Implementazione della procedura di soddisfacibilità per la Teoria delle Liste

Alessandro Gottoli vr352595

16 giugno 2011

Introduzione

La procedura di soddisfacibilità per la teoria delle liste di questo progetto è stato implementato nel linguaggio di programmazione Java per essere eseguibile su diversi sistemi operativi ed è stato testato su Windows e Linux. Il progetto si divide in tre: interfaccia, parser, chiusura di congruenza.

1 Interfaccia

L'inserimento della formula avviene tramite un interfaccia grafica e si può digitare direttamente nell'area di testo oppure aprire comodamente da file. La stringa contenuta nel file viene visualizzata, dando modo all'utente di modificarla e salvarla in un altro file (con "alert" in caso si tentasse di sovrascrivere un file esistente). Però in formule molto lunghe questa operazione appesantirebbe troppo l'interfaccia, quindi se la dimensione è maggiore di 50000 caratteri, si indica solamente il percorso del file caricato. Tutti i pulsanti vengono abilitati solo all'occorrenza, limitando al minimo i possibili errori degli utenti. Per esempio; il pulsante "Start" che avvia l'analisi non è attivo se la formula è vuota; oppure finché l'algoritmo è in esecuzione, la barra dei menu risulta non attive e l'area di testo non modificabile, costringendo l'utente ad attendere la fine dell'algoritmo o a terminarla con il pulsante "Abort" che appare quando inizia l'algoritmo.

Per apprezzare l'interfaccia grafica occorre utilizzare formule corpose, così si noterà che fase dell'algoritmo è in esecuzione. Durante la fase di parsing verrà visualizzata una barra di caricamento che indica la progressione con la percentuale elaborata. Per l'algoritmo di chiusura di congruenza non è possibile stimare il tempo necessario, quindi si usa una barra "indeterminata" animata per indicare che il programma sta elaborando. Al completamento di ogni fase verrà visualizzato il tempo impiegato. Passiamo alle fasi più importanti.

2 Parser

Una volta inserita la formula della quale vogliamo calcolare la soddisfacibilità, questa stringa viene passata ad un parser che la analizza carattere per carattere effettuando contemporaneamente analisi sintattica e costruzione del grafo.

2.1 Analisi sintattica

L'analisi sintattica controlla che la formula sia stata scritta rispettando le seguenti regole sintattiche:

elemento: nome senza spazi. Esempio: elem.

funzione: il nome rispetta la regola per l'elemento, i parametri sono a loro volta funzioni o elementi e sono racchiusi tra parentesi tonde '(' e ')' e separati ',' o ';'. Esempio: fun(el, g(a)).

uguaglianze/disuguaglianze: sono costituite da funzioni o elementi separati da '='/'!='. Esempio: c = fun(b), a != fun(b).

atom: viene trattato come una funzione con un solo parametro e per indicare la sua negazione si fa precedere da '!'.

clausole: sono uguaglianze, disuguaglianze, atom o !atom separati da virgole ',' o punti e virgole ';'. Esempio: a = f(b); !atom(g(a,b)).

Gli spazi vengono tutti ignorati tranne quelli presenti nel nome di una funzione o di un elemento (che restituiscono errore sintattico) per dare più flessibilità all'utente di scrivere le formule nel formato che preferisce.

2.2 Costruzione del grafo

La costruzione del grafo istanzia degli oggetti "nodi" ed ognuno rappresenta una funzione o un elemento riconosciuto durante il parsing. Siccome nella stringa un elemento può comparire più di una volta, è importante non duplicare il nodo corrispondente a quell'elemento. Controllare se un nodo esisste già può essere costoso quindi serve una struttura dati che permetta estrarre (se c'è) il nodo in questione nel minor tempo possibile. Per questo motivo si è scelto di usare una *tabella hash* ad *indirizzamento aperto* in caso di collisioni. La scelta è dovuta al fatto che se si verifica una collisione possiamo utilizzare il *doppio hashing* con esplorazione della tabella in modo diverso per ogni chiave, velocizzando l'estrazione. Rispetto al metodo del concatenamento, inoltre non usa liste per memorizzare i nodi che collidono quindi non si consumerà ulteriore memoria. Caratteristica fondamentale è che i nodi non vengono mai eliminati e quindi cancellati dalla tabella e questa è proprio la situazione ideale per questo tipo di tabella.

In fase di costruzione, ogni nodo 'funzione' viene collegato tramite il campo '`param`' ai nodi che sono suoi parametri e siccome sui parametri non ci sono operazione di unione si è scelto di usare un array visto che sappiamo esattamente l'arietà di una funzione. Dualmente, ai nodi argomenti viene aggiunto il nodo funzione al campo '`ccpar`' che rappresenta i suoi padri. Qui però l'uso dell'array è improponibile perché questo campo è soggetto a frequenti unioni, quindi si è dovuti ricorrere ad un altra struttura dati diversa come vedremo successivamente. Poi è presente anche il campo '`fn`' che identifica il nome della funzione del nodo.

Per gestire il fatto che un nodo deve essere atomo o lista sono stati inseriti i booleani '`atom`' e '`cons`' che se messi a true indicano rispettivamente se un nodo è atomo o !atom. Anche se sembra che siano rindondanti, perché una situazione esclude l'altra, si sono usati due campi distinti perché se sono messi entrambi a

false significa che non si sa di che tipo sia il nodo. Questo ci permette di effettuare la prima ottimizzazione riconoscendo come insoddisfacibili le stringhe che contengono un elemento 'cons' al posto di 'atom' direttamente durante il parsing, per esempio car(cons(x,y)) oppure cons(cons(a,b),b). Poi il campo '`find`' punta al rappresentante della classe di congruenza a cui appartiene il nodo (all'inizio ogni nodo appartiene ad una classe differente, quindi punterà a se stesso), in questo modo ogni classe è rappresentata come un albero e le varie classi come una foresta. Per ottimizzazione si è infine aggiunto il campo '`diversi`' che contiene tutti i nodi che non possono appartenere alla stessa classe del nodo e anche qui è importante scegliere una buona struttura dati.

Assieme al grafo vengono poi costruite due code fifo che contengono coppie di nodi che sono in uguaglianza o disuguaglianza tra loro. In questo modo si passa la coda delle uguaglianze all'algoritmo di chiusura di congruenza e in caso questi restituisca "soddisfacibile", si verificano che anche le disuguaglianze siano rispettate.

3 Algoritmo di chiusura di congruenza

Per ogni uguaglianza rilevata nel parsing, si effettua il *merge*. Già in questa fase si possono fare delle ottimizzazioni. Cominciamo col fatto che invece di passare i nodi al merge, si passano direttamente i loro *rappresentanti* e solamente se questi sono diversi. Questo ci permette di non effettuare controlli rindondanti risparmiando le operazioni di *find*(n1), *find*(n2) e il loro confronto. Infatti notiamo che anche l'algoritmo per trovare gli elementi congruenti le fa già prima del merge. In più, nella *union* quando si imposta un nodo come rappresentante dell'altro creerebbe una catena di nodi fino ad arrivare al vero rappresentante; quindi trovare il rappresentate sarebbe dispendioso perché bisognerebbe scorrerla tutta. Il tempo richiesto verrebbe ridotto al minimo adottando l'accorgimento precedente. Comunque si può sempre adottare la *compressione dei cammini* che durante la risalita della catena fa puntare il campo `find` di ogni nodo incontrato al vero rappresentante.

Passiamo ad analizare il campo `ccpar`. Qui occorre scegliere attentamente la struttura dati perché i nodi in esso contenuti vengono continuamente uniti nella *union*. Si potrebbe decidere di utilizzare una lista con puntatore al primo ed all'ultimo elemento con unione in tempo costante, però se andiamo a modificare così una lista si incontra un problema quando chiamiamo la funzione *espandiCongruenze* che analizza ogni possibile coppia tra i campi `ccpar` per trovare qualche nuova congruenza su cui fare il merge. Quindi nel 'merge' si deve salvare il campo `ccpar` dei due nodi dati in input. Si sono pensate tante soluzioni, la più semplice è quella di salvare i nodi contenuti in `ccpar` in due array stando sicuri che li sarebbero rimasti inalterati anche dopo l'unione. Questo però ha un costo *lineare* nel numero di nodi contenuti nel campo. Poi c'è da considerare i doppi. Attaccando semplicemente una lista in fondo all'altra non ci permette di analizzare i nodi ripetuti e questo, oltre a consumare risorse, ci aumenterebbe il numero delle coppie da analizzare.

A questo punto si sono sviluppate due diverse implementazioni, una per gestire efficientemente i doppi e l'altra cercando di usufruire dell'unione in tempo

lineare. Per il primo scopo si è utilizzata una *lista ordinata* così ogni volta che cerco di unire un nodo che c'è già, lo trovo mentre cerco di inserirlo nella posizione ordinata. Inoltre essendo entrambe già ordinate mi basta scorrerle entrambe solo una volta risultando lineare nella somma del numero di nodi. In questo modo ci accorgiamo che conviene costruire una nuova lista ordinata invece di unirla a quella del rappresentante. Il tempo necessario è sempre lineare, però non c'è bisogno di salvare le liste originali in un array. Per il secondo scopo invece si è creata una struttura dati ad hoc chiamata *DoppiaLista* che non è altro che una lista doppiamente concatenata nella quale ci sono dei campi che puntano alle due liste che la compongono così le liste sottostanti rimangono inalterate e si può chiamare l'*espandiCongruenze* su di esse. Evitando la copia negli array (come nel caso precedente) ed avendo un unione costante!. Ma non è tutto oro quello che luccica, infatti in fase implementativa si è notato che non gestendo gli elementi doppi si ha uno spreco di memoria importante ed *espandiCongruenze* considera combinazioni già incontrate in precedenza. Per ridurre questo fenomeno nell'*espandiCongruenze* si è implementato un meccanismo di rimozione dei doppi, vanificando di fatto il vantaggio. In entrambi i casi si è utilizzato l'ottimizzazione dell'*unione per rango* in modo da unire il più piccolo al più grande, così i cammini verso il rappresentanti vengono allungati di uno solo sul numero minore di nodi. Una caratteristica importante di entrambe le liste è l'estrazione in tempo costante di ogni elemento se preso in ordine (proprio come accade in *espandiCongruenze*) siccome la lista viene scansionata sempre tutta si utilizza un campo aggiuntivo `index` che ad ogni chiamata restituisce l'elemento puntato e si agiorna sul successivo.

Una volta completate le merge e le uguaglianze, ci restano da controllare le disuguaglianze così confrontiamo i rappresentanti dei nodi che devono essere diversi. Però così facendo ci accorgiamo solo alla fine di una disuguaglianza che non viene rispettata magari dopo la prima merge. Per questo è utile il campo "`diversi`" che viene mantenuto *solo* nel rappresentante aggiungendo quelli di tutta la sua classe. Questo campo viene controllato dopo ogni *union* così ci accorgeremmo tempestivamente di una eventuale insoddisfacibilità. Anche qui sono state fatte diverse prove. La lista ordinata è quella più facile da gestrire e non rallenta troppo l'esecuzione. Si è provato anche una tabella hash particolare (*HashOpenPlusList*) nella quale vengono usate delle liste per avere subito accessibili i nodi senza dover scandire tutte le celle della tabella. Questa classe crea tabelle dalle dimensioni opportune (senza troppo spreco di memoria). Questo rallenta un po' l'algoritmo però permette di non dover trovare un compromesso sulla dimensione iniziale della tabella evitando dimensioni esagerate o frequenti rehash. Utilizzando questa hashtable basta controllare che la classe di equivalenza di un nodo sia disgiunta dalla tabella dei diversi.

Le diverse implementazioni comportano che le "merge" possono essere eseguite in ordine diverso tra le due soluzioni, quindi si è analizzato il caso peggiore che si ha quando una formula è soddisfacibile perché in entrambi i casi vengono analizzate tutte. Dai rilevamenti fatti risulta più efficiente la seconda versione con "DoppiaLista" costruita ad hoc. La differenza non si nota tanto nell'esecuzione dell'algoritmo di chiusura di congruenza, ma nella costruzione del grafo.