propagar

Per satisfer el primer requisit hi ha dos aproximacions igual de vàlides. La primera opció és guardar els dos literals explícitament. És a dir que necessitaríem un vector, de longitud *numClausules,* de tuples de dos enters.

La segona manera és guardar els dos literals implícitament en les dues primeres posicions de les clàusules. Aquesta aproximació requereix menys memòria i ens estalvia una estructura de dades extra a mantenir. Es va decidir fer servir aquesta última.

Per complir el segon requisit necessitarem les *ocurrence lists*, o sigui, dues llistes de llistes. En necessitem dues perquè cada variable pot aparèixer en positiu i en negatiu. Necessitem llistes de llistes perquè cada literal pot aparèixer observat a més d'un

a clàusula i necessitem saber a totes les que apareix observat. Els hi direm **watchListPos** a la que conté els literals positius i **watchListNeg** per els negatius.

Per fer que la nostre aproximació sigui eficient hem de poder modificar les clàusules per intercanviar les posicions dels literals quan passin o deixin de ser observat*.* Per fer això sense haver de crear una nova clàusula cada vegada ens hem d'assegurar d'usar una estructura de dades mutable*.*

El tercer requisit s'ha implementat amb una pila mutable de tipus *ArrayStack* anomenada ***toPropagate****.* En aquesta estructura s'hi empilaran els literals cada cop que es faci una assignació per posteriorment executar l'algoritme de propagació unitària i mantenir les condicions dels *2WL.*

Tot i que no és necessari pel funcionament del *DPLL,* es va trobar oportú guardar quina clàusula ha propagat cada literal, tal com fa el *CDCL,* per tal de poder-lo veure en la interfície gràfica, per això es va posar el camp a la classe abstracte *TwoWatchedLiteralSolver.*

Per simplificar la implementació dels *2WL* es va decidir fer un preprocés en que totes les clàusules unitàries de la fórmula s'eliminen (s'aplica propagació unitària). Cal notar que això pot comportar detectar la insatisfactibilitat de la fórmula en temps de preprocés.

Això es fa en el mètode *initialUnitProp(),* que es crida al inicialitzar qualsevol instància en un solucionador *DPLL* o *CDCL.* Aquest mètode fa les propagacions unitàries necessàries al problema original perquè al carregar-lo ja no hi hagin clàusules unitàries. Si el problema és trivial es pot arribar a resoldre sense fer una sola decisió.

Encara hi ha una altre manera en la que podem tenir clàusules unitàries i és si les aprenem durant l'execució del *CDCL.* Afortunadament no ens cal tractar aquest cas perquè ja es tracta sol. Quan trobem un conflicte i aprenem una clàusula unitària hi ha un breu moment en que tenim una clàusula unitària no satisfeta. A continuació es farà el *backjump* al nivell de decisió 0 i es farà la propagació unitària de la clàusula unitària, que farà que la clàusula passi a estar satisfeta sense cap decisió prèvia, i per tant es pot eliminar.

8.1.4.4 Propagació unitària amb 2WL

La tècnica dels *2WL* ens estalvia haver de recórrer totes les clàusules per buscar literals unitaris o un conflicte. A continuació veurem el funcionament de l'algoritme.

```
//Pre: Les clausules tenen almenys dos literals o be en tenen un de satisfet
//Post: Fa les propagacions unitaries necessaries retorna la clausula que ha generat el
conflicte, -1 en cas de no conflicte
protected def unitPropagation(): Int ={

  while(toPropagate.nonEmpty){

    val lit = toPropagate.pop()
    val falseLit = -lit
    val atom = math.abs(lit)

    //Buscar clausules on apareix -lit, s'hauran de comprovar
    val watchListFalse = if(lit<0) watchListPos(atom) else watchListNeg(atom)

    //iterem sobre una copia perque el real pot ser que es modifiqui
    val watchListFalseCopy = watchListFalse.toArray
    for(clauseIndex <- watchListFalseCopy){
      val clause = clauses.getClause(clauseIndex).getClause

      if (!isSatisfied(clauseIndex)) { //70% del temps del programa aqui

        //apartem el literal fals cap a la segona posicio
        if (clause(0) == falseLit) {
          clause(0) = clause(1)
          clause(1) = falseLit
        }

        var j = 2
        var undefinedFound = false
        val clLength = clause.length
        while (j < clLength && !undefinedFound) {
          if (varValue(math.abs(clause(j))) == State.UNDEF) { //busquem un literal no
assignat
            swapWL(clauses.getClause(clauseIndex),clauseIndex,1,j)
            undefinedFound = true
          }
          j += 1
        }

        if (!undefinedFound) { //la clausula no conte cap altre literal undefined
          val firstLit = clause(0)
          if(varValue(math.abs(firstLit))==State.UNDEF){ //fer UP
            assign(firstLit,Reason.UNITPROP,clauseIndex) //afegeix al toPropagate
          }
          else{ //contradiccio
            toPropagate.clear()
            return clauseIndex
          }
        }
      }
    }
  }
  -1
}
```

L'algoritme comença amb un bucle mentre que seguirà fins que haguem buidat la pila *toPropagate* o bé fins que trobem un conflicte.

En cada iteració del bucle processarem el literal del cim de la pila, se'n farà referència com el literal "actual". Amb l'ajut de les *occurrence lists* recorrerem totes les clàusules que tenen observat el negat del literal actual perquè són les candidates a crear clàusules unitàries o conflictes.

Comprovem si la clàusula està satisfeta, i en, cas afirmatiu, la ignorem, ja que una clàusula satisfeta no pot crear propagacions ni conflictes. En cas contrari seguim.

Apartem el literal fals cap a la segona posició en cas de que no hi sigui i comencem a buscar un literal no definit d'entre els no observats (a partir de la 3a posició). Si el trobem re establim la invariant dels *2WL* i anem a per la següent clàusula.

En cas de no trobar un literal no definit en la resta de la clàusula farem el següent: si el primer literal no està definit hem de fer propagació unitària, altrament hem trobat un conflicte per tant netejarem de la pila *els* li*terals que hi pugui haver pendents de propag*ar i retornarem l'index de la clàusula que l'ha provocat.

8.1.5 Solucionador Backtracking final

La re-implementació del Backtracking es va fer partint del *DPLL*.

Es va deixar d'estendre la classe del *2WL*, desactivar la propagació unitària inicial i re-implementar el bucle principal.

8.2 Estructures dels solucionadors

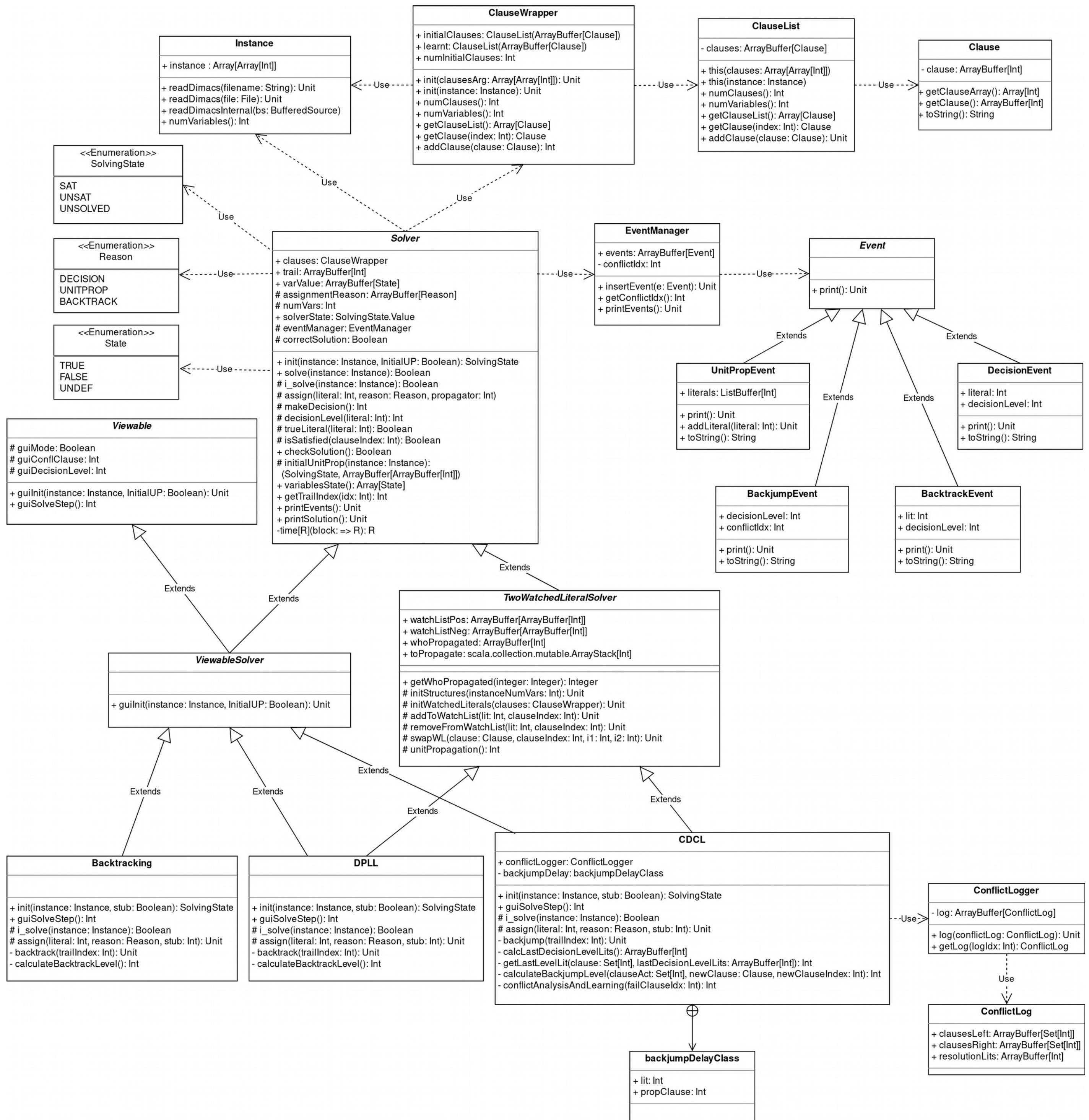


Diagrama de classes dels paquets solver i structure

8.2.1 Solver

Solver és una classe abstracte que representa un solucionador genèric, el seu mètode més important és solve(), que apart d'iniciar la resolució s'encarrega de cronometrar-la, guardar-ne la solució i comprovar que aquesta sigui correcte.

Implementa el mètode makeDecision() que tots els solvers hereden i no sobreescriuen, pel que es pot canviar l'heurístic de tots els solucionadors només modificant aquest mètode.

8.2.2 BackjumpDelay

BackjumpDelay és una classe privada del *CDCL* que serveix com a contenidor per guardar les dades del *backjump* des del moment en que es calcula el literal que es propagarà fins que es propaga.

8.2.3 Event

Event és una classe abstracte que representa una acció del solver. Pot ser una decisió, una propagació unitària, un *backjump* o un *backtrack*. Un event de propagació unitària pot contenir múltiples literals propagats en el mateix nivell de decisió

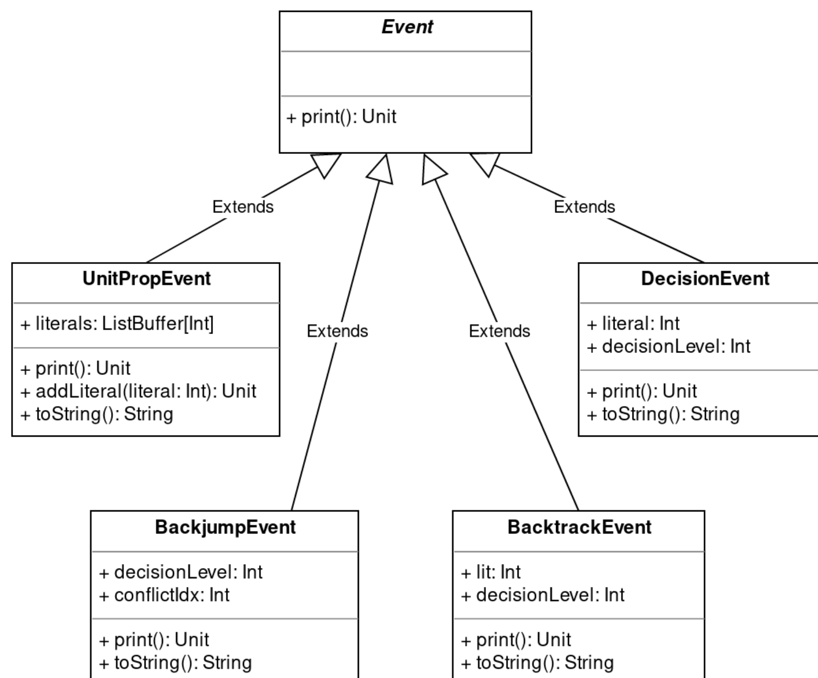


Diagrama de la classe Event i les seves subclasses

8.2.4 Instance

Instance
+ instance : Array[Array[Int]]
+ readDimacs(filename: String): Unit + readDimacs(file: File): Unit + readDimacsInternal(bs: BufferedSource) + numVariables(): Int

Aquesta classe representa una instància del problema *CNFSAT*. Per defecte s'inicialitza amb una instància buida sense clàusules ni variables. Permet carregar una instància donat un nom de fitxer o donat un descriptor de fitxer. Un cop carregada la instància la podem usar per inicialitzar un solucionador amb el problema carregat.

L'aplicació accepta fitxers amb el format *DIMACS CNF,* molt estandaritzat en el camp.

8.3 Problemes i solucions durant la implementació

Als primers tests de les implementacions sovint s'obtenien solucions incorrectes o el programa petava per algun lloc però cap d'aquests errors van ser difícils d'arreglar.

L'error més gran que hi va haver durant el projecte va ser durant la implementació del *CDCL.* El solver semblava que funcionava però al intentar resoldre alguna instància mitjanament gran es quedava encallat durant la fase d'aprenentatge, concretament en l'anàlisi de conflicte.

Un cop analitzat amb el depurador es va comprovar que per algun motiu la clàusula amb la que es feia resolució no contenia el literal contrari, pel que no es resolia res i es quedava en en bucle intentant fer resolució amb la mateixa clàusula.

Es va fer la hipòtesis que l'error seria a prop, en la implementació de la resolució. Es van detectar petits errors en el calcul dels literals de l'últim nivell de decisió i es van solucionar però el problema no es va resoldre. Tocant el mètode *getLastLevelLit*, que retorna un literal de l'ultim nivell de decisió feia que el solució petés en comptes de quedar-se encallat, que va semblar un pas endavant, però va resultar no ser l'origen de l'error.

El problema al depurar aquest error era que només apareixia en instàncies grans, que són molt difícils de depurar i no semblava que el problema passés en els passos previs a quedar-se

encallat. Es va crear un script per generar instàncies petites amb clàusules aleatòries usant un generador de fórmules [TFG10], però no es va aconseguir reproduir el problema.

Es va revisar el codi buscant l'error i es van afegir moltes assercions, per comprovar que en les instàncies grans tot estigués bé i no sortissin index de lloc ni es trenquessin assumpcions fetes en la implementació. Això va permetre detectar alguns errors més però el problema persistia.

Es va recórrer a la literatura [TFG04] per comprovar que no fos un problema de concepte. Allà es van trobar algunes invariants estrictes que semblaven necessàries pel correcte funcionament del 2WL. Es va veure que no es complien posant diverses comprovacions.

Es va intentar entendre perquè no s'estaven complint les condicions perquè es creia que podien ser l'origen del problema i es van provar diverses solucions que no van donar resultat. Un cop parlat amb els tutors es va arribar a la conclusió que les condicions eren massa restrictives i no calia que es complissin.

Finalment es va trobar l'error. Aquest era un *off-by-one* (un index que està una posició més enllà d'on ha d'estar) en la implementació del *backjump*. Quan la primera propagació unitària després d'una decisió creava un conflicte, es feia *backjump* a un nivell de decisió superior del que tocava.

A primera vista, no sembla un problema important, ja que saltar més enrere no ens faria perdre solucions. El problema estava en que després de fer *backjump*, en el CDCL es 'força' la propagació unitària de la clàusula apresa. Quan el nivell de *backjump* era incorrecte, aquesta propagació unitària no era conseqüència de la clàusula apresa i impedia aplicar resolució si apareixia durant un anàlisi de conflicte.

L'error es va arreglar eliminant la línia de codi que decrementava l'index.

Un error important però fàcil d'arreglar es va manifestar amb literals repetits a les solucions del problema. L'error era degut a que el mètode *initialUnitProp()* intentava fer totes les propagacions unitàries d'una sola vegada i voler fer masses coses alhora va acabar resultant en un error. Es va arreglar fent les propagacions una per una.

9 - Proves

Per fer proves es va fer una interfície de terminal per generar instàncies del *problema de les n reines* amb el codi resultant de la pràctica de l'assignatura *Programació declarativa. Aplicacions*. Les instàncies amb n senar semblen tardar molt menys que les altres.

Es va usar un generador d'instàncies anomenat *cnfgen* per generar instàncies del *pigeonhole problem (php)* i *random k-sat*.

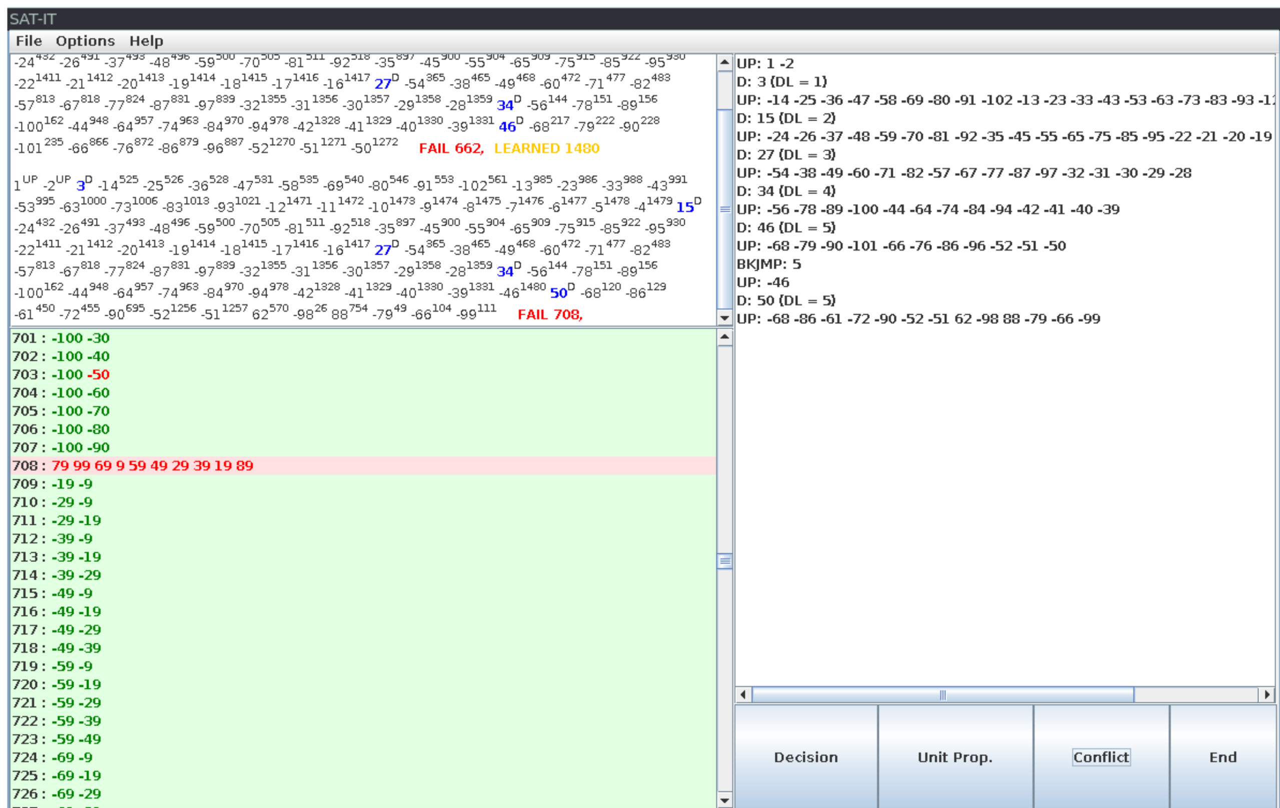
També es van usar instàncies d'altres problemes: *hanoi4, cc3-2-1*, *ssa0432-003* i es van crear manualment algunes instàncies petites per testejar funcions com la propagació inicial.

10 - Resultats

Aquesta aplicació s'ha ensenyat als professors de l'assignatura *Programació declarativa. Aplicacions* que pel que sembla estan disposats a fer-la servir aquest mateix quadrimestre en l'assignatura.

L'aplicació no requereix cap procediment d'instal · lació més enllà de disposar de l'executable.

10.1 Interfície de l'aplicació

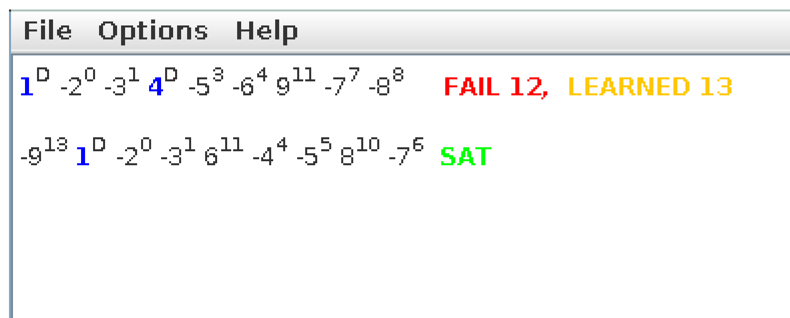


Vista general de l'interfície

En aquest apartat es descriuran les diferents parts de la interfície, el seu propòsit i el significat dels elements que hi apareixen.

10.1.1 Visualitzador del trail

La finestra superior esquerra correspon al visualitzador del *trail*. El *trail* que havíem descrit anteriorment guardava l'assignació actual de variables en ordre. En aquesta finestra en canvi hi apareix tot l'històric d'assignacions de variables que s'han fet al llarg de la traçada de la instància. Cada vegada que es fa un *backtrack* o *backjump*, que és quan s'han de desfer assignacions, s'inicia una nova linia. El conjunt d'assignacions més recent apareix sempre a la part inferior.



Detall del visualitzador del trail

Els literals es representen amb números enters. Un enter positiu representa un literal positiu i al revès. En el superíndex de cada literal hi poden aparèixer diferents símbols en funció del solucionador seleccionat.

- Una *D* significa que el literal s'ha afegit degut a una **decisió**
- Un *BK* significa que el literal s'ha afegit degut a un ***backtrack***
- Un *UP* significa que el literal s'ha afegit degut a una **propagació inicial**
- Un número *n* significa que s'ha afegit degut a la **propagació unitària** de la clàusula *n*

Els literals decidits apareixen de color blau per veure ràpidament els nivells de decisió. El símbol *BK* només pot aparèixer si usem el solucionador de *backtracking* o el DPLL. El símbol UP només pot aparèixer quan usem els solucionadors DPLL o CDCL i sempre a l'inici del trail. Això és degut a que les propagacions inicials no requereixen cap decisió i per tant no es desfaran mai.

Quan es troba un conflicte apareix el missatge, de color vermell, FAIL *c*, on *c* és la clàusula que ha provocat el conflicte. Això significa que no podem continuar amb aquella assignació de variables i que es procedirà a fer *backtrack* o *backjump*, en funció del solucionador.

Després d'un FAIL poden aparèixer 3 missatges:

- UNSAT, que vol dir que la instància és insatisfactible i per tant hem acabat
- BACKTRACK, que vol dir que hem desfet l'ultima decisió feta
- LEARNED *c*, que vol dir que hem fet anàlisi del conflicte i hem après una nova clàusula amb index *c*.

Si aconseguim assignar totes les variables sense crear un conflicte apareixerà SAT, i l'assignació que tenim al *trail* serà un model de la fórmula.

10.1.2 Visualitzador de clàusules

La finestra inferior esquerra és el visualitzador de clàusules. En aquesta finestra hi podem veure les clàusules inicials i les clàusules apreses junt amb el seu estat i l'estat dels seus literals.

```
3269 : -694 -711 652
3270 : -694 -712 653
3271 : -694 -713 654
3272 : -701 -714
3273 : -701 -708
3274 : -701 -709
3275 : -701 -711 659
3276 : -701 -712 660
3277 : -701 -713 661
3278 : -673 669
3279 : -686 670
3280 : -694 671
3281 : -701 672
3282 : -687 669
3283 : -674 670
3284 : -676 671
3285 : -678 672
3286 : 687 715 -669
3287 : -670 716 674
3288 : -671 717 676
```

Detall del visualitzador de clàusules

Cada fila de la llista representa una clàusula. El primer número que veiem és l'índex de la clàusula, que comença per 0 i va fins al número de clàusules menys 1.
Si l'índex de la clàusula és negre vol dir que és una clàusula inicial del problema, si és blava vol dir que és una clàusula apresa.

A partir dels dos punts comença la clàusula, que és una llista de literals separats per un espai. Els literals poden aparèixer de 3 colors segons el seu estat:

- Color verd significa literal satisfet
- Color vermell significa literal falsificat
- Color negre significa literal no definit

Si un sol literal de la clàusula està satisfet, la clàusula també ho està i es veurà pel color verd de fons que té la clàusula. Si tots els literals de la clàusula estan falsificats la clàusula serà conflictiva i apareixerà amb fons de color vermell. Finalment si no passa cap de les dos anteriors la clàusula apareixerà amb el fons blanc.

Implícit en les clàusules hi tenim els literals observats que són sempre els dos primers. Es pot veure la seva evolució quan s’executa el solucionador pas a pas.

10.1.3 El visualitzador d'events

La finestra del visualitzador d'events es troba a la part dreta.

```
UP: 1 -2
D: 3 (DL = 1)
UP: -14 -25 -36 -47 -58 -69 -80 -91 -102 -13 -23 -33 -4
D: 15 (DL = 2)
UP: -24 -26 -37 -48 -59 -70 -81 -92 -35 -45 -55 -65 -75
D: 27 (DL = 3)
UP: -54 -38 -49 -60 -71 -82 -57 -67 -77 -87 -97 -32 -31
D: 34 (DL = 4)
UP: -56 -78 -89 -100 -44 -64 -74 -84 -94 -42 -41 -40 -3
D: 46 (DL = 5)
UP: -68 -79 -90 -101 -66 -76 -86 -96 -52 -51 -50
BKJMP: 5
UP: -46
D: 50 (DL = 5)
UP: -68 -86 -61 -72 -90 -52 -51 62 -98 88 -79 -66 -99
BKJMP: 5
UP: -88
D: 50 (DL = 5)
UP: -68 -86 -61 -72 -90 -52 -51 62 -98
BKJMP: 5
UP: -50 90 -72 -99 -68 -79 -101 -86
BKJMP: 4
UP: -34
D: 39 (DL = 4)
UP: -66 -84 -50 -61 -72 -79 -89 -99 -42 -41 -40
D: 44 (DL = 5)
UP: -88 -64 -74 -94 -52 -51 -46 101 -68 -90 -100 -98 -9
BKJMP: 5
```

Detall del visualitzador d'events

Aqui es pot veure, per ordre cronològic, cadascún dels events que el solucionador ha processat. Un event pot ser una decisió, un conjunt de propagacions unitàries, un *backtrack* o un *backjump* i es poden diferenciar per l'identificador a l'inici de cada event que és *D, UP, BKJMP* o *BT* respectivament.

Després de l'identificador de tipus d'event i del doble punt hi apareixen dades que tenen diferent significat segons el tipus d'event:

- Les decisions mostren el literal decidit seguit del *nivell de decisió* al que estan
- Les propagacions unitàries mostren tots els literals propagats, separats per espai

- El *backtrack* i *backjump* mostren el nivell de decisió on s'ha fet el salt

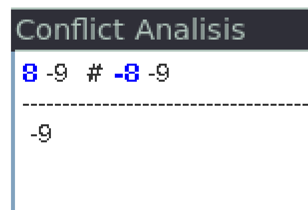
Després de cada *backjump* sempre hi ha almenys una propagació unitària, resultat de la propagació de la clàusula apresa.

Si fem doble clic sobre un event de *backjump* s'obrirà una finestra on podrem veure la seva anàlisi de conflicte.

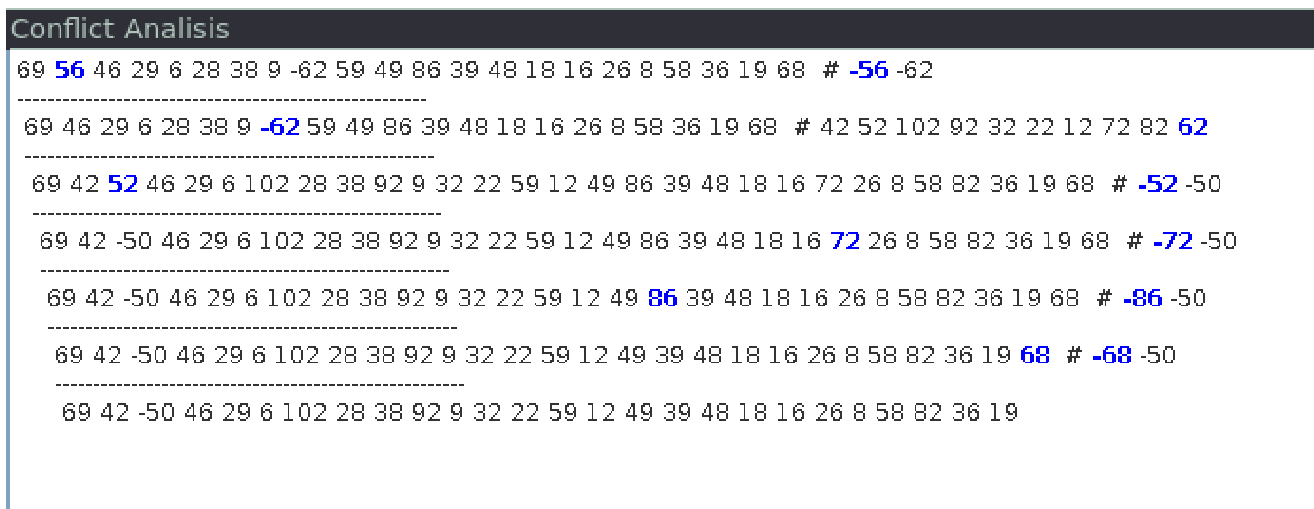
10.1.4 Finestra d'anàlisi de conflicte

Aquesta finestra mostra l'anàlisi de conflicte que ja ha estat explicat prèviament. A cada pas de resolució hi intervenen dos clàusules, separades per el símbol de coixinet #.

A la part esquerra i dreta del coixinet apareixen els resolvents. La primera clàusula que apareix és sempre la que ha causat el conflicte i l'última és la clàusula apresa.



Exemple d'un anàlisi de conflicte petit

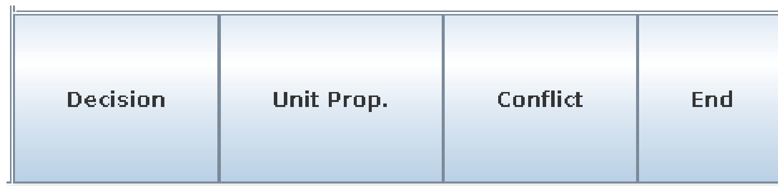


Exemple d'un anàlisi de conflicte més gran

A cada nivell de resolució hi ha marcat, en color blau, el literal amb qui es fa resolució.

10.1.5 Botons interactius

Els botons interactius, situats a la part inferior dreta, serveixen per interactuar amb el solucionador.



Detall dels botons interactius

El solucionador distingeix 6 tipus d'esdeveniments: decisió, prop. unit., conflicte, *backjump*, *backtrack* i fi (quan acaba l'algorisme de solució).

Tots els botons arranquen el solucionador i s'aturen quan arriben a certs esdeveniments:

- El botó de decisió en tots els esdeveniments excepte quan troba una propagació unitària
- El boto de propagació unitària es para en tots els esdeveniments excepte quan troba una decisió
- El botó de conflicte es para en tots els esdeveniments excepte decisió o propagació unitària
- El botó de fi només para quan troba un fi

10.2 Interfície de terminal

Inicialment creada per testejar i depurar aquesta interfície es va deixar com a opció per fer proves de rendiment i usar com a solucionador autònom.

Permet resoldre instàncies del problema *CNFSAT* amb els 3 solucionadors disponibles. Té la opció de veure tots els e*vents* que s'han generat al llarg de la resolució.

Aquesta és la interfície que ofereix la versió de terminal:

```
% java -jar SAT-IT_term.jar
Usage: java -jar SAT-IT-terminal.jar <input-file> [options]

  where input-file is a DIMACS CNF file.

OPTIONS:

  -bt        Backtracking solver
  -dpll      DPLL solver
  -cdcl      CDCL solver (default)

  -events    Show all the solver steps when finished
  -h, --help  Show help message
```

Exemples d'ús:

```
% java -jar SAT-IT_term.jar ../../../cnf/easy/php10-5.cnf
Solved in: 162ms
UNSAT

% java -jar SAT-IT_term.jar ../../../cnf/easy/random_ksat.cnf -bt
Solved in: 15ms
SAT
1 2 3 4 5 6

% java -jar SAT-IT_term.jar ../../../cnf/easy/NQueens5quad.cnf  -dpll -events
Solved in: 83ms
SAT

Events:
UP: 1 -2
D: 3 (DL = 1)
UP: -9 -15 -21 -27 -8 -13 -18 -23 -7 -6 -5 -4
D: 10 (DL = 2)
UP: -14 -16 -22 -20 -25 -12 -11 17 26 -24 19
1 -2 3 -4 -5 -6 -7 -8 -9 10 -11 -12 -13 -14 -15 -16 17 -18 19 -20 -21 -22 -23 -24 -25
26 -27
```

10.3 Normativa i legislació

Aquest projecte no presenta cap problema de cara a la normativa vigent.

Com que l'aplicació no guarda cap tipus de dada personal no l'afecten ni la *LOPDE* ni la *GPDR*. Tampoc li és aplicable la LSSICE ja que el projecte no involucra una activitat econòmica.

11 - Conclusions

Els objectius del projecte eren, a grans trets, crear una eina interactiva per ús educatiu que permetés traçar les passes dels diversos solucionadors estudiats a classe i poder-ne visualitzar i estudiar les tècniques. Els objectius s'han complert creant una aplicació visual, interactiva i responsiva que facilita l'estudi dels solucionadors SAT.

Crec que l'eina serà útil en l'entorn acadèmic perquè amplia els recursos disponibles per l'estudiant i ho fa com cap recurs tradicional ho pot fer. L'eina permet a l'estudiant treballar al seu ritme i practicar amb la tranquil·litat de saber que no està cometent errors. També pot incentivar a fer més exercicis ja que pot utilitzar l'eina per validar més instàncies que les donades pel professor. En cas d'error, poder avançar per la resolució pas a pas permet trobar ràpidament l'origen de l'error i corregir-lo.
Intentar crear instàncies difícils i veure com es resolen pot ser un exercici interessant per aprofundir en el camp.

Si hagués tingut una eina així a disposició quan vaig fer l'assignatura segur que l'hagués fet servir. Les eines de suport de les que s'ha disposat al llarg de la carrera com ara el Matlab, el Logicweb, l'ACME o el TILC quasi sempre m'han sigut útils.

Crec que ha quedat una interfície neta, simple, visual i usable que serà podrà ser útil per la docència. Com a aspectes negatius destacaria el disseny millorable de la finestra d'anàlisis de conflicte i les possibles característiques interessants que ara no es suporten, com les que s'indiquen a l'apartat treball futur. Tot i això crec que el software s'ha dissenyat de manera que sigui fàcilment ampliable.

Amb aquest projecte he pogut aprofundir en el món de SAT, he pogut fer un tast de la dificultat d'idear i implementar estructures per millorar els algoritmes, he vist i entès tècniques i estratègies de l'estat de l'art i cap on es dirigeixen i he vist la quantitat de feina que comporta avançar en un camp tant exigent i desafiant com aquest.

12 - Treball futur

L'objectiu d'aquest projecte no ha sigut mai fer un solucionador ràpid tot i que si que s'han intentat triar unes bones estructures de dades per, si algun dia es vol estendre l'aplicació, no haver-la de refactoritzar completament.

Per millorar la velocitat del solucionador *CDCL* els 3 canvis que s'haurien de fer primer, per ordre d'importància serien:

1. Implementar un heurístic de decisió
2. Buscar una estratègia per evitar comprovar si la clàusula està satisfeta abans de fer la propagació unitària
3. Afegir estructures de dades per reduir la complexitat d'algunes operacions

Un cop fetes aquestes modificacions es podrien implementar altres tècniques de l'estat de l'art com ara un heurístic de decisió molt potent anomenat *VSIDS (Variable State Independent Decaying Sum),* els restarts, i l'esborrat de clàusules o *forgetting.*

Algunes idees que han quedat pendents són implementar un visualitzador de graf d'implicacions dels conflictes, poder continuar obtenint solucions un cop s'ha trobat la primera, tornar enrere el moment d'una decisió anterior i poder triar les variables de decisió manualment durant l'execució.

En temes de recerca podria ser interessant visualitzar els resultats de fer servir diferents estratègies com el 2-UIP, la elecció de la clàusula amb la que fer resolució quan múltiples clàusules causen un conflicte o la influència de qualsevol d'aquestes estratègies en el VSIDS.

13 - Bibliografia

Bibliografia

TFG02: Carsten Sinz and Edda-Maria Dieringer, DPVIS– A Tool to Visualize the Structure ofSAT Instances, 2005
TFG06: Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik, Chaff: Engineering an Efficient SAT Solver, 2001
TFG05: Sol Swords, Basics of SAT Solving Algorithms, 2008
TJC00: Coll, Jordi, Scheduling Through Logic-Based Tools, 2019
TFG10: Massimo Lauria, CNFgen Combinatorial benchmarks for SAT solvers,
TFG04: Mathias Fleury, Jasmin Christian Blanchette, Peter Lammich, A Verified SAT Solver with Watched LiteralsUsing Imperative HOL,
TFG00: N. Een and N. Sörensson., An extensible SAT-solver., 2004
TFG03: Allen Van Gelder, Generalizations of Watched Literals for Backtracking Search, 2001
TFG07: Albert Oliveras and Enric Rodr´ıguez-Carbonell, From DPLL to CDCL SAT solvers, 2009
TFG08: Laurent Simon, mplementation of CDCL SAT solver, 2016
TFG09: Mateu Villaret i Jordi Coll, Presentació SAT assignatura Programacio Declarativa. Aplicacions,

14 - Annexos

14.1 Implementacions solucionadors inicials

```
abstract class Solver{
  protected var solutionFound = false
  protected var oldtrail: List[Int] = Nil
  protected var eventManager: EventManager = new EventManager
  protected var correctSolution = false

  def solve(instance: Instance): Boolean = {
    oldtrail = Nil
    eventManager = new EventManager

    solutionFound = i_solve(instance)

    correctSolution = checkSolution(instance)

    solutionFound
  }

  protected def i_solve(instance: Instance): Boolean

  def variablesState: Array[State]

  def printEvents: Unit = eventManager.printEvents

  def printSolution: Unit = {
    if(solutionFound)
      println(oldtrail.sortWith(math.abs(_)<math.abs(_)))
  }

  def printSolutionCheck: Unit ={
    if(solutionFound)
      println(if(correctSolution) "OK" else "----->ERROR<-----")
    if(solutionFound && !correctSolution)
      throw new Exception
  }

  private def checkSolution(instance: Instance): Boolean ={
    val sol = oldtrail.toSet
    val allClausesSat = instance.instance.map(_.foldLeft(false)(_ ||
sol.contains(_))).forall(_==true)
    val allVarsAssigned = oldtrail.size == instance.numVariables
    allClausesSat && allVarsAssigned
  }

}
```

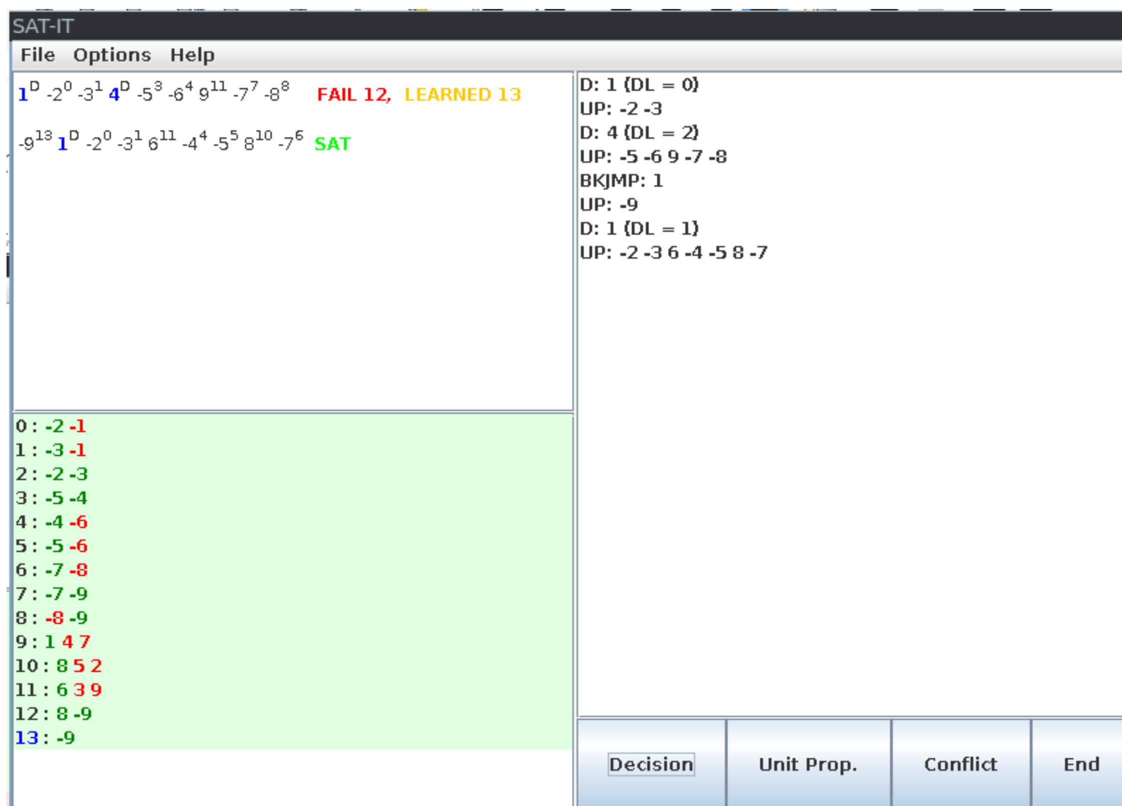
```
class RecursiveBacktracking extends Solver {

  override protected def i_solve(instance: Instance): Boolean = {
    ii_solve(new ClauseSet(instance))
  }

  def ii_solve(clauseSet: ClauseSet): Boolean = {
    if(clauseSet.containsEmptyClause) {
      eventManager.insertEvent(new oldBacktrack)
      return false
    }

    val nextLit = clauseSet.nextLit
    if(nextLit == 0) return true
    else{
      oldtrail = nextLit :: oldtrail
      eventManager.insertEvent(new Decision(nextLit))
      if(ii_solve(clauseSet.process(nextLit))) return true
      oldtrail = oldtrail.tail

      oldtrail = -nextLit :: oldtrail
      eventManager.insertEvent(new Decision(-nextLit))
      if(ii_solve(clauseSet.process(-nextLit))) return true
      oldtrail = oldtrail.tail
    }
    eventManager.insertEvent(new oldBacktrack)
    false
  }

}
```

```
class RecursiveDPLL extends Solver {

  override protected def i_solve(instance: Instance): Boolean = {
    ii_solve(new ClauseSet(instance))
  }

  def ii_solve(clauseSet: ClauseSet): Boolean = {
    var cs = clauseSet

    val numAssignedVarsBefore = oldtrail.length
    cs = unitPropagation(cs)
    val numPropagations = oldtrail.length - numAssignedVarsBefore

    if(cs.containsEmptyClause) {
      eventManager.insertEvent(new oldBacktrack)
      oldtrail = oldtrail.drop(numPropagations)
      return false
    }

    def nextLitStub: Int ={
      for(i <- 1 to clauseSet.numVariables)
        if(!oldtrail.contains(i) && !oldtrail.contains(-i))
          return i
      0
    }

    val nextLit = cs.nextLit
    if(nextLit == 0) return true
    else{
      eventManager.insertEvent(new Decision(nextLit))
      if(ii_solve(cs.addClause(Set(nextLit)))) return true

      eventManager.insertEvent(new Decision(-nextLit))
      if(ii_solve(cs.addClause(Set(-nextLit)))) return true
    }

    eventManager.insertEvent(new oldBacktrack)
    oldtrail = oldtrail.drop(numPropagations)

    false
  }

  def unitPropagation(clauseSet: ClauseSet): ClauseSet = {
    var cs = clauseSet
    val unitPropagations = new UnitPropagation

    var unitLit = cs.unitLiteral
    while(unitLit!=0){
      oldtrail = unitLit :: oldtrail
      unitPropagations.addLiteral(unitLit)

      cs = cs.unitSubsumption(unitLit)
      cs = cs.unitResolution(unitLit)

      unitLit = cs.unitLiteral
    }
    eventManager.insertEvent(unitPropagations)

    cs
  }

}
```

```
class ClauseSet(clauseSet: Set[Set[Int]]) {

  def this(instance: Instance) {
    this(instance.instance.map(_.toSet).toSet)
  }

  def nextLit: Int = {
    if (!clauseSet.isEmpty) {
      return math.abs(clauseSet.head.head)
    }
    0
  }

  def process(lit: Int): ClauseSet = {
    var cs: Set[Set[Int]] = clauseSet.filter(!_.apply(lit)) //unit-subsumption
    cs = cs.map(_.-(-lit)) //unit-resolution
    new ClauseSet(cs)
  }

  def containsEmptyClause: Boolean = {
    for (clause <- clauseSet)
      if (clause.isEmpty)
        return true
    false
  }

  def unitLiteral: Int = {
    for (clause <- clauseSet)
      if (clause.size == 1)
        return clause.head
    0
  }

  def addClause(clause: Set[Int]): ClauseSet = {
    new ClauseSet(clauseSet.+(clause))
  }

  def unitSubsumption(lit: Int): ClauseSet = new ClauseSet(clauseSet.filter(!
_.apply(lit)))

  def unitResolution(lit: Int): ClauseSet = new ClauseSet(clauseSet.map(_.-(-lit)))

  def numVariables: Int = {
    val maxAbs = (x: Int, y: Int) => math.max(math.abs(x), math.abs(y))
    clauseSet.map(_.fold(0)(maxAbs)).fold(0)(maxAbs)
  }

}
```

15 – Manual d’usuari i/o instal · lació

Per executar l'aplicació cal complir els requisits de l'apartat 6.



15.1 Instruccions per executar l'aplicació

A *GNU/Linux* i probablement altres entorns basats en *Unix* es pot executar l'aplicació simplement fent doble clic sobre l'executable "SAT-IT.jar".

En cas de que el mètode anterior falli o s'usi un altre sistema operatiu cal obrir un terminal, anar a la ruta on es troba l'executable i fer "java -jar SAT-IT.jar". Opcionalment podem passar un fitxer com a paràmetre perquè s'obri directament.

Per executar la interfície de terminal s'ha d'anar a la ruta on es trobi l'executable i fer "java -jar SAT-IT_term.jar *<input-file>*" on *<input-file>* és la ruta a a la instància a resoldre.