

Implementazione Algoritmo di Chiusura di Congruenza Nelson-Oppen

Federico De Meo

11 giugno 2011

Introduzione

Questo documento descrive l'implementazione della procedura di Nelson-Oppen descritta a lezione per il calcolo della soddisfacibilità di formule scritte nell'unione delle signature della teoria dell'uguaglianza e di quella delle liste. Oltre alla semplice implementazione dell'algoritmo proposto dal libro sono state introdotte delle euristiche che hanno portato a notevoli migliorie prestazionali che saranno esaminate nel seguito. Il linguaggio di programmazione scelto è Java, il quale offre un variegato insieme di strutture dati molto utili e ben implementate. Si farà anche un confronto con l'algoritmo proposto da Downey, Sethi e Tarjan implementato da Marco Tamassia.

1 Parser

L'implementazione del progetto parte con la stesura del Parser con lo scopo di leggere le formule nella teoria dell'uguaglianza (T_E) e nella teoria delle liste (T_{cons}). A tal scopo si è scelto di utilizzare il compilatore *JavaCC* il quale, definita una grammatica context-free BNF, genera le classi necessarie al riconoscimento delle formule in input. La peculiarità di *JavaCC* è quella di generare insieme un analizzatore lessicale e uno sintattico partendo semplicemente dal file che definisce la grammatica. La grammatica scelta che descrive la sintassi delle formule accettate è la seguente:

```
Formula ::= Clausola (& Clausola)*
Clausola ::= Token = Token | Token != Token | atom(Token) | -atom(Token)
Token ::= cons(Token, Token) | car(Token) | cdr(Token) | Token | Fun(Token)
Token ::= [a-z]+[0-9]*
Fun ::= [A-Z][a-z0-9]*
```

Come riportato dal testo, ci concentriamo solo sul frammento senza quantificatori. Questa grammatica è stata riportata nel file `grammar.jj` alla quale è stata aggiunto codice Java che definisce la semantica per la costruzione del DAG su cui opera l'algoritmo. Inoltre si è scelto di generare un errore qualora nella formula ci siano due funzioni con stesso nome e arietà diversa.

2 Scelte implementative

Oltre alla possibilità di utilizzare il progetto tramite la riga di comando si è scelto di implementare una piccola interfaccia grafica al fine di agevolare la verificabilità dell'uso dell'applicazione. Si è principalmente adoperata la riga di comando per la generazione automatizzata di test prestazionali. La classe principale è `verifica/Main.java` che avvia l'una o l'altra modalità. Nello stesso file è stato anche implementato l'algoritmo di Chiusura di Congruenza di Nelson-Oppen nelle due versioni semplice e con euristiche. Per separare le due implementazioni si è scelto di riscrivere ogni metodo differenziandolo dall'analogo nell'altra implementazione. In particolare la signature dei metodi prevede che:

- i metodi che fanno parte dell'implementazione con euristiche terminino tutti con il carattere H (e.g.: `mergeH(...)`)
- i metodi che fanno parte dell'implementazione semplice inizino tutti con il carattere _ (e.d.: `_merge(...)`)

L'avvio dell'algoritmo di chiusura di congruenza è dato dalla chiamata a `execCC(String formula, JResult window)` dove `formula` è la formula di cui verificare la soddisfacibilità e `window` è un oggetto `JResult` che rappresenta la finestra che visualizzerà l'output dell'algoritmo (parametro usato solo nell'invocazione con GUI).

2.1 Strutture dati di supporto

Nella prima fase di esecuzione viene chiamato il Parser, al quale vengono passate, oltre alla stringa da parserizzare, diverse strutture dati di supporto:

- Graph: `HashMap` che mantiene la struttura del grafo;
- equals: `ArrayList` che mantiene le chiavi dei letterali uguali nella formula;
- noEquals: `ArrayList` che mantiene le chiavi dei letterali disuguali nella formula;
- consList: `ArrayList` che mantiene le chiavi dei `car()` e `cdr()` degli operatori `cons()` di cui fare la prima serie di MERGE;

- `atoms`: `ArrayList` che mantiene le chiavi degli `atom()`;
- `consfn`: `ArrayList` che mantiene le chiavi dei `cons()` da confrontare con le chiavi degli `atoms()`;

Si noti una particolarità: le strutture `equals`, `noEquals` e `consList` rappresentano logicamente coppie di chiavi che devono essere usate per effettuare le `MERGE` ma la struttura `ArrayList` non gestisce coppie ma bensì singoli elementi in successione. Si è ugualmente scelta questa struttura avendo cura di riempire gli `ArrayList` in modo che un elemento in posizione pari fosse in relazione con il successivo elemento dispari. ($2i \sim 2i + 1$)

2.2 Chiavi

Per meglio identificare ogni elemento all'interno delle strutture dati di appoggio si è scelto di utilizzare una stringa come chiave univoca. I caratteri che compongono la stringa sono li stessi che identificano un termine all'interno della formula. Ad esempio:

$$F : car(x) = cdr(y) \wedge x = y$$

genererà un grafo con 4 nodi le cui chiavi saranno le stringhe: `car(x)`, `cdr(y)`, `x`, `y`, ognuna delle quali identifica un nodo.

Questo consente una facile gestione logica dei nodi unito ad una notevole velocità di accesso ai nodi stessi tramite l'uso della struttura `HashMap`.

2.3 Algoritmo

Dopo la fase di parsing viene chiamato uno dei due metodi `NelsonOppenSpeedUp(String formula)` o `NelsonOppen(String formula)` che rispettivamente eseguono l'algoritmo con euristiche e senza euristiche. Non mi soffermerò sulla specifica dell'algoritmo privo di euristiche in quanto ampiamente descritto sul testo di riferimento nel Capitolo 9 (pag. 251-258).

2.4 Euristiche

Al fine di migliorare le prestazioni dell'algoritmo di Nelson-Oppen, sono state introdotte diverse euristiche qui riportate in dettaglio:

- **Compressione dei cammini:** l'algoritmo di Nelson-Oppen utilizza gli insiemi disgiunti per rappresentare le classi di congruenza che dovranno essere unite. Ogni elemento di un insieme disgiunto ha un campo `find` che lo unisce in catena al rappresentante della classe a cui appartiene. La compressione dei cammini è un'euristica che fa in modo che il percorso da un nodo al suo rappresentante sia diretto, collegando direttamente ogni nodo di una classe con il suo rappresentante. Per fare ciò, ogni qualvolta viene chiamato il metodo `find(id)` ricorsivamente si sale nella catena e quando lo stack viene disfatto all'indietro vengono modificati tutti i campi `find` dei nodi visitati aggiornandoli con l'id del rappresentante della classe.
- **Unione per rango:** anche questa euristica va a migliorare la struttura per insiemi disgiunti e in particolare definisce un criterio con il quale scegliere il nuovo rappresentante di una classe in fase di `union`, criterio non definito nell'algoritmo presente sul testo dove la scelta del nuovo rappresentante è del tutto arbitraria. Con l'unione per rango viene scelto come nuovo rappresentante la radice del sottoalbero con più nodi, a cui unire la radice del sottoalbero con meno nodi. Per semplificare il mantenimento del numero di nodi di ogni sottoalbero è stato introdotto un nuovo campo nell'oggetto `Node` chiamato `rank`. Inizialmente inizializzato a 0 per ogni nodo mantiene un limite superiore all'altezza del nodo, la radice con il rango più piccolo viene fatta puntare alla radice con rango maggiore. Così facendo avremo un minor numero di nodi ad avere una più lunga catene di `find` prima di raggiungere il nuovo rappresentante.
- **Letterali vietati:** quest'ultima euristica non riguarda gli insiemi disgiunti ma aiuta a terminare la computazione di una formula non soddisfacibile appena si scopre una contraddizione. L'idea è quella di scoprire una contraddizione in fase di `merge` di due classi, terminando subito. Per fare questo ogni `Node` è stato arricchito con una struttura `HashSet` che mantiene l'insieme dei termini proibiti (*forbidden*) per quel nodo. La lista viene arricchita in fase di parsing, ogni qualvolta si incontra una disuguaglianza $F : ...s_{m+1} \neq t_{m+1}...$, viene arricchito l'insieme *forbidden* di s_{m+1} aggiungendo t_{m+1} e analogamente si popolano i *forbidden* di t_{m+1} aggiungendo s_{m+1} . Con questi due insiemi popolati restano da fare alcuni semplici controlli prima di effettuare una `merge(id1, id2)`. Il primo controllo consiste nel verificare che nella lista *forbidden* del rappresentante di `id1` non sia presente il nodo `id2`. Vista la natura della struttura dati `HashSet` questo controllo viene fatto in tempo atteso $O(1)$. Gli altri due controlli verificano in quale classe di congruenza si trovano i *forbidden* di `id1` e di `id2`. In particolare se il rappresentante di un *forbidden* di `id1` è uguale al rappresentante di `id2` (e viceversa) allora avrò una contraddizione e termino la computazione restituendo insoddisfacibile. Se invece questi controlli danno esito negativo, ovvero non sono ci sono problemi nel fare la `merge` delle due classi, allora si procede con la fase di unione all'interno della quale l'algoritmo si preoccupa di propagare la lista *forbidden* al nuovo rappresentante della classe mantenendo l'insieme sempre aggiornato.

Le prime due euristiche sono applicabili al caso generale di formula soddisfacibile o non soddisfacibile migliorando molto l'algoritmo semplice di Nelson-Oppen. L'ultima euristica aiuta solo in caso di formula insoddisfacibile abbattendo il tempo di computazione anche sotto all'implementazione di Downey, Sethi e Tarjan la quale non si presta a questo genere di euristica.

2.5 Generatore di formule

Al fine di ottenere benchmark considerevoli, si è implementato un piccolo generatore di formule casuali. Questo generatore è accessibile solo dall'interfaccia grafica e consente di generare una qualsiasi formula casuale che rispetti il numero di uguaglianze, disuguaglianze, atomi e non atomi impostati dall'utente. La probabilità di generare formule insoddisfacibili è molto alta ma specificando solo clausole di uguaglianza, o un numero considerevole di formule di uguaglianza e un numero molto inferiore di disuguaglianze, è possibile generare formule soddisfacenti.

3 Test

Nel seguito sono riportati i risultati dei test effettuati sulle medesime formule con le tre diverse implementazioni: Nelson-Oppen (NO), Nelson-Oppen con euristiche (NO+) e Downey, Sethi e Tarjan (DST) quest'ultimo implementato da Marco Tamassia. I test sono stati effettuati su due macchine distinte aventi:

1. processore i5, frequenza di 2,4GHz con sistema operativo MacOS X e quindi con una JVM v1.6.0 proprietaria Apple
2. processore i7 a 2,4GHz con sistema operativo Ubuntu 11.04 con JVM v1.6.0 prodotta da Sun/Oracle

I medesimi risultati su entrambe le macchine indicano che l'architettura di contorno non ha apportato contributi.
GRAFICI E TABELLE

4 Conclusioni

L'algoritmo di base proposto da Nelson-Oppen presenta un andamento estremamente altalenante nonostante la sua complessità polinomiale $O(n^2)$, la medesima versione con euristiche ha portato ad una stabilità quasi inaspettata trattandosi di miglione che non abbassano il grado di complessità dell'algoritmo di partenza. L'algoritmo DST presenta invece una maggior stabilità in tutti i test effettuati e quindi è da preferire a quello proposto da Nelson-Oppen anche nel caso la probabilità di insoddisfacibilità della formula sia elevata. L euristica dei nodi vietati su Nelson-Oppen consente di terminare prima di DST ma a conti fatti il tempo risparmiato non è sufficiente da renderlo una valida alternativa.

Riferimenti bibliografici

- [1] T. Norvell - JavaCC tutorial - www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - McGraw-Hill - *Introduzione agli algoritmi e alle strutture dati*, Strutture dati per insiemi disgiunti, pagine 427-236
- [3] Aaron R. Bradley, Zohar Manna - Springer - *The calculus of computation* Capitolo 9
- [4] P. J. Downey, R. Sethi, R. E. Tarjan - Variations on the common subexpression problem - 1980
- [5] D. Cyrluk, P. Lincoln, N. Shankar - On Shostak's decision procedure for combinations of theories - 1996