SECONDA UNIVERSITA' DEGLI STUDI DI NAPOLI

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria della Conoscenza e Sistemi Esperti



Riconoscimento di concetti algoritmici

Relatori:

Prof. Beniamino Di Martino

Studenti:

Carlo Esposito

Pasquale De Stefano

Table of Contents

[Introduzione 3](#_Toc289697319)

[Scenario di riferimento 4](#_Toc289697320)

[Cocktail 4](#_Toc289697321)

[Prolog 6](#_Toc289697322)

[Scheduling e obiettivi del progetto 6](#_Toc289697323)

[Descrizione concetti algoritmici 8](#_Toc289697324)

[Cholesky decomposition 8](#_Toc289697325)

[QR decomposition 8](#_Toc289697326)

[Descrizione Regole prolog 10](#_Toc289697327)

[Glossario KB 10](#_Toc289697328)

[Regole Prolog 11](#_Toc289697329)

[Conclusioni e sviluppi futuri 15](#_Toc289697330)

# Introduzione

In informatica e matematica, con il termine algoritmo si intende, in genere, un metodo per ottenere un certo risultato (risolvere un certo tipo di problema) attraverso un numero finito di passi.

Lo scopo di questa relazione è quello di descrivere passo passo tutte le attività che sono state fatte al fine di implementare uno strumento per il riconoscimento di algoritmi fornendo allo stesso tempo una base a chi si voglia cimentare ad uno sviluppo futuro.

Il riconoscimento di concetti algoritmici è un attività che tipicamente viene svolta al fine di rimpiazzare o comunque modificare la particolare istanza di tali algoritmi con una loro versione più performante, magari parallelizzata o adattata a quello che è l’hardware sottostante.

Si parla di “riconoscimento” perchè evidentemente un algoritmo può essere implementato in innumerevoli forme: l’implementazione di un algoritmo può differire da un altra implementazione per esempio andando a considerare come criterio di paragone il linguaggio di programmazione, il numero di variabili usate, il nome di tali variabili, etc etc.

La prima domanda che ci siamo posti è stata “Puo’ il riconoscimento essere cosi generale da riuscire a riconoscere tutte le possibili implementazioni di un concetto algoritmico?”.

Vediamo un semplice esempio:

Supponiamo di voler riconoscere una moltiplicazione di una variabile qualsiasi per un intero e supponiamo di analizzare il seguente codice:

*For(i=0;i<n;i++)*

*{*

*var= var + var;*

*}*

Ebbene questo ciclo for implementa effettivamente il concetto di moltiplicazione della variabile ‘var’ per un intero 2n.

L’implementazione di un sistema che riconosca tale porzione di codice come un implementazione del concetto “moltiplicazione di una variabile per un intero” è pero’ probabilmente poco utile.

Un implementazione sensata un riconoscitore del suddetto concetto dovrebbe pero’ riconoscere:

*var=var\*5;*

*var=5\*var;*

*costante= 3; var=var\*costante;*

Nel nostro lavoro ci siamo posti come ‘goal’ quello di implementare un riconoscimento che sia il piu generale possibile ma ovviamente abbiamo definito dei limiti nella generalizzazione delle regole prolog.

# Scenario di riferimento

Il problema dell’analisi delle codice al fine di riconoscere concetti algoritmici puo essere visto come la sequenza di due attività distinte:

* Prelievo delle informazioni necessarie all’analisi
* Analisi di tali informazioni

Il lavoro svolto all’interno del progetto partiva infatti da un architettura software organizzata in due moduli distinti:

* Modulo di estrazione delle informazioni rappresentate come fatti prolog
* Modulo di analisi dei fatti prolog mediante apposite regole prolog

Tale soluzione ha degli evidenti vantaggi in termini di facilità di testing e manutenibilità .

Tale scelta inoltre permetteva di suddividere efficacemente il lavoro in due parti avendo supposto che la mole di lavoro necessaria su i due moduli sia grosso modo la stessa.

## Cocktail

Vediamo ora in dettaglio il primo modulo del quale si è occupato lo studente Pasquale De Stefano.

Lo strumento scelto per analizzare il codice al fine di estrarre le informazioni sotto forma di fatti prolog è il Cocktail Toolbox, un set di tool che consentono di affrontare praticamente ogni fase della costruzione di un compilatore.

Anche in questo caso la scelta di questo strumento è legata al fatto che l’applicativo finale risulterà implementato in maniera fortemente modularizzata in modo tale da rendere facile implementare l’analisi su codici sorgenti di diversi linguaggi di programmazione. Dei diversi moduli presenti nel Cocktail Toolbox quelli di interesse per il nostro obiettivo sono il modulo AST ed il modulo Puma.

Il modulo AST è un generatore di abstract syntax tree che costruisce in modo automatico

moduli software e strutture dati per la manipolazione degli attributi di un albero.

Le specifiche per questo tool sono basate su grammatiche generative estese definite in appositi file “.ast”.

Ast fornisce inoltre la possibilità di visualizzare graficamente il risultato della sua elaborazione il che è stato utile nella fase di debugging.

Puma invece è un generatore automatico di moduli operanti trasformazioni su abstract

syntax tree. Puma è basato sul pattern-matching e sulla ricorsione.

Il lavoro da fare era essenzialmente quello di verificare la correttezza di tale modulo e, se necessario, correggerlo.

Il file responsabile della generazione dei fatti prolog è il file *TraverseSyntaxTree.puma* presente nella directory *java grammar\alcor\sources.*

In allegato la versione precedente alla modifica (doppio clic sull’icona o vedere il file allegato “File Puma vecchio.txt”):



Tale file presenta **1272** righe di codice con **0** righe di commento organizzate in un unica procedura. Inoltre permette di esprimere sotto forma di fatti prolog solamente i seguenti 9 concetti:

method(...) concetto di metodo

class(...) concetto di classe

parameter(...) concetto di parametro

constructor(...) concetto di costruttore

val\_inst(…) concetto di costante

scalar\_var\_inst(…) concetto di istanzazione di una variabile scalare

array\_elem\_inst(…) concetto di istanzazione di un element di un array

scalar\_var\_def(…) concetto di definizione di uno scalare

array\_var\_def(…) concetto di definizione di un array

Come verrà chiarito in seguito, di tali fatti prolog, i primi 4 non sono utili all’analisi dei concetti algoritmici e gli ultimi 5 non sono sufficenti al riconoscimento dei concetti algoritmici scelti.

E’ quindi stato necessario modificare il file allegato in modo estrarre ,sotto forma di fatti prolog, la nostra base di conoscenza (KB).

E’ stato scelto anche di modificare la struttura del fali in modo da renderlo più modulare (rimpiazzando l’unica unica procedura di 1200 righe di codice con piu procedure) e più performante (non era stata utilizzata la funzionalità di TRIP, procedura con attraversamento automatico dell’ast).

In allegato la versione successiva alla modifica (doppio click sull’icona o vedere il file allegato “File Puma nuovo.txt”):



Il nuovo file è sicuramente più manutenibile e traduce in fatti prolog tutte le informazioni necessarie al riconoscimento di fatti prolog.

Inoltre presenta un numero di righe di codice inferiore del 20%!

Questo è dovuto all’utilizzo della già citatà funzione TRIP che permette l’attraversamento automatico dell’ast.

Infine resta da segnalare un bug che mandava in crash il programma di parsing del codice. Il file responsabile del crash è il file TraverseBuildPDG\_s.puma, quello responsabile della generazione dei fatti relativi al program dependance graph.

Visto che al momento sia i fatti relativi alla dependenza sul controllo che sulla dipendenza sui dati vengono aggiunti a mano il problema è stato risolto temporanemanete commentando la riga 43 di tale file.

## Prolog

Vediamo ora in dettaglio secondo modulo del quale si è occupato lo studente Carlo Esposito.

Questo modulo è essenzialmente un insieme di regole prolog che, applicato ai fatti prolog (KB) generati dal primo modulo, permette il riconoscimento dei concetti algoritmici richiesti.

La prima attività svolta era quella di confrontare due diversi file (doppio click sulle icone oppure vedere i file allegati “rul\_gen.pl.txt” e “rul\_gen\_java.pl.txt”) :



Il file rul\_gen\_java.pl risulta successivo al file rul\_gen.pl e presenta numerose aggiunte e modifiche. Si è quindi deciso di prendere il file [rul\_gen\_java.pl](http://rul_gen_java.pl/) come base per il lavoro di progetto.

Una volta capito quale è il file da analizzare il primo passo è stato quello di definire l’insieme di fatti prolog sul quale si basano le regole (KB).In pratica si è verificato, per ogni riferimento ad un fatto prolog, se tale fatto era presente nel file oppure era necessaria la sua generazione da parte del primo modulo.

In allegato la KB necessaria al file rul\_gen\_java (doppio click sull’icona o vedere il file allegato “KB.txt”):



La seconda fase di analisi è stata quella di testare le regole prolog.

Anche in questo caso è stato creato un file apposito che contiene il risultato del testing, in allegato (doppio click sull’icona o vedere il file allegato “testing.txt”:



In breve un analisi puntuale delle regole ha evidenziato degli errori sin dalle prime regole testate.

Alcuni errori sono stati corretti in quanto erano chiaramente dovuti dell’uso di una naming diversa tra il modulo Prolog ed il modulo Puma. Poi si è deciso di affrontare il problema in maniera diversa (vedi Scheduling e obiettivi del progetto).

## Scheduling e obiettivi del progetto

Il primissimo obiettivo del progetto era quello di correggere l’architettura software di riferimento al fine di riconoscere il concetto di livello gerarchicamente più alto (il prodotto matriciale).

Le due risorse erano state quindi assegnate ai due moduli citati precedentemente.

Il secondo obiettivo del progetto era invece quello di riconoscere 2 nuovi concetti( la descrizione dei 2 concetti verrà data nel prossimo paragrafo).

Dopo alcune settimane di lavoro pero’ ci siamo resi conto che le due attività richiedevano diverso effort a causa di svariati motivi.

Il primo modulo è scritto in un linguaggio sul quale una delle risorse aveva già lavorato ed era quindi familiare; inoltre i requisiti da soddisfare sono stati chiari dopo pochi giorni e il lavoro non ha presentato particolari problemi.

Il secondo modulo invece prevedeva la correzione di complesse regole scritte in prolog strutturate in maniera poco modulare, non commentate e spesso basate su ‘fatti’ prolog il cui significato non era comprensibile (vedi documento il allegato “Rul\_Gen\_Java.docx”).

Alla luce di queste considerazioni il lavoro è stato distribuito diversamente, in pratica entrambi abbiamo lavorato assieme sul secondo modulo andando a riscrivere da zero l’insieme di regole per il riconoscimento dell’algoritmo di moltiplicazione matriciale cercando di scrivere una regola che sia la più generale possibile.

Occasionalmente è stato necessario tornare a lavorare sul primo modulo in modo da modificare/ampliare la KB generata.

In allegato la KB necessaria al nuovo file prolog.

In definitiva il prodotto finale presenta le seguenti caratteristiche:

-alta modularità e leggibilità

-ogni procedura e regola prolog è stata commentata e documentata

-la KB è stata ampliata in modo da rendere,di riflesso, le regole prolog più semplici

# Descrizione concetti algoritmici

Vediamo ora in dettaglio quali sono i concetti algoritmici da riconoscere:

Cholesky decomposition -Carlo Esposito

QR decomposition -Pasquale De Stefano

## Cholesky decomposition

In [algebra lineare](http://it.wikipedia.org/wiki/Algebra_lineare) la decomposizione di Cholesky è la [fattorizzazione](http://it.wikipedia.org/wiki/Fattorizzazione) di una [matrice hermitiana](http://it.wikipedia.org/wiki/Matrice_hermitiana) [definita positiva](http://it.wikipedia.org/wiki/Matrice_definita_positiva) in una [matrice triangolare](http://it.wikipedia.org/wiki/Matrice_triangolare) inferiore e nella sua [trasposta coniugata](http://it.wikipedia.org/wiki/Matrice_trasposta_coniugata).

L'algoritmo di Cholesky Banachiewicz dà una formula per calcolare direttamente le entrate della matrice triangolare inferiore L. Esso inizia formando l'angolo superiore sinistro della matrice L e procede a calcolare la matrice riga per riga.

\forall i = 1,\dots,m

\forall j = 1,\dots,(i - 1)

l_{i,j} = \frac{1}{l_{j,j}} \left(a_{i,j} - \sum_{\iota = 1}^{j-1} l_{i,\iota} l_{j,\iota}\right)

 l_{i,i} = \sqrt{ a_{i,i} - \sum_{k=1}^{i-1} l_{i,k}^2 }. 

Vediamo ora una semplice implementazione di tale algoritmo in Java:

## QR decomposition

Sia A una matrice mxn con m>n.

La decomposizione QR è la scomposizione di tale matrice in 2 nuove matrici :

- Q, matrice mxn le cui colonne sono le ortonormalizzazioni delle colonne di A

- R, matrice nxn triangolare superiore

In matematica, una trasformazione di Householder in uno spazio tridimensionale è la riflessione dei vettori rispetto ad un piano passante per l'origine.

La matrice Q può essere usata per riflettere un vettore in modo tale che tutte le sue coordinate eccetto una scompaiono. Sia x un arbitrario vettore colonna m-dimensionale di lunghezza |α| (per la stabilità numerica si assume che α abbia lo stesso segno della prima coordinata di x).

Se e1 è il vettore (1,0,...,0)T, e se || ... || denota la norma euclidea, si considerano le costruzioni

u = x − αe1,

v = u / ||u||,

Q = I - 2 vvT .

Per la matrice di Householder Q si ha

Qx = (α,0,...,0)T.

Questo genere di trasformazione può essere usato per trasformare gradualmente una matrice A di tipo m × n nella forma triangolare superiore. Innanzitutto si moltiplica A per la matrice di Householder Q1 ottenuta scegliendo x per la sua prima colonna. Questa risulta in una matrice QA che presenta zeri nella colonna sinistra, ad eccezione della sola prima riga.

Q_1A = \begin{bmatrix}
                   \alpha_1&\star&\dots&\star\\
                      0    &     &     &    \\
                   \vdots  &     &  A' &    \\
                      0    &     &     & \end{bmatrix}

Questa modifica può essere ripetuta per la A mediante una matrice di Housholder Q2. Si noti che Q2 è più piccola della Q1. Poiché vogliamo che sia reale per operare su Q1A invece di A' abbiamo bisogno di espandere questa nella parte superiore sinistra, riempiendola di entrate 1, o in generale:

Q_k = \begin{bmatrix} I_{k-1} & 0\\ 0 & Q_k'\end{bmatrix}

Dopo t iterazioni di questo processo, con t = min(m − 1, n), si giunge ad

R = Qt...Q2Q1A

che è una matrice triangolare superiore. In tal modo, con

Q = Q1Q2...Qt,

A = QR è una decomposizione QR di A.

# Descrizione Regole prolog

## Glossario KB

Descrizione della nuova KB introdotta (vedere il file allegato “KB\_NEW.pl.txt”).

* **Definizione di un array**

array\_var\_def(Nome,Tipo,Dimensione,ID,[DimRighe,DimColonne],Classe,Metodo).

* **Riferimento ad un array**

array\_elem\_ref(ID\_Ref,Nome,[Nome\_VarRighe,Nome\_VarColonne]).

* **Instanziazione di un array**

array\_elem\_inst(ID,Nome,[NomeVarRighe,NomeVarColonne],Metodo,Classe).

* **Riferimento ad una costante**

constant\_ref(ID,Valore).

* **Chiamata di una funzione**

chiamata(ID,NomeFunzione,[Parametri]).

* **Operazione aritmetica: Divisione**

divide(ID\_Expr,Op1,Op2,Classe,Metodo).

* **Cilco For**

for\_r(ID,ID\_init,ID\_exit,ID\_incr,Classe,Metodo).

* **Operazione di confronto “maggiore o uguale”**

greater\_equal(ID\_Expr,Op1,Op2,Classe,Metodo).

* **Operazione di confronto "minore"**

less(ID\_Expr,Op1,Op2,Classe,Metodo).

* **Operazione aritmetica: Sottrazione**

minus(ID\_Expr,Op1,Op2,Classe,Metodo).

* **Operazione di post decremento (i--)**

post\_decr(ID\_Expr,Op,Classe,Metodo).

* **Operazione di post incremento (i++)**

post\_incr(ID\_Expr,Op,Classe,Metodo).

* **Operazione aritmetica: Addizione**

plus(ID\_Expr,Op1,Op2,Classe,Metodo).

* **Definizione di una variabile scalare**

scalar\_var\_def(NomeVariabile,ID,Classe,Metodo).

* **Istanziazione di una variabile scalare**

scalar\_var\_inst(ID,NomeVariabile,Classe,Metodo).

* **Riferimendo ad una variabile scalare**

scalar\_var\_ref(ID,NomeVariabile).

* **Operazione aritmetica: Moltiplicazione**

times(ID\_Expr,Op1,Op2,Classe,Metodo).

* **Operatore di assegnazione**

uguaglianza(Op1,Op2,Classe,Metodo).

## Regole Prolog

Descrizione delle regole Prolog utilizzate (vedere i file allegati “regole\_di\_base.pl.txt”, “regole\_decomposizione\_Cholesky.pl.txt” e “regole\_prodotto\_matriciale.pl.txt”)

* **plus\_commutativo(Expr,Op1,Op2,Classe,Metodo)**

Descrizione: Riconosce l’operazione commutativa di somma.

Parametri:

* + - Expr: Identificativo espressione
    - Op1: Primo operatore
    - Op2: Secondo operatore
    - Classe: Classe
    - Metodo: Metodoù
* **times\_commutativo(Expr,Op1,Op2,Classe,Metodo)**

Descrizione: Riconosce l’operazione commutativa di prodotto.

Parametri:

* + - Expr: Identificativo espressione
    - Op1: Primo operatore
    - Op2: Secondo operatore
    - Classe: Nome della classe
    - Metodo: Nome del metodo
* **Ciclo\_for(ID, Variabile, Funtore\_init, Funtore\_exit, Funtore\_incr, Classe, Metodo)**

Descrizione: Riconosce una vasta gamma di cicli for.

Parametri:

* + - ID: Identificativo ciclo for
    - Variabile: Variabile del ciclo for
    - Funtore\_int: Condizione di inizializzazione
    - Funtore\_exit: Condizione di uscita
    - Funtore\_incr: Condizione di incremento
    - Classe: Nome della classe
    - Metodo: Nome del metodo.
* **copia\_array(ID,Array1,Array2)**

Descrizione: Riconosce la copia di un array

Parametri:

* + - ID: Identificativo dell’espressione
    - Array1: nome dell’array di destinazione
    - Array2: nome dell’array di origine
* **appartenenza(Equazione,Id,Posizione)**

Descrizione: Riconosce la presenza di una variabile in un ‘espressione

Parametri:

* + - Equazione: Identificativo espressione
    - Id: identificativo variabile
    - Posizione: può essere o usata [ a destra (right)] o definita [ a sinistra (left)]
* **equivalenza(Equazione,Id1,Oggetto\_equivalente )**

Descrizione: rappresenta il fatto che in una certa equazione è equivalente avere un riferimento ad una variabile id1 piuttosto che ad un'altra espressione.

Parametri:

* + - Equazione: Identificativo espressione
    - Id1: Identificativo variabile
    - Oggetto\_equivalente: funtore o un identificativo di una variabile
* **righe\_matrice(Matrice, N)**

Descrizione: Riconosce una Matrice con dimensione delle righe pari a N

Parametri:

* + - Matrice: nome della matrice
    - N: dimensione delle righe della matrice
* **colonne\_matrice(Matrice, N)**

Descrizione: Riconosce una matrice con dimensione delle colonne pari a N

Parametri:

* + - Matrice: nome della matrice
    - N: dimensione delle colonne della matrice
* **matrice\_bidimensionale(Matrice)**

Descrizione: Riconosce una matrice di dimensione 2.

Parametri:

* + - Matrice: nome della matrice
* **true\_data\_dependence(ID1, ID2)**

Descrizione: Restituisce “true” quando ID2 è data dependent da ID1

Parametri:

* + - ID1: Identificativo prima espressione
    - ID2: Identificativo seconda espressione
* **prodotto\_matriciale(Matrice1,Matrice2,Matrice3)**

Descrizione: Riconosce il prodotto matriciale C = A\*B

Parametri:

* + - Matrice1: nome della matrice A
    - Matrice2: nome della matrice B
    - Matrice3: nome della matrice risultato C
* **triangolarizzazione\_inferiore\_matrice(Matrice)**

Descrizione: Riconosce la triangolarizzazione inferiore di una matrice.

Parametri:

* + - Matrice: Nome della matrice triangolarizzata.
* **decomposizone\_di\_Cholesky(Matrice1, Matrice2)**

Descrizione: Riconosce la decomposizione di Cholesky

Parametri:

* + - Matrice1: Nome della matrice da decomporre
    - Matrice2: Nome della matrice decomposta

# Conclusioni e sviluppi futuri

Lo strumento Alcor (algorithm concepts reconizer) riesce a riconoscere correttamente i due concetti presentati precedentemente.

Entrambi i moduli dove si è intervenuto ora sono sicuramente più mantenibili e più facilmente espandibili. Un possibile sviluppo futuro è sicuramente quello di implementare l’estrazione delle informazioni sulle data dependence e control dependence che al momento sono aggiunte a mano di volta in volta.

Un altro possibile sviluppo futuro potrebbe invece essere quello di applicare una trasformazione al codice responsabile di uno specifico algoritmo in modo da renderlo più performante.