PLATAM: implementazione in Prolog di Tecniche criptografiche basate su reticoli

Progetto per il Corso di Logica Computazionale II

Andrea Carcano (61040) e Moreno Carullo (608371)
andrea.carcano@gmail.com,moreno.carullo@pulc.it

maggio-luglio 2007

1 Introduzione

La maggior parte degli schemi di cifratura a chiave pubblica affida la propria sicurezza a due problemi difficili: la fattorizzazione di interi ed il logaritmo discreto su campi finiti. I due problemi inoltre sembrano essere correlati, di conseguenza algoritmi efficienti per l'uno si applicano con alcune modifiche anche all'altro.

Questo potrebbe essere potenzialmente pericoloso, poichè gli algoritmi utilizzati quotidianamente (RSA per esempio) nelle transazioni bancarie, finanziarie e commerciali appoggiano la loro sicurezza sull'intrinseca difficoltà dei problemi sopra menzionati. Si aggiunga inoltre il fatto che se in futuro le tecnologie di quantum computing riuscissero ad arrivare in qualche modo alla produzione industriale, seppur di piccola scala, le versioni quantistiche degli algoritmi per la risoluzione della fattorizzazione di interi potrebbero creare un vero e proprio problema di sicurezza. Questo tuttavia rimane ancora nel futuribile, dal momento che allo stato attuale delle cose la produzione in scala di elaboratori quantistici sembra ancora irraggiungibile.

Nasce quindi l'esigenza di cercare altri problemi computazionalmente difficili, magari imponendo la difficoltà di risoluzione anche con algoritmi quantistici. Negli ultimi dieci anni di ricerca una delle tecniche criptografiche alternative venute alla luce costruisce la sicurezza attorno ai reticoli (*lattice* in inglese), una costruzione algebrico-geometrica su cui sono definiti alcuni problemi NP-hard.

Scopo del presente progetto è implementare in Prolog delle primitive criptografiche basate su reticoli. Il documento è strutturato in altre cinque sezioni:

- sezione 2, dove si analizzano gli algoritmi e gli schemi implementati nel presente progetto.

- sezione 3, dove vengono presentati ad alto livello i predicati che si vogliono ottenere, e dove verranno fatte alcune considerazioni in merito ai costi ed ai limiti incontrati nell'implementazione Prolog.

- sezione 4, dove viene dettagliata la suddivisione in moduli dell'implementazione, e dove per ogni modulo ne viene documentata la struttura e le scelte particolari relative ai predicati ivi contenuti.

- sezione 5, dove viene esemplificato l'utilizzo del programma per mezzo di query.

- infine le conclusioni in cui vengono tirate le somme del progetto realizzato.

Le fasi di definizione dei requisiti di alto livello e la suddivisione in moduli sono state portate avanti parallelamente da entrambi gli autori, nessuna nota verrà quindi aggiunta in tali sezioni del documento. Nella sezione 4 verrà invece dettagliato per ogni predicato o gruppo di predicati l'autore dell'implementazione. **PLATAM** è acronimo di **P**rolog **LAT**tice by **A**ndrea e **M**oreno.

2 Tecniche criptografiche basate su reticoli

Le tecniche criptografiche basate su reticoli consentono di costruire algoritmi di cifratura a chiave pubblica, di firma digitale, e di dimostrare la correttezza di funzioni hash criptografiche. In questo progetto si è voluto realizzare un'implementazione di schema di cifratura a chiave pubblica ed un'implementazione di funzione hash criptografica.

Le costruzioni e gli algoritmi a cui faremo riferimento sono documentate nei lavori di Goldreich et al., rispettivamente in [1] per la funzione hash criptografica *collision resistant* ed in [2] per lo schema a chiave pubblica. L'implementazione qui documentata segue più o meno letteralmente quanto si può trovare nei due paper, con i dovuti adattamenti necessari per venire incontro ai limiti dei programmi Prolog: in particolare si richiedeva la capacità di effettuare operazioni complesse su matrici (moltiplicazione, determinante, inversione), dal punto di vista pratico la dimensione di tali matrici su cui opereremo sarà per forza di cose ridotta.

2.1 Reticoli

Un reticolo $L(B)$ è definito dall'insieme finito $B = \langle b_1, \ldots, b_n \rangle$ di vettori $b_i \in \mathbb{R}^n$ linearmente indipendenti, più precisamente è costituito da tutti i vettori generati

come combinazione lineare della *base* B a coefficienti interi:

$$L(B) = \left\{ \sum_{i}^{n} a_i b_i \ : \ a_i \ \in \mathbb{Z} \right\}$$

Dato un reticolo L sono definiti tre problemi difficili, correlati tra loro:

- **Vettore più piccolo** (*SVP*, Shortest Vector Problem): data una base B trovare $p \in L(B) : \|p\| < \|x\|, \ \forall x \in L(B)$.
- **Vettore più vicino** (*CVP*, Closest Vector Problem): data una base B ed un punto $c \in R^n$, trovare il punto $p \in L(B)$ più vicino a c.
- **Base più piccola** (*SBP*, Smallest Vector Problem) data la base B di $L(B)$, trovare una base $B' : L(B') = L(B)$ con difetto ortogonale minimo. Non esiste un algoritmo polinomiale in n, dove n è la dimensione del reticolo

Esistono principalmente due algoritmi per la risoluzione approssimata di questi problemi: LLL e la tecnica di arrotondamento di Babai (*Babai Roundoff*). Il primo risolve direttamente il problema SBP con un fattore di approssimazione esponenziale in n e come conseguenza di questo ottiene un'approssimazione del problema SVP. Il secondo ottiene un'approssimazione di *CVP*.

2.2 Funzione hash criptografica

Al fine di definire una funzione hash con i parametri di sicurezza n, m, q (dove $n \ log(q) < m \leq \frac{q}{2n^4}$ e $q = O\left(n^c\right)$) serve considerare una matrice $M \in \mathbb{Z}_p^{n \times m}$ ed una stringa $s \in \{0,1\}^m$, la funzione $h_M(s)$ è quindi definita nel modo seguente:

$$h_M(s) = Ms \bmod q = \sum_i s_i M_i \bmod q$$

tale funzione gode della proprietà "collision-free", ovvero risulta molto difficile/improbabile trovare due stringe s ed s' tali che $h_M(s) = h_M(s')$: la dimostrazione di sicurezza viene data in [1].

2.3 Cifratura a chiave pubblica

La cifratura a chiave pubblica prevede due funzioni Encrypt e Decrypt: la prima data una chiave pubblica ed un messaggio *plaintext* produce un messaggio cifrato, il *cyphertext*; la seconda dato un *cyphertext* e la coppia di chiavi pubblica e privata ottiene nuovamente il *plaintext*. Vengono di seguito descritte le versioni basate su reticoli delle due funzioni:

Encrypt preso in input $\vec{x}$ un vettore di interi e $B, \sigma, \vec{e}$ restituisce $c = f_{B,\sigma}(\vec{x}, \vec{e}) \equiv B\vec{x} + \vec{e}$.

- il vettore $\vec{x}$ rappresenta il messaggio in forma binaria da cripatare.
- il vettore $\vec{e}$ è generato random in $\mathbb{R}^n$
- ogni componente di $\vec{e}$ può valere $\pm\sigma$ con probabilità $1/2$

Decrypt dati R, c, B in input si inverte la funzione utilizzando *Babai's round off algorithm* o **LLL**

- tramite R si trova il punto del lattice p, tramite una versione approssimata di CVP.
- si calcola $x = B^{-}1p$

si rimanda al paper di Golderich et al. [2] per i dettagli e la dimostrazione di sicurezza.

3 Requisiti e definizione Prolog del problema

I requisiti alla base dell'implementazione proposta in questo progetto definiscono cinque predicati di alto livello, a cui dovà essere affiancato un numero opportuno di predicati ausiliari e non.

- un predicato che data una matrice M ed una stringa binaria s calcoli la funzione hash definita nella sezione 2.

- un predicato di generazione della chiave privata, da usarsi nella cifratura asimmetrica.

- un predicato di generazione della chive pubblica, da usarsi nella cifratura asimmetrica.

- un predicato di criptazione che data la chiave pubblica ed un messaggio in chiaro (plaintext) fornisca in uscita la versione cifrata (cyphertext).

- un predicato di decifratura che data la coppia chiave privata/chiave pubblica ed un messaggio cifrato permetta di risalire al plaintext.

Nel presente documento verrà utilizzata la sintassi PlDoc [3] nell'illustrare i predicati, quindi gli argomenti dei predicati saranno preceduti da un + ad indicare il requisito di istanziazione in una query, e dal simbolo ? qualora l'istanziazione non sia richiesta. L'implementazione Prolog utilizzata per lo sviluppo ed il test del progetto è SWI-Prolog 5.6.34 nelle versioni per Linux e Mac OS X.

4 Suddivisione in moduli

La definizione dei moduli sarà effettuata nel modo più semplice possibile, cercando di evitare predicati e direttive specifiche di un particolare interprete Prolog. Di conseguenza non vengono utilizzati predicati extralogici per la definizione della visibilità al di fuori del modulo, ma solamente una versione più corretta del predicato consult per il caricamento dei moduli esterni: `ensure_loaded(FILE)`.

Le tecniche criptografiche basate su reticoli si basano sul calcolo matriciale su campi finiti: sarà quindi necessario un modulo matrici, che a sua volta conterrà al suo interno operazioni di base su vettori e liste. Si richiedono quindi tre moduli di utilità:

1. liste
2. vettori
3. matrici

A questi si aggiungono i due moduli dedicati rispettivamente alla funzione hash criptografica ed alla cifratura a chiave pubblica. È bene notare che la suddivisione tra gli autori dei predicati qui documentata non è esaustiva, dal momento che i predicati "minori" non vengono qui commentati.

4.1 Modulo liste

Contiene un totale di sedici predicati, tra cui alcuni elementari come il calcolo della lunghezza di una lista, del massimo e del minimo. Verranno documentati quelli non già meglio commentati nel libro del corso:

- `lista_elem(+Lista, +Posizione, ?EnnesimoElemento)`: restituisce il valore contenuto nella posizione data in input. [Andrea]
- `lista_elem_canc(+Lin,+N,?Lout)`: istanzia nella variabile Lout con una la lista di partenza meno l'elemanto in posizione N. [Andrea]
- `lista_notnull(+Lista,?Pos,?A)`: data una Lista istanzia A con il primo elemento della lista diverso da zero, e Pos con la posizione in cui tale elemento si trova. [Moreno]
- `list_join(+Lista1,+Lista2,?Lista3)`: istanzia la variabile Lista3 con la concatenazione in coda della Lista2 alla Lista1. [Moreno]
- `list_blank(?Lista,+Lunghezza)`: istanzia la variabile Lista con una lista di Lunghezza specificata. [Moreno]

- `lista_fillto(+ListaIn,?ListaOut,+Lunghezza)`: istanzia la variabile ListaOut con una lista di Lunghezza desiderata, in cui i primi elementi sono coincidenti con quelli di ListaIn ed i rimanenti sono messi a 0. [Moreno]

- `lista_scroll(+Lista,?ListaRisult)`: ruota la Lista ed istanzia in ListaRisult il risultato, dove per ruotare si intende prendere il primo elemento in testa e spostarlo in coda. [Moreno]

- `lista_cut_length(+Lista,+Lunghezza,?ListaTagliata)`: istanzia nella variabile ListaTagliata i primi Lunghezza elementi di Lista. [Moreno]

- `lista_split(+Lista,+Block,?Out)`: divide la Lista in blocchi la cui dimensione è specificata dal parametro Block, ed istanza in Out una lista di liste lunghe al più Block. [Moreno]

4.2 Modulo vettori

Il modulo contiene al suo interno alcune operazioni elementari eseguite sui vettori:

- `vector_number_mult(+Vettore,+Numero,?VettoreRisultato)`: di interesse pubblico, moltiplica ogni componente di un Vettore per un numero. [Andrea]

- `vector_sum(+Vector1,+Vector2,?Result)`: somma due vettori dati in ingresso. [Andrea]

- `vector_sum_if0(+Vettore1,+Vettore2,?Result)`: restituisce in Result la somma dell'i-esima componente di Vettore2 a Vettore1 se questa è uguale 0, altrimenti mantiene il valore di Vettore1. [Moreno]

- `vector_sum_gf(+Vector1,+Vector2,?Result,+Gf)`: somma due vettori in ingresso sul campo di Galois. [Andrea]

- `vector_random_gf(?Vector,+Rows,+Columns,+Gf)`: genera un vettore random sul campo di Galois. [Andrea]

- `vector_random_int(+N,+Intervallo,?Vout)`: genera un vettore calcolando ogni componenete in modo random all'interno di un intervallo. [Andrea]

- `vector_round(+Vettore,?VettoreRounded)`: arrotonda ogni componente del vettore d'ingresso all'intero più vicino. [Andrea]

- `vector_dot_product(+Vector1,+Vector2,?Result)`: esegue il prodotto scalare tra Vector1 e Vector2 ed istanzia il risultato in Result. Esiste anche una variante del predicato che opera su campi finiti, definita in modo analogo ma

con postfisso `_gf` e con l'ulteriore parametro Gf che specifica la dimensione del campo. [Moreno]

Sono state inoltre implementate alcune funzioni che trovano la loro utilità nel calcolo delle basi pubbliche e private:

- `vector_error(+Dimensione,+Sigma,?Vout)`: genera un vettore random di dimesione N assegnando a ogni componente $\pm Sigma$. [Andrea]

- `vector_uno(+Dim,+Pos,?Vout)`: genera un vettore di 0 mettendo 1 nella posizione data in input. Questo predicato di interesse pubblico viene utilizzato nel modulo metrici per generare la matrice identica. [Andrea]

- `vector_random_noise(?VettoreRandom,+Rows)`: genera un vettore detto di rumore. Ogni elemento pu assumere un valore che varia tra $\langle -1, 0, 1 \rangle$. Il predicato è utilizzato all'interno del modulo chiave pubblica per generare partendo dalla chiave privata(una matrice a basso difetto ortogonale) la chiave poubblica. [Andrea]

- `vector_round(+Vettore,?VettoreRounded)`: Arrotonda un vettore di reali all'intero più vicino. [Andrea]

4.3 Modulo matrici

Il modulo prevede che le matrici vengano rappresentate per mezzo di una lista di liste, ed in particolare una lista di colonne. Alcuni dei predicati richiedono che la matrice sia quadrata: questo perchè negli schemi di cifratura a chiave pubblica le matrici su cui si opera sono sempre quadrate, ed il conteggio unificato di colonne e righe consente di evitare alcuni sottogoal.

- `matrix_det(+Matrice,?Det)`: il predicato calcola il determinante di una matrice quadrata 2x2, 3x3 (con regola di Sarrus) e 4x4 (con espansione di Sarrus) e NxN con la condensazione pivotale [4]. La presenza di 3 differenti *casi base* permette di ridurre la profondità dell'albero di dimostrazione, permettendo quindi di aumentare la dimensione massima della matrice su cui è possibile calcolare il determinante. Il predicato è deterministico, e viene fatto uso del taglio per semplificarne la struttura. È stato inoltre segnato come *dinamico*, consentendo così l'utilizzo dei predicati della famiglia `assert*()` e `retract*()`. [Moreno]

- `matrix_build_piv(+Matrice,?Det)`: costruisce la condensazione pivotale della matrice in ingresso, effettuando l'espansione seguendo la prima riga. Vengono utilizzati due predicati ausiliari per la gestione del calcolo degli elementi

per ogni riga e per ogni colonna, ed in tutti viene fatto uso del taglio verde. [Moreno]

- `matrix_invert(+Matrice,?MatriceInversa)`: inverte Matrice e restituisce il risultato in MatriceInversa, utilizzando il calcolo dei minori. Il predicato è stato definito in due versioni: la prima per calcolare immediatamente la matrice inversa nel caso di dimensione 2x2 e la seconda per il caso generico, tagliando con un cut verde nel caso di unificazione con la prima versione. Così come per il determinante sono stati definiti due predicati ausiliari per il calcolo degli elementi di ogni riga e di ogni colonna. Per evitare di aggiungere ad ogni predicato l'argomento "numero di righe" è stato fatto uso dei predicati `asserta()` e `retract()`. [Moreno]

- `matrix_random_gf(?Matrix,+Row,+Columns,+Gf)`: preso in input il campo Galois, il numero di righe e il numero di colonne genera una matrice random all'interno del campo specificato. Ogni colonna viene generata casualmente utilizzando il predicato *vector_random_gf* definito nel modulo vettori. [Andrea]

- `matrix_random_inteval(?Matrix,+Row,+Columns,+Int)`: genera una matrice random in un intervallo dato in input. [Andrea]

- `matrix_elem(+Matrix,+Row,+Col,?A)`: restituisce l'elemento alla riga Row e alla colonna Col, utilizza al suo interno il predicato *lista_elem* definito nel modulo liste. [Andrea]

- `matrix_i(?M,+Rows,+Cols)`: genera la Matrice identica, utilizzando il predicato definito nel modulo vettori: *vector_unoi* [Andrea]

- `matrix_addnoise(+M,+I, +Dim,?Mout)`: aggiunge nella colonna I del rumore. Per rumore s'intende un vettore dove ogni componente può valere $\langle -1, 0, 1 \rangle$. [Andrea]

Non vengono qui riportati e commentati tutti i predicati di utilità utilizzati all'interno del modulo, ed in particolare tutti i predicati ausiliari richiesti da `matrix_det` e `matrix_invert()`.

4.4 Modulo hash criptografico

In questo modulo viene implementata la funzione hash criptografica descritta nella sezione 2. Al suo interno si trova un'unico predicato di interesse pubblico e due utilizzati internamente.

- `latthash_check_sp(+N,+M,+Q)`: verifica che i tre parametri di sicurezza soddisfino i vincoli dettagliati nella sezione 2.2. [Moreno]

- `latthash_base(+Mat,+Gf,+BlockString,?Hash)`: di interesse privato, applica la funzione hash definita dalla matrice *Mat* su di un singolo blocco *BlockString* istanziando nella variabile *Hash* il risultato. La funzione hash criptografica, descritta in [1], è una moltiplicazione di matrici sul campo di Galois *Gf* - a tale semplice definizione corrisponde una complessa dimostrazione di sicurezza basata sul problema CVP. [Moreno]

- `latthash(+Mat,+Gf,+LongString,?Hash)`: di interesse pubblico, applica la funzione di hash sulla lista *LongString* ed istanzia la variabile *Hash* con il risultato. La matrice *Mat* ed il campo di Galois *Gf* identificano una particolare istanza della funzione hash. [Moreno]

- `latthash_list(+Mat,+Gf,+BlockList,?Hash)`: di interesse privato, è utilizzato dal predicato precedente per applicare la funzione hash su liste di numeri lunghe a piacere. La definizione della testa del predicato al contrario delle precedenti ha come terzo argomento una lista di liste, su cui viene applicato induttivamente il predicato di base. [Moreno]

4.5 Modulo chiave pubblica

I predicati di tale modulo utilizzano le funzioni offerte dalle librerie vettori, liste e matrici per implementare la cifratura a chiave pubblica basata su reticoli.

- `lattice_pk_priv_gen(+Dimensione,+Maxint,?R)`: genera la chiave privata della dimensione specificata ed i cui valori risiedono in un intervallo definito da Maxint, ed il risultato è restituito nell'istanziazione di R. La generazione della chiave prevede la creazione di una matrice random a cui viene aggiunta una matrice ottenuta moltiplicando la matrice identica (in cui ad una delle colonne viene aggiunto del rumore) per lo scalare $2n$, dove n è la dimensione specificata. [Andrea]

- `lattice_pk_pub_gen(+PrivateKey, ?PublicKey)`: genera la chiave pubblica partendo dalla chiave privata. La generazione della chiave prevede la moltiplicazione della matrice(private key) per una matrice identica a cui viene aggiunto del rumore. All'ennsimo passaggio viene modificata l'ennesima colonna della matrice identica.

  [Andrea]

- `lattice_pk_encrypt(+PlainText,?CypherText,+ChiavePubblica)`: cifra il messaggio PlainText utilizzando la ChiavePubblica ed istanzia CypherText con il *cyphertext*. Al suo interno utilizza alcuni predicati ausiliari che consentono di gestire Messaggi di lunghezza arbitraria, ed i predicati `asserta()` e `retract()` per evitare da una parte l'aggiunta di un ulteriore parametro e dall'altra l'invocazione di goal complessi ad ogni iterazione: questo è utile per esempio per la gestione della dimensione della matrice. [Moreno]

- `lattice_pk_decrypt(+CypherText,?PlainText,+PrivateKey,+PublicKey)`: decifra il CypherText utilizzando la coppia chiave pubblica e chiave privata, ed istanzia in PlainText il risultato della trasformazione inversa. Al pari del predicato di cifratura utilizza i predicati `asserta()` e `retract()` per la generazione di lemmi utili ad evitare la ripetizione di sottorami nell'albero di dimostrazine. [Moreno]

I predicati qui presentati consentono di operare su messaggi di lunghezza arbitraria, per questo motivo fanno uso di predicati ausiliari per la suddivisione del compito in operazioni più semplici. In particolare i predicati di cifratura e decifratura dividono il messaggio da decifrare in blocchi: in seguito all'applicazione di un predicato che opera su blocchi di lunghezza fissa i sotto-cyphertext vengono concatenati nuovamente ed istanziati nella variabile di ritorno.

5 Prove di query

Vengono qui riportate alcune prove di query, utili alla sperimentazione pratica con i predicati del progetto.

5.1 Funzione hash criptografica

Un'istanza della funzione hash implementata è definita nello spazio delle possibilità dalla matrice M: primo passo nella definizione e nell'utilizzo della funzione hash criptografica è la generazione di una matrice adatta.

A tale scopo si utilizza il predicato `matrix_random_gf(?Matrice,+Righe,+Colonne,+Gf)`, con alcuni vincoli sui parametri:

```
?- matrix_random_gf(M,200,8,2097152).
M = [omissis]

Yes
```

i vincoli dei predicati della sezione 2 corrispondono in questo modo: le righe della matrice è il parametro m, le colonne il parametro n e q è il campo Gf.

Una possibile invocazione completa del predicato in cui si verifica la proprietà di assenza di collisioni è la seguente, in cui due differenti stringhe di bit (in cui un solo bit viene variato) vengono codificate e stampate sul prompt.

```
?- matrix_random_gf(M,200,8,2097152).
   latthash(M,2097152,[0,1,1,0,1,0],HashUno),
   latthash(M,2097152,[0,1,0,0,1,0],HashDue),

M = [omissis]
Hash = [883291,1947550,1391636,221071,1580643,1034153,1183208,
       505000],
Hash2 = [2033960,1025484,1450139,1788920,318689,1053857,757559,
       461851]

Yes
```

La natura casuale della matrice M rende l'esecuzione qui riportata unica, ma dimostra la totale scorrelazione delle due immagini nel codominio benchè siano vicine nel dominio.

5.2 Cifratura a chiave pubblica

La cifratura a chiave pubblica prevede almeno tre distinte fasi: la generazione delle chiavi, la criptazione e la decriptazione.

La generazione delle chiavi prevede due parametri liberi scelti dall'utente: la dimensione del reticolo ed un numero utilizzato nella generazione dei vettori "errore", che determina in qualche modo la varianza all'interno delle singole componenti. La query seguente genera la coppia chiave pubblica e chiave privata di dimensione 8, ovvero due matrici quadrate 8x8:

```
?- lattice_pk_priv_gen(8,2,Private), lattice_pk_pub_gen(Private,Public).

Private = [omissis]
Public = [omissis]
```

Ottenuta la coppia chiave pubblica/chiave privata è possibile procedere con l'utilizzo dei predicati di cifratura e decifratura, per verificare il corretto funzionamento di mappatura ed inversione delle funzioni. La clausola di query completa per la generazione delle chiavi, cifratura e decifratura della frase "Corso di Logica II" è la seguente, dove viene utilizzato il predicato string_to_list di SWI-Prolog (che data una lista di codici ASCII ne produce una stringa di testo):

```
?- lattice_pk_priv_gen(8,2,Private),
   lattice_pk_pub_gen(Private,Public),
   lattice_pk_encrypt("Corso di Logica II", CypherText, Public),
   lattice_pk_decrypt(CypherText, PlainText, Private, Public),
   string_to_list(Testo,PlainText).

[omissis]
```

La natura probabilistica delle funzioni di generazione delle chiavi lasciano una probabilità non nulla che le basi corrispondenti a chiave privata e pubblica non siano adatte alle approssimazioni di decifratura dell'algoritmo *Babai Roundoff*, conseguentemente è possibile che in alcune esecuzioni la frase risulti cambiata in alcune lettere, o addirittura incomprensibile. Questo tipo di evento, nelle prove da noi effettuate, si è presentato solo di rado. Le differenti implementazioni Prolog, ed in particolare le routine di generazione dei numeri casuali, potrebbero inficiare il comportamento dei predicati in merito.

6 Test automatico

Pressochè tutti i moduli delle librerie di base sono sotto test automatico per mezzo di PlUnit [5], modulo dell'interprete SWI-Prolog: per ogni predicato è stato definito almeno un caso di test, cercando di testare ogni clausola nelle istanziazioni più significative. In corrispondenza di ogni modulo è stato creato un file sorgente `test-NOMEMODULO.pl`, ed è possibile eseguire automaticamente tutti i test dal file Prolog `test-all.pl` oppure in ambiente Unix digitando `make test` dalla root del progetto.

7 Conclusioni

Il linguaggio Prolog ha permesso di sviluppare le implementazioni in maniera snella ed efficace, ottenendo programmi compatti e facilmente interpretabili. Gli strumenti messi a disposizione dal progetto *SWI-Prolog* hanno permesso di costruire test automatici in modo semplice e chiaro, ereditando pragmaticamente una pratica Agile tipica del mondo procedurale.

Unica nota dolente riguarda le operazioni matriciali più complesse, che tuttavia avrebbero potuto essere implementate in linguaggio C sacrificando per esse l'impostazione dichiarativa in virtù di una maggiore velocità esecutiva. Rimane comunque il fatto che l'implementazione si è dimostrata per noi valida a fini didattici, per meglio comprendere gli algoritmi implementati ed i loro punti deboli.

Riferimenti bibliografici

[1] S.Halevi O.Goldreich, S.Goldwasser. Collision-free hashing from lattice problems. Technical Report TR96-056, Electronic Colloquium on Computational Complexity (ECCC), 1996.

[2] S.Halevi O.Goldreich, S.Goldwasser. *Public-key cryptosystems from lattice reduction problems*, volume 1294, chapter Lecture Notes in Comput.Sci., pages 112–131. Springer, 1997.

[3] Pldoc, documentation for Prolog. `http://www.swi-prolog.org/packages/pldoc.html`.

[4] Chiò Pivotal Condensation. `http://mathworld.wolfram.com/ChioPivotalCondensation.html`.

[5] Prolog Unit Tests. `http://www.swi-prolog.org/packages/plunit.html`.