PREF, un algoritmo per il calcolo delle Preferred Extensions

M.Bono, P.Bertoni

20 novembre 2013

Indice

Capitolo 1

Modellizzazione del problema

1.1 Strutture dati

Una *preferred extension* è visibile come un *insieme* di *SetArguments*, ma

Capitolo 2

Pseudocodice del programma

```
Input: Γ, C
Data: S, E*, e, O, I, E'_P
Result: E_P
(e, I) ← Grounded(Γ, C)
E_P ← {e}
if I.isEmpty then
 |  return Ep
end
Γ ← Γ ↓ I
S ← SCCSSEQ(Γ)
foreach S_i ∈ S do
 |  E'_P ← φ
 |  foreach e ∈ E_P do
 |  |  (O, I) ← boundcond(Γ, S_i, e)
 |  |  if O.isempty then
 |  |  |  if I.isempty then
 |  |  |  |  E* ← φ
 |  |  |  end
 |  |  |  else
 |  |  |  |  E* ← SATPref(Γ ↓ S_i, I ∩ C)
 |  |  |  end
 |  |  end
 |  |  else
 |  |  |  E* ← Pref(Γ ↓ (S_i\O), I ∩ C)
 |  |  end
 |  |  E'_P ← E'_P ∪ (e ⊗ E*)
 |  end
 |  E_P ← E'_P
end
return E_P
```

Algorithm 1: Algoritmo Pref.

Input: Γ, C
Data: N, ANC, ANI
Result: e, I
$e \leftarrow \emptyset$
$I \leftarrow \Gamma.A$
`/* N sono tutti e soli i nodi in C non attaccati da nodi in I */`
while $N \neq \emptyset$ **do**
  $e \leftarrow e \cup N$
  $ANC \leftarrow$sottoinsieme di C contenente tutti i nodi attaccati dai nodi in N
  $ANI \leftarrow$sottoinsieme di I contenente tutti i nodi attaccati dai nodi in N
  $C \leftarrow C \backslash (N \cup ANC)$
  $I \leftarrow I \backslash (N \cup ANI)$
end
return (e, I)

Algorithm 2: Algoritmo Grounded.

Input: Γ, S_i, e
Result: O, I
$O \leftarrow \{n |\ n \in S_i, \quad n \text{ is attacked by } m \in e\}$
$temp \leftarrow \{n |\ n \in \Gamma \setminus (S_i \cup e), \quad n \text{ is attacked by } m \in e\}$
$I \leftarrow \{n |\ n \in S_i \setminus O,$

$$(\nexists n \in I : n \text{ is attacked by } m \in \Gamma \setminus S_i) \ \oplus$$
$$(\nexists n \in I : n \text{ is attacked by } m \notin temp)$$

return (O, I)

Algorithm 3: Algoritmo Boundcond.

Capitolo 3

Ottimizzazione del codice

3.1 Miglioramenti suggeriti

È stata implementata una serie di regole utili ad evitare delle chiamate alle procedure "pesanti" usate dall'algoritmo *Pref*, che sono qui elencate. Il codice utilizzato è comunque pseudocodice e non standard C_{++}. Non dovrebbe esser difficile però individuarlo nel sorgente.

3.1.1 Sulla chiamata a Grounded e Pref

La regola espressa informalmente è: *la chiamata a una di queste due procedure produce risultati noti a priori quando il primo argomento è uguale al secondo e coincide con un grafo di un solo nodo. In questo caso, l'insieme restituito è composto da un singoletto contenente quell'unico nodo.*

```
if Γ = C ∩ Γ.cardinality= 1 then
    e ← Γ
    I ← ∅
end
else
    Grounded(Γ, C, e, I)
end
```

Algorithm 4: Miglioramento nella gestione della chiamata a Grounded.

```
A ← Γ ↓ S_i\O
B ← I ∩ C
if A = B ∩ A.cardinality= 1 then
|  E* ← {A}
end
else
|  E* ←Pref(A, B)
end
```

Algorithm 5: Miglioramento nella gestione della chiamata a Pref.

3.1.2 Sulla chiamata a PrefSAT

La regola espressa informalmente è: *la chiamata alla procedura produce risultati noti a priori quando il primo argomento è uguale al secondo e coincide con un componente fortemente connesso (*SCC*) di cardinalità 1 o 2. Nel primo caso, l'insieme restituito è composto da un singoletto contenente proprio l'*SCC*, nel secondo caso è composto da due singoletti, contenenti ciascuno uno dei due nodi dell'*SCC.

Nel codice si troveranno istruzioni leggermente diverse dalle seguenti, ma chiaramente equivalenti, per sfruttare meglio caching e lazy evaluation.

```
A ← Γ ↓ S_i
B ← I ∩ C
if A = B ∩ S_i.cardinality= 2 then
|  {S'_i, S''_i} ← S_i.reduceSingletons()
|  E* ← {S'_i, S''_i}
end
else
|  E* ←PrefSAT(A, B)
end
```

Algorithm 6: Miglioramento nella gestione della chiamata a PrefSAT.

Durante il testing però la prima parte di questa regola non si è rivelata coerente con l'output di *PrefSAT* e quindi generalmente ha prodotto risultati non corretti. Si è notato in particolare che talvolta *PrefSAT* veramente si comporta come qui specificato, talvolta restituisce un insieme vuoto. Prudenzialmente quindi si è ristretta la miglioria al solo caso ove la cardinalità è 2. Per giustificare quanto detto, si riportano qui gli output ottenuti a questo proposito, in due dei tre esempi forniti nel documento delle specifiche. Si spera che questo aiuti chi è più esperto a identificare casi degeneri della regola espressa.

3.1.2.1 caso: esempio 1

PrefSAT viene chiamato due volte in totale, per due e differenti.

$$\Gamma \downarrow S_i = I \cap C = \{n9\}$$
$$S_i = \{n9\}$$
$$e = \{n1\ n3\ n6\ n13\}$$
$$E^*_{suggested} = \{\{n9\}\}$$
$$E^*_{returned} = \{\{\}\}$$

il medesimo risultato si ottiene nella seconda chiamata, ove cambia solo

$$e = \{n1\ n3\ n6\ n14\}$$

In definitiva *Pref* restituisce

$$\{\{n1\ n3\ n6\ n9\ n13\}\ \ \{n1\ n3\ n6\ n9\ n14\}\}$$

invece che

$$\{\{n1\ n3\ n6\ n13\}\ \ \{n1\ n3\ n6\ n14\}\}$$

come previsto dall'oracolo (si faccia riferimento a 4.2).

3.1.2.2 caso: esempio 3

PrefSAT viene chiamato una volta in totale.

$$\Gamma \downarrow S_i = I \cap C = \{n3\}$$
$$S_i = \{n3\}$$
$$e = \{n1\ n4\}$$
$$E^*_{suggested} = \{\{n3\}\}$$
$$E^*_{returned} = \{\{n3\}\}$$

Pref restituisce $\{\{n1\ n2\ n3\ n4\}\}$ come previsto.

3.1.3 Sulla chiamata a Boundcond

La regola espressa informalmente è: *sia Pref(Γ, C), ove $\Gamma = C$. La chiamata a Boundcond per ogni SCC senza alcun SCC padre, ovvero non attaccato da altri SCC, restituirà necessariamente come O un insieme vuoto e come I l'SCC stesso, per qualunque e.*

```
if Γ = C ∩ S_i .fathers= ∅ then
    O ← ∅
    I ← S_i
end
else
    (O, I) ←Boundcond(Γ, S_i, e)
end
```

Algorithm 7: Miglioramento nella gestione della chiamata a Boundcond.

3.2 Miglioramenti personali

3.2.1 Gestione dei padri di S_i

Si definisce $\alpha_i \triangleq \bigcup \beta_i$, dove β_i è un SCC S_j che attacca S_i. Formalmente dunque α_i è un unione di *SetArguments* e quindi un *SetArguments* a sua volta, ma se si osserva che $S_i \cap S_j \ \forall i \neq j$ (ovvero gli SCC sono insiemi disgiunti di *Arguments*), è possibile considerare α_i come un semplice insieme generico. Per semplicità è stato quindi considerato un **std::vector**.

MAX SEI PREGATO DI INSERIRE QUA QUALUNQUE MIGLIORAMENTO ALGORITMICO/STRUTTURA DATI. CHE TI VENGA IN MENTE. TIENI NEL CAPITOLO "STRUTTURE DATI" SOLO UNA VERSIONE DI "PARTENZA", COSÌ SEMBRA ABBIAM FATTO PIÙ ROBA :D

Capitolo 4

Analisi computazionale

4.1 Individuazione dei blocchi procedurali d'interesse

La complessità temporale di **Pref** dipende fortemente, oltre che dalle dimensioni del problema, dal numero di chiamate ai seguenti algoritmi:

- *Grounded*
- *SCCSEQ*
- *Boundcond*
- *PrefSAT*
- *Pref* stesso.

L'analisi quindi è stata strutturata per fornire due indici:

- η - una misura del tempo "*tic-toc*" per ciascuna chiamata a una delle suddette procedure, più il tempo totale di esecuzione del programma;
- ξ - il numero di chiamate a ciascuna procedura, nell'arco dell'intera esecuzione.

Mentre il primo indice è atto a relazione la dimensione del problema al tempo in sè, ed è fortemente mediato dalle prestazioni della macchina di esecuzione, il secondo è una misura molto buona delle differenze tra la versione *base* di **Pref** e la versione *improved*, implementazione dello studio presentato nel capitolo 3. Chiamando il programma sullo

stesso problema nelle due modalità, infatti, è rapidamente visibile il risparmio in termini di chiamate introdotto dalle ottimizzazioni. Si noti che *SCCSEQ* non è stato introdotto nell'indice ξ siccome non è sottoposto ad ottimizzazioni, pertanto il numero delle sue chiamate è sempre pari a quello relativo a *Pref*.

Entrambi gli indici sono sempre restituiti dal programma sullo standard output.

4.2 Verifica di correttezza: gli oracoli

Per testare innanzitutto i requisiti funzionali ci si è avvalsi dell'oracolo *ASPARTIX* disponibile come web service dall'*URL* `http://rull.dbai.tuwien.ac.at:8080/ASPARTIX`. Si noti che la rete di Facoltà dovrebbe bloccare la porta 8080. L'applicazione oracolo, oltre a confermare gli output dei tre esempi forniti insieme alle specifiche, ha validato anche altri test, generati secondo il metodo descritto in 4.5.

4.3 Metodo di misura del tempo

La scelta è ricaduta sulla libreria **chrono** del C_{++}, che costituisce anche un *subnamespace std::chrono*. I concetti principali implementati sono tre:

- le *durate temporali* misurano lassi di tempo, fino all'ordine dei millisecondi. In questa libreria sono rappresentate come oggetti della classe template *duration*, che accoppiano una rappresentazione di conto e una precisione di periodo (*e.g.*, 10 millisecondi ha 10 come rappresentazione di conto e 'millisecondi' come precisione di periodo.)
- gli *istanti temporali* sono riferimenti a dei punti precisi sull'asse temporale, espressi come una *duration* da un punto fissato per convenzione.
- un *clock* è un framework collegante un *istante temporale* a un'istante del mondo reale.

La libreria offre almeno tre clocks diversi per esprimere il tempo corrente come un valore numerico: *system_ clock*, *steady_ clock* e *high_ resolution_ clock*, a miglior precisione e che sarà per questo utilizzato nelle benchmark.

Si noti che per utilizzare **chrono** è obbligatoria una compilazione conforme allo standard $C_{++}11$.

4.4 Configurazione del calcolatore

Le benchmarks sono state eseguite su un notebook così caratterizzato:

- **OS** Linux Ubuntu 10.04 con istruzioni a 32 bit
- **CPU** Intel Pentium(R) Dual-Core CPU T4200 , clock 2.00GHz e tecnologia *Hyper Threading*
- **RAM** 4GB DDR3, di cui 3 disponibili all'utente.
- **Compiler** Gnu C Compiler (gcc) in versione $C^{++}11$ e ottimizzato.

4.5 Un generatore di problemi

Le tre istanze di prova allegate alle specifiche del problema vengono risolte in un tempo dell'ordine dei 10 millisecondi, con una varianza elevata, e quindi non sono utilizzabili per una profilazione consistente della complessità temporale. Per generare grafi più onerosi è stato implementato un piccolo applicativo indipendente, che libera l'analista dallo sforzo di pensare e codificare argumentation framework anche di migliaia di nodi o archi. I parametri richiesti sono:

- numero N di *Arguments*, ovvero nodi del grafo. Verranno automaticamente denominati n_i, $i \in [1, N]$.
- il numero massimo $\overline{A}$ di *Arguments* che attaccano o sono attaccati da ogni *Argument*. Ricordando che un grafo orientato di N nodi completamente magliato è dotato di $N(N-1)$ archi, il valore $\frac{N(N-1)}{N} = N - 1$ costituisce un upperbound per $\overline{A}$. Un grafo denso "a metà" è chiaramente caratterizzato da un $\overline{A} = \frac{N}{2}$.
- nome del file in formato *dl* dove il *framework* verrà salvato.
- $\{graph, dot, nograph\}$ per indicare se si desidera anche un output grafico in formato jpg, rappresentante il grafo. È qui utilizzato, come nella documentazione del sorgente scritta con **Doxygen**, il linguaggio **dot**. Nel caso "dot" verrà scritto solo il file dot senza generar l'immagine.

Applicativo **graphGenerator** e relativo sorgente vengono allegati al progetto finale. Vengono inoltre forniti degli esempi casuali generati con esso, utilizzati nei test che saranno presentati a breve. I loro nomi rispettano la sintassi $N\_\overline{A}trial.dl$, dove trial è una lettera $a, b, \ldots n$ che distingua i problemi di uguale dimensione generati. È stato deciso inizialmente di esplorare gli ordini di grandezza di N da 10 a 10^4, ma, come si vedrà, già il passaggio da 10^2 a 10^3 si mostrerà proibitivo.

Se si suppone una complessità $O(1)$ per elaborare ciascun nodo, esclusa dalla chiamata la gravosissima elaborazione dell'immagine, la complessità di graphGenerator è chiaramente $O(N\overline{A})$. Lo spazio richiesto per la memorizzazione dei grafi si comporta anch'esso linearmente rispetto alle singole dimensioni $N, \overline{A}$, come suggerito in figura 4.1.

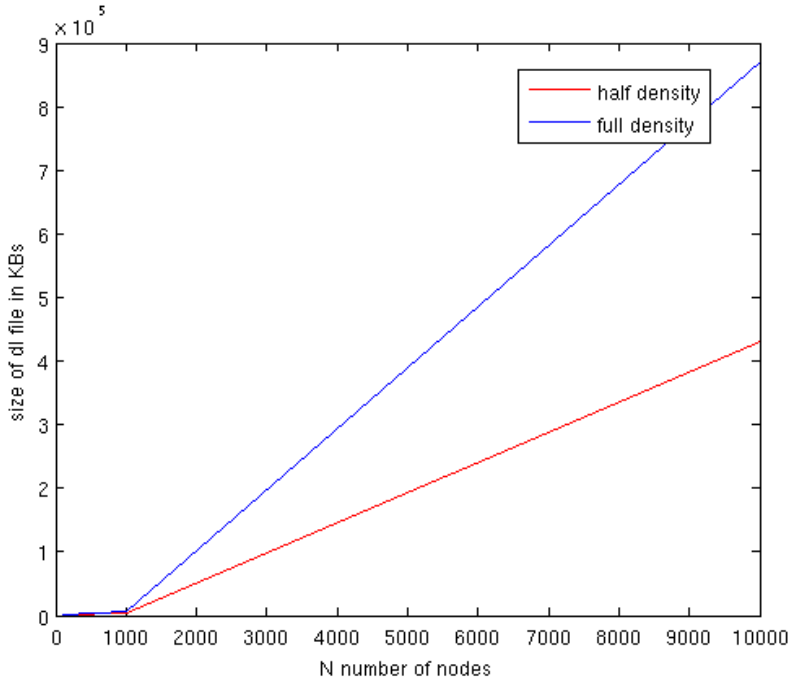

Figura 4.1: Dimensione su disco delle istanze del problema, al variare di N (ascisse) e con due $\overline{A}$ fissate relativamente a N.

4.6 Benchmarks sui test generati

Note: un trial contrassegnato dal "+" rappresenta l'esecuzione migliorata su quel grafo. Nella parte destra della tabella, per non appesantire la lettura sono stati omessi i valori invariati tra ciascuna coppia di esecuzioni. I tempi sono espressi in millisecondi.

4.7 Considerazioni finali

In tabella sono stati riportati solo i tempi totali e non alle singole chiamate, ma si registra che il tempo utilizzato da *PrefSAT* domina fino al 90% della durata totale di ogni esecuzione. È altresì la più onerosa in termini di memoria principale allocata, e siccome è codice scritto da terzi si sospetta che presenti varie memory leak. Si può dunque affermare che l'ottimizzazione più rilevante di ogni altra sia quella che eviti la chiamata al problema SAT, peraltro notorialmente *NP-hard*.

Moltissime volte, mentre *Pref* gira in versione migliorata, viene risparmiata la chiamata a *Boundcond* poichè il relativo *SCC* non ha nessun padre. Questo è possibile che sia legato alla generazione puramente pseudocasuale dei grafi, che non cerca di costruire nessun percorso arbitrariamente lungo sui nodi, ma si limita a computare un certo numero di probabilità di collegamento. Versioni migliorate della struttura di test potrebbero incorporare un'euristica della "connessione" del grafo generato; il trascurabile vantaggio del miglioramento su *Boundcond* non ne giustifica per ora lo sforzo.

Si noti inoltre la grande differenza nel numero di chiamate, tra tre istanze note come *esempio1*, *esempio2* e *esempio3*, e invece i grafi generati casualmente. È evidente che argumentation framework strutturati senza una logica apparente rimangono alquanto sconnessi e, oltre a non avere interesse reale nella loro risoluzione, ne hanno ben poco dal punto di vista dell'indice ξ. Qualora fossero disponibili istanze di grafi di dimensioni considerevoli e ben strutturati, saremo lieti di incorporarli in una seconda e più informativa versione del testing.

Parimenti si può dire che le istanze generate casualmente, dato il numero sempre minimo di chiamate alle varie procedure, costituiscono una sorta di *lower bound* sulle prestazioni di *Pref* in generale.

N	$\overline{A}$	*trial*	*total time*	*#Grounded*	*#Boundcond*	*#PrefSAT*	*#Pref*
esempio		1	4	1	5	5	1
		1^+			2	4	
		2	1	1	1	1	1
		2^+			0	0	
		3	3	1	3	3	1
		3^+				2	
50	25	a	100	1	1	1	1
		a^+	95				
		b					
		b^+					
		c					
		c^+					
50	49	a	63000	1	1	1	1
		a^+					
		b					
		b^+					
		c					
		c^+					
100	50	a	900	1	1	1	1
		a^+	850		0		
		b	1150	1	1	1	1
		b^+	1160		0		
		c	550	1	1	1	1
		c^+	560		0		
100	99	a	6000	1	1	1	1
		a^+	6000		0		
		b	4150	1	1	1	1
		b^+	4150		0		
		c	4700	1	1	1	1
		c^+	4700		0		
200	100	a	63000	1	1	1	1
		a^+	60000		0		
		b	34000	1	1	1	1
		b^+	33000		0		
		c	17900	1	1	1	1
		c^+	17800		0		

Capitolo 5

Varie ed eventuali

5.1 Documentazione del codice

La documentazione del codice *SemOPT* era pressochè assente ed è stata largamente migliorata, basandosi sullo studio del codice. Sia per quella che per quella relativa a *Pref* è stato utilizzato **Doxygen per generare del codice HTML navigabile, e il linguaggio dot per aggiungervi dei grafi. Si rimanda all'html allegato alla Documentazione, ed eventualmente al codice sorgente.**