Implementazione di un dimostratore di teoremi per risoluzione ordinata basato su ciclo della clausola data à la Otter e à la E Ragionamento Automatico 2012-2013

Alessandro Gottoli vr352595

6 febbraio 2013

Introduzione

Il dimostratore di teoremi per la logica del primo ordine senza simboli interpretati sviluppato in questo elaborato per il corso di ragionamento automatico è stato implementato nel linguaggio di programmazione Java per essere eseguibile su diversi sistemi operativi. Il codice è contenuto nel package thProver e si divide in cinque parti principali: Parser, Interfaccia, Clausole, Ordinamento e 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[1] (come da specifica ho implementato solo la 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 all'utente di definire opzioni come precedenze e pesi personalizzate per ogni problema per migliorare l'ordinamento sulle clausole, poi si è deciso di aggiungere la possibilità di aggiungere tali opzioni anche all'inizio di un file in formato 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. Grazie all'uso di questo generatore di parser 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`. Da questi file sono state generate automaticamente le classi java con il 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:` $precedenza_1$ `;` $precedenza_2$ `; ...;` $precedenza_n$.

Si possono inoltre definire i pesi da utilizzare nell'ordinamento di Knuth-Bendix:

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

Adesso vediamo come inserire le clausole. 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 uniformarmi alla convenzione usata in classe) e seguito da una eventuale lista di termini racchiusi tra '`(`' e '`)`'. I

[1] TPTP, `http://www.cs.miami.edu/~tptp/`

termini si scrivono in minuscolo e possono essere funzioni (se seguite dai propri argomenti che saranno ulteriori termini racchiusi tra '(' e ')'), variabili o le costanti.

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 esiste il campo *ruolo* e inserirò in sos le clausole indicate con *"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 e sarà quello che poi verrà passato alla procedura di soddisfacibilità identificando il problema in ingresso. Esso contiene diverse HashMap che servono a diversi scopi. Ci sono quelle in cui salvo i termini e gli atomi costruiti fino a quel momento, così se sono presenti più elementi che rappresentano lo stesso oggetto posso ricavarmi efficientemente un'istanza già creata e condividerla senza sprecare spazio. Poi ce nè una che mappa simbolo token con la propria arietà per controllare che non vengano passati un numero diverso di argomenti allo stesso simbolo (in caso contrario il parser avviserà l'errore di arietà). Ci sono poi anche altre strutture per salvare le precedenze e i pesi.

Quando si processano le variabili, siccome è comune utilizzare lo stesso simbolo in diverse clausole anche se l'insieme di variabili va considerato disgiunto, ho deciso di rinominarle durante la creazione assegnandogli come nuovo simbolo univoco che le identifica `simbolo_indiceclausola`. Così facendo non c'è bisogno di rinominarle ad ogni risoluzione.

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

Poi si può specificare di utilizzare la strategia dell'insieme di supporto con "`-sos`" che se sono presenti delle clausole derivate dalla negazione della congettura mette queste in `To_Select` mentre il resto delle clausole vanno direttamente in `Selected`.

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

Siccome nella maggior parte dei problemi provati non vengono specificate le precedenze sui simboli della segnatura o i pesi da dare nell'ordinamento kbo, si è deciso utilizzare di default i valori definiti da me nella classe dell'ordinamento. Se l'utente vuole definire i propri valori, può farlo digitando "`-usP`" e verranno utilizzate le precedenze e i pesi definiti nell'input, in questo caso sarà onere dell'utente verificare che pesi e precedenze rispettino le regole previste dall'ordinamento (per esempio in kbo $\exists p > 0$ t.c. $\forall var \in Variable.\ \lambda(var) = p$ o $\forall cost \in Constant.\ \lambda(cost) > \lambda(var)$).

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 (di default non è previsto un tempo limite).

In alternativa si può utilizzare una comoda *interfaccia grafica* attivabile con l'opzione "`-gui`" e verrà visualizzata una finestra in cui poter inserire direttamente l'input oppure selezionare un file da cui leggere il problema. Le opzioni, in questo caso, si possono abilitare semplicemente cliccando il menù relativo e selezionando quella desiderata. Questa interfaccia offre la possibilità di fare un solo parsing e poi eseguire diverse prove di soddisfacibilità anche con parametri diversi.

In entrambe le interfacce c'è la possibilità (se l'input risulta insoddisfacibile) di salvare un file con sintassi 'dot' in cui è specificato l'albero di refutazione per la clausola vuota trovata, specificando le clausole genitrici e le regole di inferenza utilizzate. Inoltre se sulla macchina è installato 'dot' (su linux pacchetto *graphviz*) è possibile esportare anche l'immagine dell'albero 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 da cui genero 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.

3 Clausole

Per rappresentare le varie "parti" di una formula si è strutturato il codice in questa gerarchia di classi. Le clausole sono state definite in Clause e al loro interno contengono dei puntatori ai letterali Literal che la compongono. I letterali sono composti da Atom e un boolean per rappresentare il segno e l'eventuale lista di argomenti *Term*. *Term* rappresenta un'interfaccia e viene implementata da Function, Variable e Constant. I simboli di predicato, funzione, variabile e costante sono definiti in una stringa symbol nella rispettiva classe.

La classe su cui è stato fatto più lavoro di ricerca di implementazione efficiente è Clause. Analizziamo brevemente le accortezze utilizzate. Per iniziare, i *letterali* che compongono la clausola sono inseriti sia nel campo `literals` sia in due liste (`posLits` e `negLits`) che servono per mantenere separati letterali positivi e negativi. Il motivo è che per applicare una regola di inferenza si bisogna unificare dei letterali, in questo modo posso lavorare solo su letterali con il segno che mi interessa.

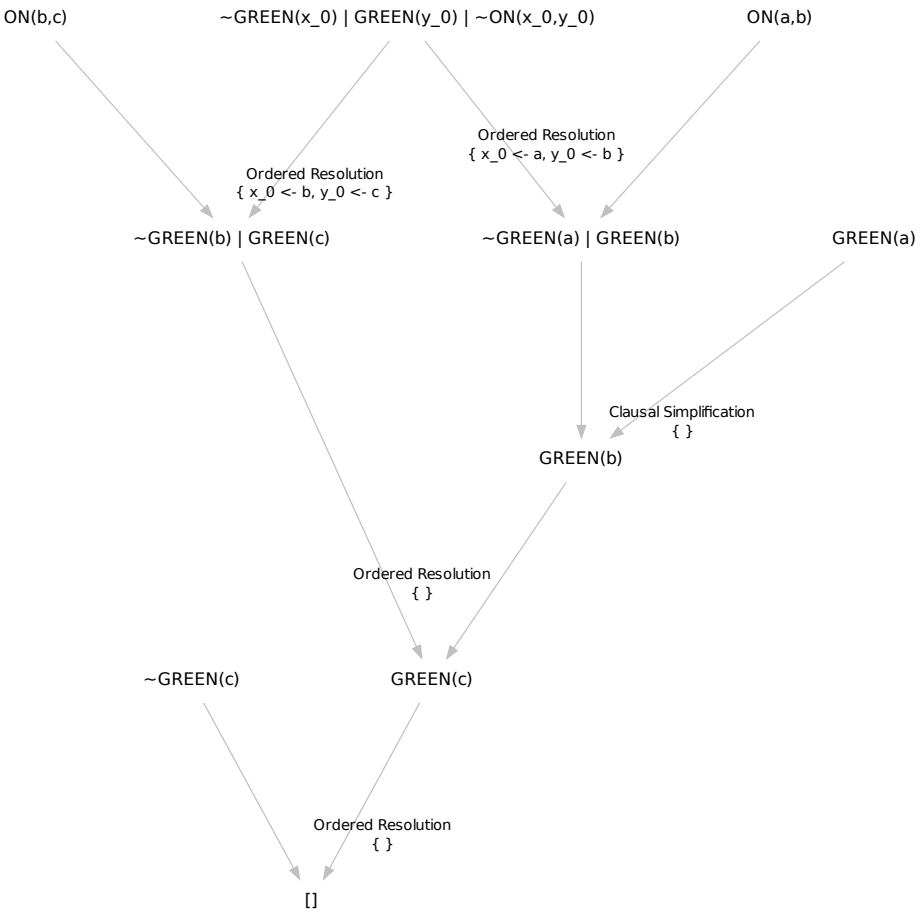

Figura 3: Albero della prova.

Durante la computazione chiamiamo molte volte metodi ricorsivi il cui risultato non cambia, quindi mi salvo il risultato per usi futuri in appositi campi. Per esempio è molto importante ordinare le clausole efficientemente quindi è stato implementato `compareTo` in modo da calcolare il "peso" delle clausole solo la prima volta contando il numero di simboli contenuti in essa che poi salvo in `numSymbs`.

Anche i letterali massimali andrebbero calcolati ad ogni applicazione di una regola di inferenza, quindi me li salvo in `maximalLiterals` e analogamente a quanto fatto per i lettarali anche in `posMaximalLits` e `negMaximalLits`.

Quando si effettua l'unificazione e si genera una nuova clausola, il metodo `mgu` ci resituisce un oggetto Substitution che contiena una HashMap<Variable, Term> che rende veloce la ricerca per sostituire la variabile nella nuova clausola generata. Nella classe si trovano anche i vari metodi per controllarne l'*idempotenza*. Quando si applica una sostituzione, le variabili rimaste vengono rinominate aggiornando l'indice della nuova clausola, ma può capitare di avere contemporaneamente `x_1` e `x_2` con stesso simbolo ma provenienti da clausole diverse, allora per disambiguarle si utilizza "'" ottenendo così `x_3` e `x'_3`. In Clause inotre sono state implementate le regole di inferenza e sono diponibili la risoluzione binaria `resolvents`, la risoluzione binaria ordinata `orderedResolvents`, fattorizzazione `factorization`, fattorizzazione ordinata `orderedFactorization`, semplificazione clausale `simplifies` e due versioni di sussunzione `subsumes` e `subsumesChangLee`. `subsumes` è la mia versione che implementa sia la *sussunzione propria* sia quella *di varianti* e confronta delle liste di letterali con lo stesso segno e simbolo di predicato cercando l'unificatore più generale per ogni letterale in queste liste, bloccando la computazione al primo mgu == null con risultato negativo oppure quando tutti i letterali della clausola corrente riescono ad unificare con risultato positivo. `subsumesChangLee` invece è l'implementazione descritta nel libro di testo del corso (Chang-Lee pag. 95) e prevede il calcolo di

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 comunque provare la versione del libro con l'opzione `-changlee` ma serve solo per test.

4 Ordinamento

Per implementare le regole di inferenza ordinate si è scelto di definire una classe Ordering che implementa i tre ordinamenti visti a lezione.

Siccome nelle api java non c'è una classe che implementi i multiinsiemi, ho definito MultiSet in cui salvo gli oggetti in un'HashMap<Object, Integer> in cui mappo il numero di occorrenze. Per poter adottare questa implementazione si è resa necessaria l'implementazione dei metodi `equals` e `hashCode` in tutte le classi.

Come accennato prima si può utilizzare la precedenza di default in cui ho definito *predicati* > *funzioni* > *costanti* e se due oggetti da ordinare sono dello stesso "tipo" risulterà maggiore il simbolo che viene prima nell'ordinamento lessicografico. Per kbo il peso di default vale 1.

Se invece scegliamo l'ordinamento definito dall'utente, può capitare che non tutti i pesi vengano inseriti, in tal caso verrà utilizzato il valore base e un avviso segnalerà la cosa.

Una "ottimizzazione" sull'ordinamento di Knuth-Bendix che è stata fatta è calcolare le occorrenze di ogni variabile direttamente quando si calcola il peso, così si esplorerà ricorsivamente i termini solo una volta.

5 Ciclo della clausola data

Il ciclo della clausola data è implementato nel metodo `satisfiable` in GivenClauseProver. L'utente può scergliere di utilizzare la versione *à la Otter* o *à la E*. Senza spiegare le differenze tra le versioni che sono state viste a lezione, analizzo solo le scelte implementative. Per estrarre la clausola data da "`To_Select`" utilizzo un ordinemento in cui scelgo quella con *minor numero di simboli* (ordinamento *equo*) e nel caso di valore uguale do precedenza alla clausola più vecchia. Per questo è stato utilizzata PriorityQueue. Se `Selected` è vuoto, estraggo la prima clausola non tautologica da `To_Select` e la metto direttamente in `Selected` senza fare un giro a vuoto. Sono stati aggiunti anche dei contatori che mi riportano il numero di clausole generate, cancellate e rimaste in `To_Select` e `Selected` per avere un'informazione completa del lavoro svolto.

6 Risultati degli esperimenti

Tutti i metodi principali sono stati testati con delle classi test su esempi fatti a mano scelti tra gli esercizi in classe. Si trovano in "./test.funzionamento.metodi/" dove è possibile ordinare i letterali dati in input, trovare i letterali massimali da un insieme di clausole, calcolare i risolventi, le clausole semplificate e sussunte.

Tutti i rilevamenti sono stati eseguiti su un notebook con processore Intel core 2 duo T7200 2.0Ghz con 2GB di RAM e Ubuntu Linux 12.04 LTS 64 bit.

Dagli esperimenti effettuati, nella maggior parte dei casi l'utilizzo di un ordinamento ha portato benefici, infatti per problemi che generano almeno 5000 clausole sono state risparmiate anche il 75% di clausole. Per problemi più piccoli il tempo necessario per calcolare i letterali massimali non viene compensato dal minor numero di clausole generate. Nessuno dei tre ordinamenti è risultato sempre la scelta migliore.

L'utilizzo dell'insieme di supporto riduce di molto il tempo di esecuzione della procedura-risultando migliore anche rispetto agli ordinamenti. Però è necessario che le clausole T che provengono dalle ipotesi formino un insieme consistente e nella realtà non sempre è vero perché sono possibili errori nella formulazione del problema.

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 dà risultato errato. Un esempio si può vedere in Figura 4. Infatti per fare risoluzione bisogna per forza unificare usando `greater_than_0(X)` che però con kbo non risulta tra

```
> 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.
```

Figura 4: Esempio di output che dimostra l'incompatibilità tra strategia dell'insieme di supporto e ordinamenti.

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 la maggior parte delle volte il tempo risparmiato a non mantenere interridotto anche `To_Selected` rende la versione à la E più performante. 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 per à la E.

Un'euristica che ho provato ad implementare nella scelta della clausola data da `To_Select` è stata la *Peak Given Ratio*, in cui ogni 4 iterazioni del ciclo, invece di prelevare la clausola in testa alla coda di priorità, seleziona la più vecchia contenuta in `To_Select`. Questa strategia non è stata soddisfacente in quanto il tempo speso per mantenere una struttura dati aggiuntiva per evitare di cercare la clausola più vecchia tra tutte si è rilevata più onerosa dei benefici. Per questo motivo non viene utilizzata.

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 gli alberi di derivazione.

Nella cartella del progetto è stato inserito anche il *javadoc* con la spiegazione dei vari metodi e delle classi.

File	**à la Otter** Opzioni	Risposta	G	D	T	S	tempo
ALG002-1.p	-limit300	*SAT*	31 959	20 178	10 973	803	*timeout*
	-lex -limit300	*SAT*	15 178	6940	7569	683	*timeout*
	-mul -limit300	*SAT*	15 156	6905	7593	672	*timeout*
	-kbo -limit300	*SAT*	46 712	37 340	28	9358	*timeout*
	-sos -limit300	UNSAT	2780	1651	963	161	7 sec. 555 millisec.
	-sos -kbo -limit300	~~SAT~~	0	0	0	14	7 millisec.
COL123-2.p	-limit300	*SAT*	13 022	2921	8973	1109	*timeout*
	-lex -limit300	UNSAT	4080	257	3316	490	36 sec. 33 millisec.
	-mul -limit300	UNSAT	3743	2627	911	212	6 sec. 823 millisec.
	-kbo -limit300	UNSAT	5887	4555	1104	235	8 sec. 664 millisec.
	-sos -limit300	UNSAT	28 691	19 587	7026	2070	4 min. 20 sec.
	-sos -kbo -limit300	~~SAT~~	501	216	174	123	1 sec. 370 millisec.
COM001-1.p	-limit300	UNSAT	13 364	12 170	903	197	7 sec. 40 millisec.
	-lex -limit300	UNSAT	1071	874	114	53	1 sec. 24 millisec.
	-mul -limit300	UNSAT	883	705	103	50	911 millisec.
	-kbo -limit300	UNSAT	2047	1810	141	61	1 sec. 513 millisec.
	-sos -limit300	UNSAT	48	5	25	28	142 millisec.
	-sos -kbo -limit300	*SAT*	20 548	13 033	6345	1181	*timeout*
PLA001-1.p	-limit300	UNSAT	272	66	145	76	1 sec. 60 millisec.
	-lex -limit300	UNSAT	216	104	44	83	581 millisec.
	-mul -limit300	UNSAT	213	101	44	83	565 millisec.
	-kbo -limit300	UNSAT	216	104	44	83	566 millisec.
	-sos -limit300	UNSAT	117	49	31	52	327 millisec.
	-sos -kbo -limit300	UNSAT	540	294	121	140	1 sec. 775 millisec.
PLA003-1.p	-limit300	UNSAT	78	40	16	31	224 millisec.
	-lex -limit300	UNSAT	734	670	1	54	1 sec. 70 millisec.
	-mul -limit300	UNSAT	989	905	1	67	1 sec. 415 millisec.
	-kbo -limit300	UNSAT	78	40	16	31	226 millisec.
	-sos -limit300	UNSAT	55	31	2	31	149 millisec.
	-sos -kbo -limit300	UNSAT	55	31	2	31	150 millisec.
PUZ001-3.p	-limit300	SAT	222	180	0	43	283 millisec.
	-lex -limit300	SAT	45	23	0	34	101 millisec.
	-mul -limit300	SAT	45	23	0	34	101 millisec.
	-kbo -limit300	SAT	132	99	0	40	226 millisec.
	-sos -limit300	SAT	100	87	0	25	151 millisec.
	-sos -kbo -limit300	SAT	12	5	0	19	29 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	-limit300	*SAT*	84 216	17 350	64 925	1931	*timeout*
	-lex -limit300	*SAT*	50 526	9856	38 167	2517	*timeout*
	-mul -limit300	*SAT*	50 400	9816	38 086	2512	*timeout*
	-kbo -limit300	*SAT*	29 610	22 182	28	7414	*timeout*
	-sos -limit300	UNSAT	4221	1214	2643	365	4 sec. 464 millisec.
COL123-2.p	-limit300	*SAT*	51 651	4333	44 907	2248	*timeout*
	-lex -limit300	UNSAT	4087	239	3341	490	6 sec. 188 millisec.
	-mul -limit300	UNSAT	3571	2178	1203	197	3 sec. 559 millisec.
	-kbo -limit300	UNSAT	5242	3576	1460	213	3 sec. 744 millisec.
	-sos -limit300	UNSAT	28 840	16 750	9980	2101	1 min. and 32 sec.
COM001-1.p	-limit300	UNSAT	8945	7188	1500	175	4 sec. 753 millisec.
	-lex -limit300	UNSAT	862	636	152	49	762 millisec.
	-mul -limit300	UNSAT	828	606	149	49	721 millisec.
	-kbo -limit300	UNSAT	1020	790	155	49	832 millisec.
	-sos -limit300	UNSAT	48	5	25	28	108 millisec.
PLA001-1.p	-limit300	UNSAT	250	28	170	67	564 millisec.
	-lex -limit300	UNSAT	152	55	49	63	402 millisec.
	-mul -limit300	UNSAT	149	52	49	63	435 millisec.
	-kbo -limit300	UNSAT	152	55	49	63	405 millisec.
	-sos -limit300	UNSAT	64	16	24	39	195 millisec.
PLA003-1.p	-limit300	UNSAT	53	21	18	23	141 millisec.
	-lex -limit300	UNSAT	175	141	6	29	488 millisec.
	-mul -limit300	UNSAT	232	186	8	35	596 millisec.
	-kbo -limit300	UNSAT	53	21	18	23	134 millisec.
	-sos -limit300	UNSAT	37	18	2	25	123 millisec.
PUZ001-3.p	-limit300	SAT	169	131	0	37	275 millisec.
	-lex -limit300	SAT	38	20	0	30	111 millisec.
	-mul -limit300	SAT	38	20	0	30	111 millisec.
	-kbo -limit300	SAT	110	82	0	35	217 millisec.
	-sos -limit300	SAT	88	80	0	20	126 millisec.
PUZ003-1.p	-limit300	UNSAT	60	18	24	19	99 millisec.
	-lex -limit300	UNSAT	50	15	17	22	97 millisec.
	-mul -limit300	UNSAT	50	15	17	22	98 millisec.
	-kbo -limit300	UNSAT	20	8	6	13	33 millisec.
	-sos -limit300	UNSAT	40	17	10	20	64 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	16 millisec.
	-lex -usP	UNSAT	5	1	0	8	9 millisec.
	-mul -usP	UNSAT	5	1	0	8	9 millisec.
	-kbo -usP	UNSAT	5	1	0	8	7 millisec.
	-sos	UNSAT	13	3	5	8	16 millisec.
2013-01-07_es01.p		UNSAT	49	24	10	18	69 millisec.
	-lex -usP	UNSAT	10	3	0	13	18 millisec.
	-mul -usP	UNSAT	10	3	0	13	25 millisec.
	-kbo -usP	UNSAT	9	4	0	11	13 millisec.
	-sos	UNSAT	49	24	10	18	89 millisec.
monkey-banana.txt		UNSAT	18	3	7	10	25 millisec.
	-lex	UNSAT	16	0	10	10	34 millisec.
	-mul	UNSAT	14	0	7	11	45 millisec.
	-kbo	UNSAT	14	6	1	9	17 millisec.
	-sos	UNSAT	18	3	7	10	44 millisec.
File	**à la E** Tipo	Risposta	G	D	T	S	tempo
blocks.p		UNSAT	9	2	3	7	10 millisec.
	-lex -usP	UNSAT	5	2	0	7	10 millisec.
	-mul -usP	UNSAT	5	2	0	7	10 millisec.
	-kbo -usP	UNSAT	5	2	0	7	8 millisec.
	-sos	UNSAT	9	2	3	7	9 millisec.
2013-01-07_es01.p		UNSAT	44	15	14	18	59 millisec.
	-lex -usP	UNSAT	10	3	0	13	29 millisec.
	-mul -usP	UNSAT	10	3	0	13	25 millisec.
	-kbo -usP	UNSAT	9	5	0	10	16 millisec.
	-sos	UNSAT	44	15	14	18	60 millisec.
monkey-banana.txt		UNSAT	15	2	6	9	19 millisec.
	-lex	UNSAT	16	0	0	10	28 millisec.
	-mul	UNSAT	14	0	7	11	36 millisec.
	-kbo	UNSAT	10	3	2	7	14 millisec.
	-sos	UNSAT	15	2	6	9	19 millisec.

Tabella 4: Ciclo della clausola data applicato ad alcuni esempi base.