Implementazione di un dimostratore di teoremi per risoluzione ordinata
Ragionamento Automatico 2012-2013

Alessandro Gottoli vr352595

4 febbraio 2013

Introduzione

L'algoritmo del ciclo della clausola data del dimostratore di teoremi di questo progetto è stato implementato nel linguaggio di programmazione Java per essere eseguibile su diversi sistemi operativi ed è stato testato su Ubuntu Linux 12.04 LTS 64 bit. Il codice è contenuto nel package thProver e si divide in tre parti principali: Parser, Interfaccia, Ciclo della clausola data.

1 Parser

Il programma accetta in ingresso file contenenti un insieme di clausole nella logica del primo ordine in due formati differenti, quindi sono stati implementati due parser. Uno accetta file in formato TPTP (solo sottogrammatica "cnf"), l'altro in un formato appositamente definito da me che ha una sintassi più simile a quella che troviamo in letteratura e che era stato pensato per permettere di definire opzioni sull'ordinamento delle le clausole specificando precedenze (prec:) e pesi (weights: e weightVars), ma poi si è deciso di aggiungere la possibilità di aggiungere tali opzioni anche all'inizio di un file tptp anche se non previsto dalla grammatica ufficiale.

Questi parser si trovano nei package thProver.parserTptp e thProver.parser e per la generazione è stato utilizzato JavaCC, quindi è stata definita la grammatica del linguaggio accettato e le azioni da fare per la costruzione delle clausole nei file `GrammarTptp.jj` e `Grammar.jj` e le classi java sono state create automaticamente col comando `javacc ‘‘file’’.jj`.

Vediamo brevemente il mio linguaggio. Per prima cosa dobbiamo specificare le costanti (se ci sono) scrivendo `const:` $costante_1$`,` $costante_2$`, ...,`$costante_n$.

Poi possiamo specificare le precedenze scrivendo `prec:` $simbolo_1 > simbolo_2 > \ldots > simbolo_n$ dove simbolo può essere un simbolo di predicato, funzione o costante. Si possono definire precedenze parziali scrivendo `prec:` $precedenza_1$ `;` $precedenza_2$ `; ...;` $precedenza_n$. Definire la precedenza non è obbligatorio perché è possibile utilizzare una precedenza "standard" definita nella classe ordinamento in cui *predicati* > *funzioni* > *costanti* e ogni predicato, funzione e

costante è ordinata in modo da definire come maggiore quella che viene prima nell'ordinamento lessicografico.

Si possono inoltre definire i vari pesi per utilizzarli nell'ordinamento di Knuth-Bendix utilizzando

- `weightVars:` *integer* per definire il peso da dare a tutte le variabili (così tutte avranno lo stesso peso senza doverle esplicitare)
- `weights:` `simbolo`$_1$ `=` *integer* `;` `simbolo`$_2$ `=` *integer* `;` ... `;` `simbolo`$_n$ `=` *integer* per definire il peso di predicati, funzioni e costanti.

Se l'utente non inserisce tutti i pesi, l'ordinamento potrebbe trovarsi in situazioni anomale, quindi ho deciso di specificare nella classe Ordering il valore base per i simboli non specificati e si possono dare valori diversi in caso si tratti di predicati, funzioni o costanti (io ho deciso di dare valore 1). Come per le precedenze è possibile utilizzare un peso "standard" definito nella classe ordinamento in cui ho definito pesi per atomi, funzioni, variabili e costanti.

Adesso vediamo la sintassi di ogni clausola. Ogni clausola è formata da uno o più letterali separati dal simbolo '`|`' che indica l'*or*. Ogni letterale è composto da un Atomo che può essere definito come negativo se preceduto dal simbolo '`!`' o '`~`'. Ogni atomo è poi composto da un simbolo di predicato che nella mia sintassi deve essere scritto con lettere maiuscole (per essere simile alla convenzione usata in classe) e seguito da una eventuale lista di termini racchiusi tra '`(`' e '`)`'. I termini si scrivono in minuscolo e si distinguono le funzioni (seguite dai propri argomenti che saranno ulteriori termini racchiusi tra '`(`' e '`)`'), le variabili e le costanti (che sono quelle definite all'inizio).

Adesso possiamo inserire le clausole nel seguente modo:

- `sos:` *clausola*$_1$ `;` *clausola*$_2$ `;` ... `;` *clausola*$_n$ per specificare le clausole da inserire nell'insieme di supporto (se usiamo questa strategia)
- `clauses:` *clausola*$_1$ `;` *clausola*$_2$ `;` ... `;` *clausola*$_n$ per specificare le altre clausole.

Anche se non si ha intenzione di utilizzare l'insieme di supporto si possono specificare lo stesso come 'sos' le clausole provenienti dalla trasformazione in forma clausale della negazione della congettura, perché se non verrà attivata la strategia verranno unite alle altre clausole.

Anche per il formato tptp è possibile utilizzare la strategia dell'insieme di supporto, infatti nella sintassi c'è un campo *ruolo* e nell'insieme di supporto inserisco quelle che hanno come ruolo *"negated_conjecture"*.

Un esempio di file di input è mostrato in Figura 1

```
const: a,b,c
prec: ON>GREEN ; a>b>c
weightVars: 1
weights: ON = 1; GREEN = 1; a = 1; b = 1; c = 1
sos: ~GREEN(x) | GREEN(y) | ~ON(x,y)
clauses: ON(a,b) ; ON(b,c) ; GREEN(a) ; ~GREEN(c)
```

Figura 1: Esempio di file di input.

Quando viene eseguito, il parser istanzia un oggetto CNFFormula che conterrà tutti i dati relativi all'input. Esso contiene diverse HashMap per salvare atomi e termini così da condividere oggetti uguali senza creare nuove istanze, e mappare simbolo con la propria arietà per controllare che non vengano passati un numero diverso di argomenti allo stesso simbolo.

Ci sono poi altre strutture per le precedenze e i pesi. Quando si processano le variabili di clausole diverse bisogna intenderle disgiunte anche se hanno lo stesso simbolo, allora ho deciso di rinominarle alla creazione assegnandogli come nuovo simbolo univoco `simbolo_indiceclausola` così non c'è problema di rinominarle ad ogni risoluzione tra clausole con nome di variabile uguale.

2 Interfaccia

L'elaborato è fornito di due interfacce: una a riga di comando e una grafica.

L'*interfaccia a riga di comando* è quella predefinita e come si può vedere dall'"usage" si possono attivare i diversi parametri specificando le diverse opzioni quando si avvia il programma.

Per prima cosa si specifica l'input e sono disponibili due alternative:

- `-i` indica che si vuole inserire la formula da standard input in modo iterativo
- `/percorso_del_file/input.p` per caricare da file

È obbligatorio specificarne una altrimenti il programma terminerà.

Poi può specificare di utilizzare la strategia dell'insieme di supporto con "`-sos`", di default non è attivata.

Di default non si utilizza nessun ordinamento, però sono disponibili le seguenti opzioni per selezionarne uno:

- `-lex` per l'ordinamento ricorsivo a cammini con status lessicografico
- `-mul` per l'ordinamento ricorsivo a cammini con status multiinsieme
- `-kbo` per l'ordinamento di Knuth-Bendix

Una volta scelto l'ordinamento il programma utilizzerà di default quello "standard" definito da me nel programma (utile se non sono state specificate le precedenze o i pesi nell'input), ma digitando "`-usP`" si possono utilizzare le precedenze e i pesi definiti nell'input.

Il ciclo della clausola data ha due versioni che differiscono principalmente sull'insieme di clausole da mantenere interridotto. L'elaborato le implementa entrambe ed utilizza di default la versione *à la Otter*, mentre è possibile selezionare la versione *à la E* con il parametro `-E`.

È stata inoltre aggiunta l'opzione "`-limit`*n_secondi*" che permette di fermare la computazione quando è passato il tempo indicato.

L'*interfaccia grafica* è attivabile specificando l'opzione "`-gui`" quando si lancia il programma e verrà visualizzata una finestra in cui poter inserire direttamente l'input oppure selezionare un file da cui leggerlo. Si possono abilitare le varie opzioni viste sopra semplicemente cliccando il menù relativo. Questa interfaccia offre la possibilità di fare un solo parsing e poi eseguire diverse prove di soddisfacibilità anche cambiando le opzioni.

In entrambe le interfacce c'è la possibilità (se l'input risulta insoddisfacibile) di salvare un file con sintassi 'dot' in cui è specificato il grafo diretto della prova di refutazione indicando le clausole e le regole di inferenza utilizzate indicando anche la sostituzione. Inoltre se sulla macchina è installato 'dot' (su linux pacchetto *graphviz*) è possibile esportare anche l'immagine del grafo in formato jpg, ps o pdf. Per esempio usando Otter e ordinamento lessicografico sull'esempio di prima si ottiene come prova di insoddisfacibiltà il sorgente in Figura 2 e l'immagine in Figura 3.

```
digraph {
   nodesep="1.5"; ranksep=2;
   node [shape=plaintext];
   edge [color=gray];
   "GREEN(c)" -> "[]"
      [labelfontcolor=black,labelfontsize="12",headlabel="Ordered␣Resolution\n{␣}",labeldistance="6"];
   "~GREEN(b)␣|␣GREEN(c)" -> "GREEN(c)"
      [labelfontcolor=black,labelfontsize="12",headlabel="Ordered␣Resolution\n{␣}",labeldistance="6"];
   "~GREEN(x_0)␣|␣GREEN(y_0)␣|␣~ON(x_0,y_0)" -> "~GREEN(b)␣|␣GREEN(c)"
      [labelfontcolor=black,labelfontsize="12",headlabel="Ordered␣Resolution\n{␣x_0␣<-␣b,␣y_0␣<-␣c␣}",labeldistance="6"];
   "ON(b,c)" -> "~GREEN(b)␣|␣GREEN(c)" ;
   "GREEN(b)" -> "GREEN(c)" ;
   "GREEN(a)" -> "GREEN(b)"
      [labelfontcolor=black,labelfontsize="12",headlabel="Clausal␣Simplification\n{␣}",labeldistance="6"];
   "~GREEN(a)␣|␣GREEN(b)" -> "GREEN(b)" ;
   "~GREEN(x_0)␣|␣GREEN(y_0)␣|␣~ON(x_0,y_0)" -> "~GREEN(a)␣|␣GREEN(b)"
      [labelfontcolor=black,labelfontsize="12",headlabel="Ordered␣Resolution\n{␣x_0␣<-␣a,␣y_0␣<-␣b␣}",labeldistance="6"];
   "ON(a,b)" -> "~GREEN(a)␣|␣GREEN(b)" ;
   "~GREEN(c)" -> "[]" ;
}
```

Figura 2: File sorgente per 'dot' del grafo della prova.

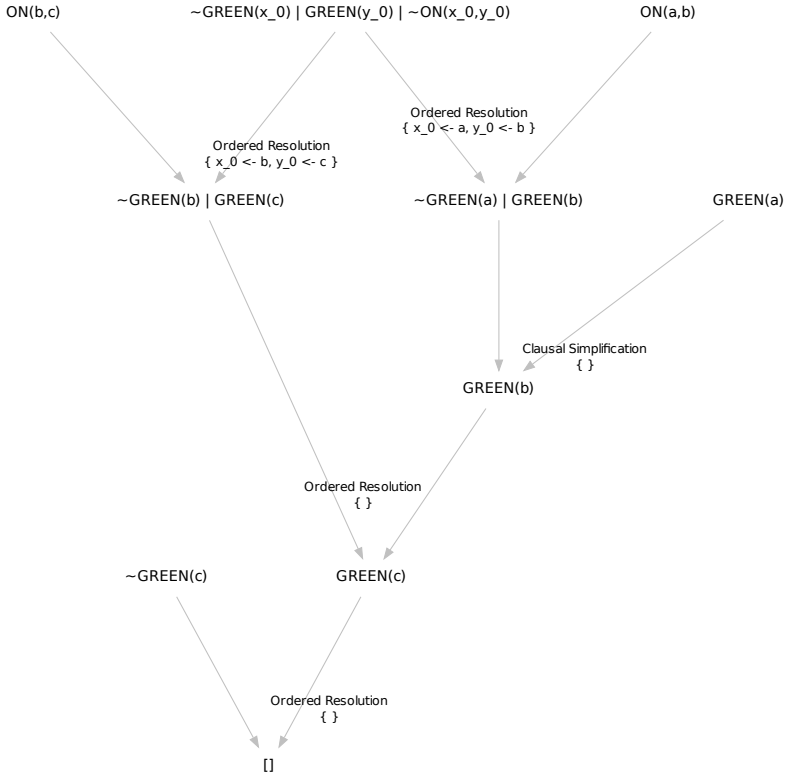

Figura 3: Grafo della prova.

3 Ciclo della clausola data

Adesso vediamo il codice che esegue la computazione vera e propria.

Il main dopo aver processato i parametri dati dall'utente, aver letto l'input ed eseguegiuto il parsing, crea un oggetto Ordering che implementa i tre ordinamenti e setta quello selezionato dall'utente. Siccome nelle api java non c'è una classe che implementi i multiinsiemi, ho definito MultiSet in cui salvo gli oggetti con il loro numero di occorrenze utilizzando un'HashMap<Object, Integer>. Per poter adottare questa implementazione si è resa necessaria l'implementazione dei metodi `equals` e `hashCode`.

Adesso possiamo verificare l'insoddisfacbilità della formula inserita. Il ciclo della clausola data è implementato nella classe GivenClauseProver nel metodo `satisfiable`. Siccome "`To_Select`" deve essere ordinato per estarre la clausoladata "migliore" l'ho implementato come una PriorityQueue in cui scelgo la clausola data con il *minor numero di simboli* che è un'ordinamento *equo* (nel caso di valore uguale la clausola più vecchia ha la precedenza). "Selected" invece è un semplice ArrayList perché devo scorrerlo tutto ogni volta per fare risoluzione.

InferenceSystem implementa le regole del sistema di inferenza e contiene anche il metodo `mgu` per calcolare l'*unificatore più generale* tra due letterali. Le sostituzioni sono definite in Substitution con una HashMap<Variable, Term> che rende veloce la ricerca. Nella classe si trovano anche i vari metodi per controllarne l'*idempotenza*. Quando si applica una sostituzione rinominiamo le variabili aggiornando l'indice della clausola, ma può capitare di avere contemporaneamente `x_1` e `x_2`, allora per disambiguarle nella nuova clausola si utilizza "`'`" ottenendo così `x_3` e `x'_3`.

Siccome si lavora principalmente sulle clausole, il lavoro di ricerca di implementazione efficiente è stato fatto proprio in Clause. I letterali che compongono la clausola sono inseriti sia in un Set chiamato `literals` sia in due liste (`posLits` e `negLits`). Questo mi permette di lavorare solo su letterali con il segno che mi interessa evitando controlli inutili e con spreco di memoria minimo.

Siccome durante la computazione chiamiamo molte volte metodi ricorsivi abbastanza impegnativi il cui risultato non cambia da chiamata a chiamata mi salvo il risultato per usi futuri. Per esempio ad ogni iterazione del ciclo bisogna riordinare la coda di priorità calcolando il numero di simboli, allora me lo salvo in `numSymbs`. Anche i letterali massimali andrebbero calcolati ad ogni risoluzione, quindi me li salvo in `maximalLiterals` e analogamente ai lettarali anche in `posMaximalLits` e `negMaximalLits`.

Poi ci sono dei campi aggiuntivi `parents`, `substitutionFrom` e `rule` che ci permettono di tenere traccia dell'albero di derivazione e di costruire il grafo.

In Clause ci sono due metodi per la sussunzione. `subsumes` è la mia versione che implementa sia la *sussunzione propria* sia quella *di varianti* e funziona confrontando delle liste con lo stesso simbolo di predicato e stesso segno e cercando di unificare i letterali tra le due clausole. `subsumesChangLee` invece è l'implementazione descritta nel Chang-Lee a pag. 95 e prevede di calcolare vari risolventi tra le clausole. A livello di test la mia versione risulta più veloce perché richiede solo di trovare gli `mgu` e non fare tutta la risoluzione. Si può provare la versione del libro con l'opzione `-changlee` ma serve solo per test.

4 Risultati degli esperimenti

Nella maggior parte dei casi utilizzando l'ordinamento si generano meno clausole rispetto a non utilizzarli e risultano più veloci.

Anche l'utilizzo dell'insieme di supporto riduce di molto il tempo di esecuzione della procedurae spesso risulta migliore anche rispetto agli ordinamenti, probabilmente anche perché le precedenze non sono state scelte ad hoc.

Si è provato ad unire ordinamenti e strategia dell'insieme di supporto per vedere se si ottenevano ulteriori miglioramenti, ma dagli esperimenti si è notato che *non sono compatibili*, infatti molto spesso da risultato errato. Un esempio è

```
> vr352595.sh
  ./Problemi/tptp.file/ALG002-1.p
  -sos

/* info */
14 clauses from parsing,
2780 clauses generated,
1651 clauses deleted,
963 clauses in To_Select,
160 clauses in Selected,
in 6 sec. and 583 millisec..
response: UNSATISFIABLE.
```

```
> vr352595.sh
  ./Problemi/tptp.file/ALG002-1.p
  -sos -kbo

/* info */
14 clauses from parsing,
0 clauses generated,
0 clauses deleted,
0 clauses in To_Select,
14 clauses in Selected,
in 5 millisec..
response: SATISFIABLE.
```

Infatti per fare risoluzione bisogna per forza unificare usando `greater_than_0(X)` che però con kbo non risulta tra i letterali massimali. Addirittura se proviamo con ordinamento lessicografico o multiinsieme la computazione sembra divergere. Si possono vedere questi risultati in alcuni esperimenti riportati in Tabella 1. Per questione di spazio è stato espresso con T il numero di clausole in "`To_Select`", S il numero di clausole in "`Selected`", G il numero di clausole generate e D il numero di clausole eliminate.

Sono state confrontate anche la versione à la Otter con quella à la E e come ci si aspettava il numero di clausole generate dalla versione à la E sono maggiori, ma nonostante questo la maggior parte dei casi è risultata più veloce. Alcuni confronti tra le versioni con e senza l'utilizzo degli ordinamenti o dell'insieme di supporto si trovano nella Tabella 1 per à la Otter e nella Tabella 2.

Ho provato anche un'approccio diverso del ciclo della clausola data. Mentre nell'originale si fa risoluzione tra la data e una clausola in "`Selected`" e si calcolano i fattori della clausola data per poi aggiungere tutte le nuove clausole in "`To_Select`"; io ho provato a calcolare tutti i risolventi tra la clausola data e le clausole in "`Selected`" compresi quelli calcolati sui fattori di esse e aggiungendo solo questi risolventi in "`To_Select`". Il risultato della prova di soddisfacibilità è lo stesso, anche se il mio esperimento risulta più lento. Se si desidera è possibile provarlo inserendo il parametro aggiuntivo `-vAll` all'avvio del programma ed è disponibile solo nell'interfaccia a riga di comando perché serve solo per test. Di default comunque verrà utilizzata la versione corretta come da specifica. I rilevamenti effettuati sono stati riportati in Tabella 3 nella quale è stato inserito anche il confronto tra i due tipi di sussunzione menzionati precedentemente.

Alcuni esercizi fatti in classe sono stati risolti con l'elaborato e i risultati si trovano in Tabella 4.

Nelle tabelle sono stati riportati solo i risultati più interessanti, tutti i rilevamenti delle prove effettuate si possono trovare nella cartella `risultati` relativa ad ogni problema, compresi i grafi con la derivazione.

File	**à la Otter** Opzioni	Risposta	G	D	T	S	tempo
ALG002-1.p		*SAT*	31 959	20 178	10 973	803	*timeout*
	-lex	*SAT*	15 229	6963	7595	685	*timeout*
	-mul	*SAT*	15 156	6905	7593	672	*timeout*
	-kbo	*SAT*	46 897	37 897	28	9395	*timeout*
	-sos	UNSAT	2780	1651	963	161	6 sec. 141 millisec.
	-sos -kbo	~~SAT~~	0	0	0	14	5 millisec.
COL123-2.p		*SAT*	12 892	2869	8900	1104	*timeout*
	-lex	UNSAT	4080	257	3316	490	35 sec. 459 millisec.
	-mul	UNSAT	3743	2627	911	212	8 sec. 553 millisec.
	-kbo	*SAT*	34 853	26 975	6932	954	*timeout*
	-sos	UNSAT	28 691	19 587	7026	2070	4 min. 24 sec.
	-sos -kbo	~~SAT~~	501	214	174	120	1 sec. 645 millisec.
COM001-1.p		UNSAT	13 364	12 170	903	197	6 sec. 336 millisec.
	-lex	UNSAT	1071	874	114	53	1 sec. 131 millisec.
	-mul	UNSAT	883	705	103	50	1 sec. 18 millisec.
	-kbo	UNSAT	1071	874	114	53	1 sec. 213 millisec.
	-sos	UNSAT	48	5	25	28	156 millisec.
	-sos -kbo	*SAT*	20 665	13 112	6377	1187	*timeout*
PLA001-1.p		UNSAT	272	66	145	76	1 sec. 139 millisec.
	-lex	UNSAT	216	104	44	83	562 millisec.
	-mul	UNSAT	213	101	44	83	583 millisec.
	-kbo	UNSAT	216	104	44	83	550 millisec.
	-sos	UNSAT	117	49	31	52	346 millisec.
	-sos -kbo	UNSAT	538	292	121	140	1 sec. 697 millisec.
PLA003-1.p		UNSAT	78	40	16	31	231 millisec.
	-lex	UNSAT	734	670	1	54	1 sec. 153 millisec.
	-mul	UNSAT	989	905	1	67	1 sec. 245 millisec.
	-kbo	UNSAT	737	673	1	54	1 sec. 158 millisec.
	-sos	UNSAT	55	31	2	31	155 millisec.
	-sos -kbo	~~SAT~~	1	1	0	11	10 millisec.
PUZ001-3.p		SAT	222	180	0	43	283 millisec.
	-lex	SAT	45	23	0	34	97 millisec.
	-mul	SAT	45	23	0	34	100 millisec.
	-kbo	SAT	45	23	0	34	99 millisec.
	-sos	SAT	100	87	0	25	152 millisec.
	-sos -kbo	SAT	3	0	0	15	11 millisec.

Tabella 1: Rilevamenti ciclo à la Otter applicato a file tptp.

File	**à la E** Opzioni	Risposta	G	D	T	S	tempo
ALG002-1.p		*SAT*	82 725	16 935	63 858	1922	*timeout*
	-lex	*SAT*	51 182	9984	38 663	2549	*timeout*
	-mul	*SAT*	50 585	9852	38 226	2521	*timeout*
	-kbo	*SAT*	29 722	22 266	28	7442	*timeout*
	-sos	UNSAT	4221	1214	2643	365	4 sec. 170 millisec.
COL123-2.p		*SAT*	51 799	4342	45 038	2253	*timeout*
	-lex	UNSAT	4087	239	3341	490	6 sec. 218 millisec.
	-mul	UNSAT	3571	2178	1203	197	4 sec. 214 millisec.
	-kbo	*SAT*	118 754	91 606	24 100	3056	*timeout*
	-sos	UNSAT	28 840	16 750	9980	2101	1 min. and 32 sec.
COM001-1.p		UNSAT	8945	7188	1500	175	4 sec. 863 millisec.
	-lex	UNSAT	862	636	152	49	826 millisec.
	-mul	UNSAT	828	606	149	49	966 millisec.
	-kbo	UNSAT	862	636	152	49	892 millisec.
	-sos	UNSAT	48	5	25	28	119 millisec.
PLA001-1.p		UNSAT	250	28	170	67	538 millisec.
	-lex	UNSAT	152	55	49	63	417 millisec.
	-mul	UNSAT	149	52	49	63	420 millisec.
	-kbo	UNSAT	152	55	49	63	415 millisec.
	-sos	UNSAT	64	16	24	39	211 millisec.
PLA003-1.p		UNSAT	53	21	18	23	141 millisec.
	-lex	UNSAT	175	141	6	29	511 millisec.
	-mul	UNSAT	232	186	8	35	683 millisec.
	-kbo	UNSAT	175	141	6	29	489 millisec.
	-sos	UNSAT	37	18	2	25	124 millisec.
PUZ001-3.p		SAT	169	131	0	37	278 millisec.
	-lex	SAT	38	20	0	30	107 millisec.
	-mul	SAT	38	20	0	30	110 millisec.
	-kbo	SAT	38	20	0	30	110 millisec.
	-sos	SAT	88	80	0	20	130 millisec.
PUZ003-1.p		UNSAT	60	18	24	19	97 millisec.
	-lex	UNSAT	50	15	17	22	98 millisec.
	-mul	UNSAT	50	15	17	22	95 millisec.
	-kbo	UNSAT	20	8	6	13	35 millisec.
	-sos	UNSAT	40	17	10	20	63 millisec.

Tabella 2: Rilevamenti ciclo à la E applicato a file tptp.

File	**à la Otter** Opzioni	Risposta	G	D	T	S	tempo
`COL123-2.p`	`-sos`	UNSAT	28 691	19 587	7026	2070	4 min. 24 sec.
	`-sos -changlee`	UNSAT	26 444	18 538	5987	1897	27 min. and 11 sec.
	`-sos -vAll`	UNSAT	29 147	20 038	7032	2068	4 min. 29 sec.
`COM001-1.p`		UNSAT	13 364	12 170	903	197	6 sec. 336 millisec.
	`-changlee`	UNSAT	12 760	11 663	820	184	35 sec. 745 millisec.
	`-vAll`	UNSAT	17 777	16 550	933	198	7 sec. 227 millisec.
`PUZ001-3.p`		SAT	222	180	0	43	283 millisec.
	`-changlee`	SAT	222	180	0	43	583 millisec.
	`-vAll`	SAT	222	180	0	43	299 millisec.

Tabella 3: Prestazioni dei due metodi di sussunzione implementati e versione sperimentale.

File	**à la Otter** Tipo	Risposta	G	D	T	S	tempo
`blocks.p`		UNSAT	13	3	5	8	23 millisec.
	`-lex`	UNSAT	5	1	0	8	7 millisec.
	`-mul`	UNSAT	5	1	0	8	9 millisec.
	`-kbo`	UNSAT	5	1	0	8	7 millisec.
	`-sos`	UNSAT	13	3	5	8	23 millisec.
`2013-01-07_es01.p`		UNSAT	49	24	10	18	124 millisec.
	`-lex`	UNSAT	10	3	0	13	18 millisec.
	`-mul`	UNSAT	10	3	0	13	19 millisec.
	`-kbo`	UNSAT	9	4	0	11	12 millisec.
	`-sos`	UNSAT	49	24	10	18	86 millisec.
File	**à la E** Tipo	Risposta	G	D	T	S	tempo
`blocks.p`		UNSAT	9	2	3	7	10 millisec.
	`-lex`	UNSAT	5	2	0	7	9 millisec.
	`-mul`	UNSAT	5	2	0	7	10 millisec.
	`-kbo`	UNSAT	5	2	0	7	9 millisec.
	`-sos`	UNSAT	9	2	3	7	9 millisec.
`2013-01-07_es01.p`		UNSAT	44	15	14	18	61 millisec.
	`-lex`	UNSAT	10	3	0	13	26 millisec.
	`-mul`	UNSAT	10	3	0	13	34 millisec.
	`-kbo`	UNSAT	9	5	0	10	23 millisec.
	`-sos`	UNSAT	44	15	14	18	62 millisec.

Tabella 4: Ciclo della clausola data applicato ad alcuni esempi fatti in classe.