Università degli Studi di Brescia - Facoltà di Ingegneria

Corso di Intelligenza Artificiale

SAT PACK - MANUALE D'USO

Prof. Alfonso Gerevini

Catalin Roman

Matr. 036915

E-mail croman@neolt.it

Indice

INDICE **3**

1 DESCRIZIONE DEL SAT-PACK **5**

2 MAKEWFF **6**

2.1 Compilazione 6

2.2 Utilizzo 6

2.3 Opzioni 6

2.4 Esempio 6

3 RELSAT **7**

3.1 Compilazione 7

3.2 Flags di compilazione 7

3.3 Esecuzione 8

4 RELSAT 2.00 **10**

4.1 Scopo 10

4.2 Sintassi 10

4.3 Descrizione 10

4.4 Flags 10

4.5 Esempi 11

4.6 Configurazione dell'output 12

SATZ **14**

4.7 Compilazione 14

4.8 Esecuzione 14

5 WALKSAT **15**

5.1 Compilazione 15

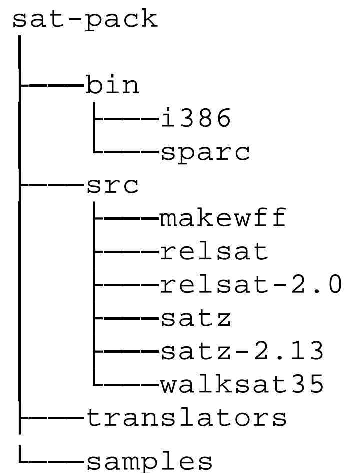
5.2 Parametri da modificare prima della compilazione 15

5.3 Parametri generali: 16

1 Descrizione del sat-pack

Questa rappresenta un SAT-package minimo.

La struttura delle directory è come segue:



La directory **bin** contiene gli eseguibili. È stata fatta la compilazione soltanto per l'architettura i386 sotto Linux. Si può fare eventualmente anche per altre architetture e/o altri OS.

src contiene i source files organizzati per directory:

makewff - un generatore di formule random - può generare formule sia in formato **f** che **cnf**.

relsat - per l'algoritmo SAT look-back enhanced di Bayardo e Schrag, come descritto in *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*.

relsat-2.0 – la versione 2.00 del relsat

satz – la versione originale del SATZ come descritto in *Look-ahead versus look-back for satisfiability problems*

satz-2.13 - la versione 2.13 del SATZ

walksat35 – Walksat versione 35.

La directory **translators** permette la conversione da un format **cnf** ad un altro. Il formato migliore da usare è il **cnf** .

Per esempio

cnf2f < il tmp.cnf > il tmp.f

o

f2cnf < il tmp.f > il tmp.cnf

2 makewff

2.1 Compilazione

Per la compilazione digitare semplicemente:

```
make
```

2.2 Utilizzo

Digitare **`makewff`** senza argomenti per avere le opzioni.

```
makewff
```

2.3 Opzioni

- **`-seed`** il seed può essere impostato usando il flag –seed, per inizializzare il generatore di numeri random. Per default il seed è determinato usando il clock.
- **`-cnf,-f,-kf`** sono tre formati differenti del file wff. -kf è un formato vecchio.
- **`clen`** è la lunghezza di ogni clausola,
- **`nvars`** il numero di variabile
- **`nclauses`** il numero di clausole.

2.4 Esempio

```
makewff -f 3 100 430 > tmp.wff
```

crea una formula 3CNF con 100 variabili e 430 clausole nel file tmp.wff.

3 Relsat

Questa sezione descrive il source code dell'algoritmo SAT look-back enhanced di Bayardo & Schrag, come descritto in: Bayardo & Schrag, *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*.

Questo particolare algoritmo è adattato per risolvere le istanze del mondo reale. C'è da notare che c'è ancora possibilità per il miglioramento di questo codice. È almeno 3 volte più lento (in termini di costanti) di una ben ottimizzata realizzazione del DP come Tableau, anche quando tutti i look-back enhancements sono disabilitati.

Finalmente, quest'algoritmo non compie alcun pre-processing delle istanze.

3.1 Compilazione

Il source è un tipico codice C++, e puo' essere compilato senza avvertimenti usando il g++ v2.7.2 per il SunOS, Solaris, SGI e RS6000. Si suppone che si possa compilare e girare su tutte le altre architetture che supportano questa versione di g++, o con minime modifiche su altri compilatori C++.

Digitare semplicemente :

```
make
```

3.2 Flags di compilazione

Si parla di vari *#define* nel file *SATSolver.c* che si possono commentare o no per avere diversi algoritmi. La maggior parte di questi flags dovrebbero essere flag di runtime.

- Il flag più importante è *RELEVANCE_BOUNDED_LEARNING* che costringe l'algoritmo ad usare *relevance-bounded learning*. Se per questo flag è eliminato il commento, l'algoritmo userà invece *relevance-bounded learning*. Si suggerisce a lasciare sempre il flag per migliori performance su istanze grandi, quanto tempo in tanti casi sembra piuttosto critico per buone performance.
- Se si vogliono disabilitare tutti i look-back enhancements, eliminare il commento al flag *NAIVE_SOLVER*. Questa scelta non è raccomandata per le scarse performance. È utile per vedere i miglioramenti ottenuti con i look-back enhancements. Se *NAIVE_SOLVER* non è commentato, *il flag RELEVANCE_BOUNDED_LEARNING* è ignorato.
- C'è in seguito un flag *ISAMP* che, se non commentato, dovrebbe essere accompagnato anche con un flag *NAIVE_SOLVER* non commentato. Questo flag permette all'algoritmo come l'algoritmo *ISAMP* randomized definito in Crawford e Backer – AAAI 94.
- Un altro flag è *MAX_TIME*, con cui si definisce l'ammontare di tempo CPU che l'algoritmo spenderà tentando di risolvere un'istanza prima di uscire. Se si desidera che l'algoritmo non abbandoni mai, eliminare il commento al *NO_TIME_LIMIT*.
- Quando il flag *OUTPUT_SOLUTION* non è commentato indica la volontà che l'algoritmo fornisca la soluzione quando/se è stata trovata. Il formato della soluzione è il nome della variabile che appare senza trattino se è *True*, ed appare con un trattino se è *False*.

- Il flag *SKIP_PURE_LITERALS* permette all'algoritmo di applicare una molto semplice regola pure-literal con un overhead basso. Sembra pero che questo non aiuti su alcuna stanza. Questo flag funziona correttamente solamente quando si applicano look-back enhancements (significa che il flag *NAIVE_SOLVER* è commentato).
- Il flag *FAVOR_SMALL_CLAUSES* aiuta l'algoritmo usando piccole clausole come reason per escludere un valore da qualche dominio di variabili. Invece quando questo è commentato, l'algoritmo selezionerà in modo random dal set di clausole che potrebbero essere usate per escludere un valore (sebbene non in maniera di uniforme.. favorisce quelle clausole verso la fine dell'elenco di clausole). L'overhead per aiutare le piccole clausole è trascurabile, così può essere lasciato non commentato.
- Il parametro *MAX_TO_TRY* specifica il numero massimo di variabili che saranno selezionate come potenziali variabili di ramificazione ad ogni ramo. Al momento questo è impostato a *10*.
- Un flag molto utile è *PRINT_STACK*, che quando non commentato permette all'algoritmo di scaricare ripetutamente fuori lo stato del backtracking stack.Questo permette di costatare il progresso dell'algoritmo che sta risolvendo un'istanza. È raccomandato di non commentarlo quando l'algoritmo è usato in modo interattivo.
- Il flag *PRINT_FREQUENCY* determina la frequenza con la quale è stampata lo stack. Questo è il termine "ramo percorsi", così l'attuale frequenza è dipendente dall'istanza. Il valore di default è stato impostato a 100.

Esempio di output è come segue:

```
[86][/home/sat-pack/bin/i386] rel_sat 4 ../samples/hanoi4.cnf
Istance is: samples/hanoi4.cnf
Random number seed: 0
Learn order is: 4
>177
>184 20 29 22 6 6 6 22 16 8 2 14 4 9 82 51 44
>184 20 29 22 6 6 6 22 16 8 2 14 8 9 39
>184 20 29 22 6 6 6 22 16 8 2 14 27 98
>184 20 29 22 6 6 6 22 16 8 2 48 32 34
>184 20 29 22 6 6 6 22 16 8 2 56 4 6 4
...
```

I numeri dopo ogni **>** risultano dal flag *PRINT_STACK*. Ogni numero rappresenta il numero di variabili unit-domained prima del prossimo ramo da espandere. Nel caso di sopra, sono etichettate *184* variabili prima di giungere al primo punto di ramificazione (un punto backtracking stack dove sono aperte gli ambi valori della variabile selezionata).

3.3 Esecuzione

Dopo avere eseguito make, digitare rel_sat insieme a 3 argomenti.

Presumendo che l'algoritmo è stato compilato con il flag *NAIVE_SOLVER* commentato, il primo argomento è un parametro di runtime che specifica l'ordine del learning da applicare.

Quando si usa il relevance-bounded learning, gli ordini di learning *1* e *2* non devono essere usati. Senza relevance-bounded learning, il *2* dovrebbe funzionare bene. 1 non farà niente più del CBJ, quanto tempo limplementazione non supporta clausole unarie. È raccomandato usare ordini di learning *3* e *4*, quali funzionano bene su più esempi. Qualsiasi cosa più di *4* causa un sostanziale overhead, tuttavia qualche volta per istanze veramente difficili ancora è utile. Un ordine di learning 0 dice all'algoritmo di applicare il conflict-directed backjumping senza nessun learning.

Se il flag *NAIVE_SOLVER* non è stato commentato durante la compilazione, comunque questo parametro è ancora richiesto dall'eseguibile, ma il suo valore è ignorato completamente.

Il secondo argomento accettato dall'eseguibile è il nome del file dell'istanza da risolvere. Attualmente, l'eseguibile accetta le istanze con il formato cnf DIMACS.

Il terzo argomento è un numero random seed. È opzionale, e se non è specificato, il seed usato è *0*.

L'eseguibile prova a risolvere l'istanza ripetutamente. Ad ogni tentativo, riporta *SAT*, *UNSAT*, o *TIME LIMIT EXPIRED*, insieme a rami esplorati (la dimensione dell'albero di ricerca), variabile etichettate (un'altra dimensione metrica dell'albero di ricerca), e CPU time (sec). Il tempo CPU non include il tempo per leggere l'istanza. Dopo *100* tentativi per risolvendo l'istanza, viene riportato il numero di tentativi, il programma fallisce e si ferma. Il tempo di abbandono è specificato dal *MAX_TIME* impostato prima della compilazione.

4 Relsat 2.00

4.1 Scopo

Risolvere un'istanza data del problema di soddisfacibilità proposizionale SAT.

4.2 Sintassi

```
relsat [- #{MaxSolutions | un | c}] [-l LearnOrder]
       [-p PreProcessLevel] [-t {TimeoutInterval | n}]
       [-u {StatusInterval | n}] [-s RandomSeed]
       [-r {RestartInterval | n}] [-f FudgeFactor]
       [-o OutputInstance] [-c PostProcessLevel]
       [-i ProcessingIterationBound] InputInstance
```

4.3 Descrizione

Tentativi di risolvere l'istanza SAT memorizzata nel file specificato da *InputInstance*. L'unico formato di input supportato è il DIMACS.

Dipendendo dal flag a linea di comando **-#**, relsat può essere usato per determinare la soddisfacibilità dell'istanza, ottenere un numero specificato di soluzioni dell'istanza, ottenere tutte le soluzioni dell'istanza, ottenere un numero di soluzioni distinte. In qualsiasi caso, se relsat è capace di determinare la soddisfacibilità, riporterà almeno una soluzione.

4.4 Flags

Nota: Tutti i flag richiedono un argomento. Se un flag non è specificato, l'effetto è come se fosse stato specificato con il valore predefinito indicato.

-# dice all'algoritmo di limitare il numero di soluzioni prodotte quando l'argomento è un numero intero positivo, di fornire le soluzioni il carattere **a**, di fornire solamente il numero di soluzioni oltre ad una soluzione singola se dovesse esistere una il carattere **c**. Il valore predefinito è *1*.

-l specifica l'ordine di relevance-bounded learning da applicare durante la fase di scoperta della soluzione - un numero intero non negativo. Il valore *0* indica che nessun learning sarà applicato, tuttavia il conflict-directed backjumping rimane attivo. Il valore predefinito è *3*.

-p gli argomenti permessi sono *0*, *1*, *2*, *3*, ognuno che specifica il livello di elaborazione da applicare all'istanza di input prima di entrare nella fase di scoperta della soluzione:

- *0* nessuna lavorazione.
- *1* rimozione delle clausole ridondanti, inferenza delle clausola unità e riduzione attraverso unit propagation, e risoluzione del terzo ordine limitata.

2 oltre a *1*, inferenza delle clausole attraverso unit propagation.

3 oltre a *2*, è fatto un tentativo per ridurre la dimensione di ogni clausola di input attraverso unit propagation.

Nessuna di queste procedure rinomina le variabile o influisce la serie di soluzioni in alcun modo. Il valore predefinito è *1*.

-t specifica il numero di secondi che l'algoritmo spende nella fase di scoperta della soluzione prima del timeout - un numero intero non negativo, o nessun timeout - il carattere **n**. Il valore di default è *43200 = 12 ore*.

-u specifica il numero di secondi che passano fra gli aggiornamenti dello stato - un numero intero positivo, o nessun aggiornamento dello stato - il carattere **n**. Durante le fasi di pre e post-processing questo controlla quanto spesso l'algoritmo riporta la percentuale fatta per la sottofase corrente. Durante la fase di scoperta della soluzione, questo controlla quanto spesso l'algoritmo riporta informazioni, incluso la profondità di ramo dello stack, le selezioni dei rami fatti, contraddizioni scoperte, la struttura dello stack, e quando usato con **-#c**, il numero di soluzioni così identificate. Il valore predefinito è *10*.

-s specifica il numero random seed. I valori permessi sono numeri interi positivi. L'algoritmo ordina in modo random le etichette delle variabili, ed anche e la selezione dei il nogood. Relsat include il proprio generatore di numeri random per sostenere il comportamento deterministico attraverso le varie piattaforme. Il valore predefinito è *1*.

-r specifica il periodo in secondi che passa tra i riavvii dell'algoritmo - numero intero positivo, o che riavvio compiuto - il carattere **n**. I riavvii possono risultare nel comportamento non deterministico a causa della velocità e l'utilizzo differenti del microprocessore. L'uso dei riavvii proibiscono anche le opzioni **-#c** e **-#a** quanto tempo la completezza non può essere garantita. In più, se è richiesta più di una soluzione, non è garantito che sia unica. Il valore predefinito è **n**.

-f fattore fudge per determinare variabili equivalenti durante la selezione della variabile. Il valore di default è *0.9*. Valori più bassi possono essere più adatti quando si usa il riavvio (l'opzione **-r**).

-o specifica il nome del file in cui scrivere l'istanza nel caso di timeout, o nel caso che nessuna istanza deve essere scritta il carattere n. Se l'istanza è trovata ad essere non soddisfacibile, o tutte le soluzioni richieste (o numero di soluzioni) sono trovate prima del timeout, allora questo ed il seguente flag sono ignorati. Questo flag è utile solamente quando sia il learning e/o il pre/post-processing sono applicati all'istanza.

-c il livello di post-processing. I valori permessi sono gli stessi come quelli per il flag **-p**. Specifica il livello di processing applicato all'istanza in uscita prima che sia scritto nel file di output (come specificato da **-o**). Il valore predefinito è *3*.

-i specifica il numero massimo di iterazioni compiuto dalla fase di processing - numero intero positivo, o che il processing deve iterare finché nessun'altra modifica sia fatta all'istanza (il carattere **n**). Il valore predefinito è **n**.

4.5 Esempi

- Dovuto ai valori di default, solamente l'istanza di input è richiesta per poter determinare la soddisfacibilità (è disponibile una soluzione se esiste una):

```
relsat input.cnf
```

- Per fornire tutte le soluzioni:

```
relsat -#a input.cnf
```

- Per fornire il numero di soluzioni all'istanza in input:

```
relsat -#c input.cnf
```

Nota: il conteggio è spesso significativamente più veloce che fornire tutte le soluzioni.

- Processing solamente dell'istanza, e salvare il risultato in un file specificato:

```
relsat -p0 -t0 -o processed_instance.cnf input.cnf
```

Nota: Il livello post-processing assume per il valore di default *3*. Abbiamo disabilitato il pre-processing per prevenire che l'istanza sia trattata due volte.

- Tentare di determinare una soluzione usando intervalli di riavvio di *30* secondi ed un fattore fudge di *0.3*, timeout di un'ora.

```
relsat -r30 -f.3 -t3600 input.cnf
```

- Usare riavvii con timeout come un metodo di pre-processing per provare la soluzione completa:

```
relsat -r30 -f.3 -t3600 input.cnf; relsat -p0 -tn timeout.cnf
```

4.6 Configurazione dell'output

Tutte le uscite sono dirette allo standard out. Motivato dalle specifiche del formato DIMACS, i messaggi di proseguimento ed altre comunicazioni informative sono forniti da un `c` nella prima colonna come segue:

```
c Instance is: input.cnf
c Preprocessing level: 1
c Output preprocessing level: 3
...
```

Le soluzioni sono fornite cosi come sono state trovate, ognuna con le sue linee, come segue:

```
Solution 1: 23 45 99 ....
Solution 2: 23 45 98 ...
...
```

Ogni numero che segue **:** indica quali variabili sono *True* per soddisfare la formula. Tutte le altre variabili dovrebbero essere impostate su *False*.

Se il relsat dovesse determinare la non soddisfacibilità, in uscita si leggerà *UNSAT*. Altrimenti, una volta che il numero desiderato di soluzioni è stato trovato (o tutte le soluzioni identificate), in uscita si leggerà *SAT*. Se capitasse un timeout prima di un tale caso, in uscita si leggerà *TIME LIMIT EXPIRED*.

Quando si conta il numero di soluzioni (l'opzione **-#c**), relsat fornirà solamente una singola soluzione se dovesse esistere una, formattata come sopra. In più, prima di riportare *SAT*, relsat fornirà il numero di soluzioni come segue:

```
Number of solutions: 120
```

Se dovesse succedere un timeout prima di determinare il numero di soluzioni, o determinare se l'istanza è non soddisfacibile, allora non sarà fornita alcuna informazione.

Satz

4.7 Compilazione

Per la compilazione sotto Unix oppure Linux digitare semplicemente:

```
gcc -O3 -o satz satz.c
```

4.8 Esecuzione

Per lanciare il programma in esecuzione digitare semplicemente:

```
satz cnf-file
```

where **`cnf-file`** è un file nel formato cnf DIMACS.

5 Walksat

Walksat tenta di trovare un modello soddisfacente di una formula cnf generalizzata.

Accetta solamente il formato .cnf, come per esempio:

```
c Optional comments
c at start of file.
c The "p" line specifies cnf format, number vars, number clauses
p cnf 3 2
1 -3 0
2 3 -1 0
```

Per avere istruzioni senza guardare a questo file, digitare semplicemente **walksat** dalla linea di comando con un parametro **-help**.

5.1 Compilazione

Per compilare walksat digitare semplicemente:

```
make
```

5.2 Parametri da modificare prima della compilazione

Questi parametri possono essere eventualmente modificati.

MAXATOM - limita il numero di atomi; si può aumentare questo se necessario per gestire problemi più grandi.

MAXCLAUSE - limita il numero di clausole; si può aumentare questo se necessario, od opzionalmente, compilare con il comando

```
make huge
```

L'opzione **huge** alloca dinamicamente l'array dei pointer delle clausole. Questo usa memoria più efficacemente (sia per problemi molto grandi che molto piccoli), ma causa un abbassamento di circa *15%* della velocità di esecuzione.

Per vedere le altre opzioni di compilazione aprire il file **makefile**.

Walksat legge da standard in e scrive a standard out.

I parametri passati a walksat sono (N e M sono numeri interi):

5.3 Parametri generali:

-seed N usare seed random specificato

-cutoff N numero massimo di flips in una prova

-tries N numero massimo di tentativi

-target N stop quando non più di *N* clausole sono non soddisfatte (default = *0*)

-numsol N stop dopo aver trovato *N* soluzioni

-init FILE assegnamento iniziale è contenuto nel **FILE**;

imposta le variabili non incluse nel **FILE** su *False*

-partial FILE assegnamento iniziale è contenuto nel **FILE**;

imposta le variabili non incluse nel **FILE** random

L'euristica sceglie una variabile all'interno di una clausola, dopo che la clausola non soddisfatta è scelta in modo random; notiamo che dovrebbe essere specificata solamente una singola di queste euristiche.

-random scelta in modo random

-best scelta la variabile che rompe il minor numero di altre clausole quando viene modificata

-tabu N scelta la migliore, eccetto con la lista tabu di lunghezza N

-novelty euristica nuova creata da McAllester

Le seguenti cambiano l'euristica precedente:

-noise N M ogni *N/M* flips, scelta random, piuttosto che dall'euristica

-noise N stesso come **-noise N 100**

-super incrementare il numero di flips ad ogni prova usando la progressione superlinear

Stampa:

-hamming *TARGET_FILE DATA_FILE SAMPLE_FREQUENCY*

-trace N stampa statistiche ogni *N* flips

-assign N stampa assegnamenti a *N*, *2N*, ...

-sol stampa assegnamenti soddisfacenti

-low stampa gli assegnamenti più bassi ad ogni prova

-bad	stampa le clausole non soddisfacibili ad ogni prova
-tail N	considera che la coda comincia a $nvars*N$
-solcnf	stampa gli assegnamenti soddisfacibili in formato cnf, ed esce
-sample N	campiona il livello di noise ad ogni N flips