Università degli Studi di Roma "La Sapienza"

Facoltà di Ingegneria

Tesi di Laurea in

Ingegneria Elettronica

Maggio, 2002

Gauss SAT Solver ed applicazioni alla Criptoanalisi del DES

Paolo Ottolino

Università degli Studi di Roma "La Sapienza"

Facoltà di Ingegneria

Tesi di Laurea in Ingegneria Elettronica

Sessione Invernale – Maggio, 2002

Gauss SAT Solver ed applicazioni alla Criptoanalisi del DES

Paolo Ottolino

Relatore Prof.ssa Luigia Carlucci Aiello

.....

Co-Relatore Prof. Ing. Fabio Massacci

.....

Indirizzo dell'Autore:
Paolo Ottolino
tel. +393478787629
ITALIA
e-mail: pottol@tiscalinet.it
www:

Prefazione

In questo lavoro viene presentato un *search algotithm* per il *propositional satisfiability problem* (SAT), finalizzato al trattamento di istanze contenenti logica affine, ovvero insiemi di clausole XOR, come ad esempio gli algoritmi di criptografia, quali il DES. La risoluzione di problemi espressi in CNF (Clausal Normal Form) è un problema NP-completo e dunque non è pensabile per essi un tempo di computazione polinomiale.

L'algoritmo qui proposto si basa sul DPLL con il trattamento delle clausole della logica affine come equazioni lineari sul campo $(Z_2, \oplus, \wedge)$, tramite la regola di Riduzione di Gauss, che richiede un tempo di risoluzione polinomiale, abbattendo, quindi drasticamente la durata della computazione su di essa. Oltre alla riduzione esplicita delle xor-clauses, la differenza fondamentale rispetto agli altri algoritmi basati sul DPLL, è la presenza contemporanea di ben 4 insiemi di clausole e di meccanismi di sincronizzazione e comunicazione fra di essi.

Si è cercato di inserire le principali tecniche algoritmiche e le linee guida sulla programmazione proprie della risoluzione per questo tipo di problemi tramite DPLL, evidenziate in letteratura negli anni, adattate alle ingenti differenze presenti in questa procedura rispetto all'originale e utilizzate in misura della applicazione su problemi di grosse dimensioni, quali la criptoanalisi.

Ne è risultato un algoritmo, sicuramente più complesso e meno immediato del DPLL, meno efficiente rispetto ad altri del suo genere su problemi di piccola entità, ma di sicuro impatto, rispetto ai concorrenti più moderni, su istanze più grosse quali la *criptoanalisi del DES*, appunto.

Ringraziamenti

Desidero ringraziare prima di tutti il mio co-relatore, il Prof. Ing. Fabio Massacci per la pazienza avuta con me durante il periodo di tesi, per l'aiuto e la guida verso l'obiettivo finale, specie nei momenti di perdita della "rotta".

Un particolare ringraziamento va all'Ing. Flavio Poletti che mi ha sopportato, specie negli ultimi mesi, senza il cui ausilio logistico, morale e "algoritmico" non sarei ora qui a scrivere questi ringraziamenti.

Inoltre, vorrei ringraziare tutti coloro che con la loro ricerca, le loro idee le loro energie, contribuiscono allo sviluppo dei linguaggi, dei protocolli, degli algoritmi, delle applicazioni etc. di "pubblico dominio", grazie interamente alle quali, questo lavoro ha visto la luce.

Infine desidero ringraziare tutte le persone che mi vogliono bene veramente, alle quali questo lavoro è dedicato.

Indice

Capitolo 1

Introduzione

1.1 Presentazione

Il Ragionamento Automatico sta rivestendo, negli anni, sempre maggiore importanza, anche a livello industriale. In particolare il problema della soddisfacibilità proposizionale (SAT) è ormai fondamentale in molte aree di applicazioni, tra le quali le verifiche formali(cfr. [Copty et al, 2001]), la pianificazione (cfr. [Kautz and Selman, 1996]), la criptoanalisi logica (cfr. [Massacci and Marraro, 2000]). Questo metodo di calcolo è molto apprezzato per via del suo alto livello di modularità, infatti non c'è bisogno di riscrivere la struttura di controllo per ogni nuovo tipo di problema, poichè i risolutori automatici sono basati sul ragionamento logico. Occorre più semplicemente un "traduttore", che codifichi il problemi in termini del sistema logico prescelto.

Negli anni si è assistito ad un abbattimento dei tempi di calcolo, in parte a causa del generale aumento di prestazioni dei calcolatori (ed il loro abbassamento di costo) ma soprattutto per il miglioramento delle procedure di ragionamento, ottenuto principalmente attraverso la *contaminatio* dei procedimenti di ricerca completa con le tecniche "d'informazione" della ricerca locale, ovvero le euristiche (cfr. [Freeman, 1995], [LI and Anbulagan, 1997]), unita all'inserimento di varie tecniche di ottimizzazione (cfr. [Zhang, 1997], [Bayardo and Schrag, 1997]).

Tuttavia la formulazione tradizionale dei problemi di inferenza logica in forma normale a clausole (Clausal Normal Form) può portare ad istanze di difficile soddisfacibilità.

Infatti, il SAT Problem è il primo problema dimostrato essere NP-Completo (cfr. [Edwards, 1996]) ed inoltre, sebbene sia sempre possibile convertire un problema reale in CNF, alcuni connettivi logici discendenti dai vincoli del problema (e.g. definizioni, XOR), propri della logica affine, generano istanze estremamente difficili da risolvere, codificati in questo formato (cfr. [Selman et al, 1997], [Li, 2000], [Warners and van Maaren, 1999]).

Viceversa la soluzione di un insieme di clausole XOR, viste come sistema lineare nel campo $(Z_2, \oplus, \wedge)$, usando opportune tecniche, come la riduzione di Gauss (cfr. [Baumgartner and Massacci, 2001]) richiede un tempo di calcolo polinomiale. Inoltre per ogni clausola XOR da n letterali sono necessarie, nella codifica CNF, ben $2^{(n-1)}$ clausole OR.

Partendo da queste osservazioni, sono stati creati solver che incorporano la eliminazione di Gauss come "Black Box", ovvero come procedura del tutto separata dal resto del ragionamento logico, tra cui: *EqSatz* (cfr. [Li, 2000]) che separa temporalmente le due computazioni, processando le XOR inizialmente, ed il "Dual Phase Algorithm" (cfr. [Warners and van Maaren, 1999]) che separa spazialmente le due computazioni.

Entrambi hanno prodotto risultati molto incoraggianti, soprattutto negli "artificial DIMACS

problems" (in particolare i Parity Problems), dove la parte di logica affine è preponderante[1].

In molti problemi del "Mondo Reale", come la criptoanalisi (cfr. [Massacci and Marraro, 2000]), le clausole XOR rappresentano una piccola parte del problema, nel qual caso un trattamento separato dei due insiemi di clausole XOR , anche per via delle numerose interdipendenze, non è sufficiente.

Si è cercato, quindi, basandosi sull'articolo scritto dai professori Massacci e Baumgartner (cfr. [Baumgartner and Massacci, 2001]) e prendendo ispirazione dai problemi criptoanalitici e da Benchmark come i "DIMACS Parity Problems", di creare un solver che ragionasse anche in logica affine, secondo una nuova architettura, più omogenea, separando i tipi di clausole in insiemi distinti ed al contempo facesse comunicare questi ultimi, tramite le nuove *Unit-Clauses* o le nuove Assegnazioni generate, in modo da far condividere i progressi compiuti ai vari insiemi del sistema.

1.1.1 Sicurezza e Crittografia

In seguito all'inarrestabile sviluppo delle telecomunicazioni, quasi sempre veicolate per mezzo di supporti e protocolli condivisi ma originariamente non pensati per la difesa da eventuali intrusioni e manomissioni (basti pensare all'IPv4 o ai moderni protocolli di comunicazione Wireless, come l'IEEE 802.11), si fa sempre più pressante la richiesta di *layers* (e.g. Secure Socket Layer, Secure HTTP etc.), posti tra i protocolli di comunicazione e le applicazioni, che assicurino protezione e riservatezza dei dati. Questi "strati" di sicurezza agiscono nascondendo, ai soggetti non autorizzati, le informazioni attraverso l'unica arma disponibile su un canale condiviso: la codifica attraverso chiavi (codici sconosciuti ai non autorizzati) , ovvero la criptografia (dal greco: $\chi\rho\upsilon\pi\tau o\varsigma + \gamma\rho\alpha\phi\omega$ scrivo in modo nascosto).

Il funzionamento base di un algoritmo criptografico è il seguente: dato il testo da trasmettere P (il "plaintext"), la chiave K (conosciuta solo dai soggetti coinvolti nella comunicazione), viene effettivamente trasmesso C (il "ciphertext"), calcolato dal computer del mittente come $C = Crypt_K(P)$. Al computer del destinatario non resta che calcolare $P = Crypt_K^{-1}(C)$, per rendere "chiaro" al suo utilizzatore il messaggio, ovviamente, conoscendo sia la chiave K, sia l'algoritmo applicato $Crypt()$.

L'utilizzo della crittografia, anche se in formati molto più semplici di quelli attualmente in uso, risale ai tempi degli antichi egizi (tramite i geroglifici) e degli antichi romani (la *"codifica di Giulio Cesare"* viene ancora oggi comunemente utilizzata per semplici scopi di segretezza). Si può intuire, quindi, la gran quantità di e varietà di algoritmi ideati nei secoli, il cui denominatore comune, specie nelle procedure più moderne, è la presenza di una funzione che effettui l'operazione $C_{i,j} = P_i \oplus K_j$ per qualche i e j.

Viceversa lo studio delle proprietà matematiche generali della codifica tramite chiavi è stato effettuato soprattutto durante l'ultimo secolo, producendo risultati di rilevante interesse applicativo.

Senza entrare nell'argomento, per il quale si rimanda a testi specifici, possiamo affermare empiricamente che per ottenere un elevato livello di sicurezza, l'algoritmo di crittografia $Crypt()$ deve essere:

- ***pubblico***: la sicurezza non sta nel nascondere la procedura poichè prima o poi qualcuno (tipicamente proprio chi non dovrebbe) ne verrebbe sicuramente a conoscenza; è, invece, preferibile che sia di dominio pubblico, di modo che chiunque ne possa scovare debolezze e/o errori

[1] Nei Parity Problems le clausole XOR rappresentano ca. il 90% del tutto, una volta tradotti i problemi in logica affine.

- ***robusto***: l'efficacia deve risiedere nelle proprietà della chiave (prima fra tutte la lunghezza), che l'algoritmo deve essere in grado di sfruttare; ovvero, fornendo un ordine di grandezza, a chi non conosce la chiave deve occorrere un tempo paragonabile a $O(2^{lenght(K)})$ per rompere il sistema. Si può infatti dimostrare che, con l'eccezione del caso in cui la chiave sia lunga quanto il messaggio (circostanza evidentemente troppo costosa ed ardua da mettere in pratica), un sistema crittografico è sempre forzabile.

1.1.2 L'importanza della Criptoanalisi

La criptoanalisi (dal greco $\chi\rho\upsilon\pi\tau o\varsigma + \alpha\nu\alpha\lambda\upsilon\omega$ sciolgo ciò che è nascosto) studia i sistemi di crittografia, proprio per verificarne l'efficacia e la robustezza, generalmente utilizzando un approccio di tipo sperimentale ma anche tramite lo studio delle proprietà matematiche presenti in ciascuno di essi.

Infatti la genesi e le caratteristiche di alcuni dei più usati algoritmi crittografici, fanno pensare che esistano, volutamente o meno, meccanismi che consentano agli ideatori degli stessi di risalire, tramite il solo ciphertext alle informazioni originali, senza eccessivo sforzo. Ovvero si ipotizza la presenza delle cosiddette "BackDoors" e "Universal Keys".

Altri algoritmi crittografici, invece, se da un punto di vista teorico appaiono molto robusti, alla verifica pratica risultano assolutamente deboli, per via di alcune subdole ed indesiderate proprietà matematiche, di cui non ci si era accorti in fase di progettazione.

La verifica matematica di queste proprietà, tuttavia, è compito tutt'altro che facile, mentre la scoperta empirica a volte risulta più agevole (e.g. la verifica sperimentale, in occasione dell'attacco al DES portato dagli *RSA Laboratories*, ha rivelato la presenza di una unica chiave per ogni coppia plaintetx, ciphertext, ma la dimostrazione matematica formale ancora non è stata effettuata).

D'altronde anche la criptoanalisi prevede metodologie ben lungi da essere di facile attuazione pratica, richiedendo capacità computazionali copiose. Nel seguito sono elencate alcune delle tecniche più utilizzate

Exaustive Search

Si basa sulla enumerazione completa ed il test di tutte le possibili chiavi. Generalmente ci si serve di macchine, appositamente sviluppate, che implementano in hardware l'algoritmo in esame. Un esempio notevole è la macchina realizzata agli RSA Laboratories (cfr. http://www.cryptography.com/des/). Il costo di queste apparecchiature è però considerevole.

Linear-Cryptoanalysis

Questa tecnica è stata inventata da Mitsuru Matsui (cfr. [Matsui, 1993], [Matsui, 1994], [Biham, 1994]) proprio per tentare di forzare il DES. Si basa sul concetto di linearizzazione di alcuni blocchi del cifratore (le S-Boxes) e sulla derivazione di una probabilità della approssimazione effettuata. Effettuando l'analisi di un grosso numero di coppie PlainText, CipherText, si ottengono alcuni bit della chiave. Il resto della ricerca procede in modo esaustivo.

Lo svantaggio fondamentale sta proprio nel grosso numero di coppie richieste: ne occorrono circa 2^{47} nel caso del DES.

Differenzial-Cryptoanayisis

Introdotto da Eli Biham e Adi Shamir nel 1991, si basa anch'esso sulle proprietà probabilistiche, applicate però alle differenze riscontrate su 2 PlainText, P_1 e P_2 calcolati "a ritroso" da 2

CipherText C_1 e C_2 di cui si sia precedentemente calcolata la differenza $C_1 \oplus C_2$. Anch'esso prevede un grosso numero di coppie PlainText, CipherText.

Linear-Differential Cryptoanalysis

Ottenuta combinando tecniche insieme lineari e differenziali, consente di utilizzare molte meno coppie PlainText e CipherText (nel caso del DES occorrono 512 coppie). I tempi di computazione restano però lunghi.

1.1.3 Logical-Cryptoanalysis

Poichè ogni algoritmo di crittografia tratta i bit del PlainText uno ad uno, è possibile interpretare i bit del PlainText P, del CipherText C e della Chiave K come variabili proposizionali, che assumono il valore *true* se il corrispondente bit è 1 e *false* altrimenti. Ovviamente è necessario "tradurre" le proprietà dell'algorirmo crittografico in esame in una formula della logica proposizionale $F(P, K, C)$, le cui clausole corrispondono ai vincoli imposti dal cifratore.

Nel caso di attacco in cui si conosca solo il CipherText, la formula viene semplificata in base alle assegnazioni effettuate su C, e dunque avremo $F = F(P, K)$.

Nel caso di attacco in cui si conosca anche il PlainText, la formula verrà semplificata anche in base alle assegnazioni su P, e dunque $F = F(K)$.

La ricerca della chiave corrisponde alla ricerca di un modello per la formula F, ovvero corrisponde ad un tipico SAT problem.

Inoltre, nella codifica dei vincoli dell'algoritmo in esame, nella logica, per non far esplodere la lunghezza delle clausole, generalmente vengono introdotte delle variabili "interne" al cifratore, il cui valore non risulta direttamente utile nell'atto della forzatura dello stesso, ma che possono rivelarsi comode nell'atto di analizzare proprietà più interne all'algoritmo, attraverso l'analisi della interdipendenza fra le variabili presenti nella soluzione, come

- esistenza di una chiave universale
- esistenza di backdoors
- reale applicazione delle auspicabili proprietà di Shannon per gli algoritmi di crittografia: *confusion* e *diffusion*
- reale utilità di alcuni passaggi procedurali del cifratore

Portare un attacco di criptoanalisi logica consiste nel generare le formule della logica proposizionale relativamente all'algortimo di interesse e successivamente cercarne un modello tramite un SAT Solver. Sfortunatamente, però, i solver esistenti, ad oggi, non riescono a venire agevolmente a capo di questo tipo di problemi (cfr. [SATLib site, 2002] e [Massacci and Marraro, 2000]).

1.1.4 Un SAT Solver per la Logica Affine

Lo scopo di questo lavoro è, quindi, ottenere un solver efficiente, principalmente su problemi espressi in *Affine Logic* (quali le "traduzioni" in logica di procedimenti di criptografici), ovvero orientato a tutti quei problemi in cui sia determinante, ai fini di una ricerca più rapida della soluzione, l'attenzione speciale posta nei confronti delle componenti espresse in formato di equivalenza, del problema.

Per la parte crittoanalitica, l'attenzione è stata volta al DES, in quanto:

1. esiste già una efficiente procedura (cfr. [Massacci and Marraro, 2000]) di traduzione in logica affine, sia generale, sia di istanze generate da coppie PlainText, CipherText.
2. attualmente è considerato una delle sfide principali per i SAT solver
3. risulta ancora oggi, nonostante i suoi ventisette anni di età, largamente utilizzato, solitamente in modo reiterato (in numero dispari[2] di volte: Triple-DES, Penta-DES) sullo stesso blocco di dati, ma con chiavi diverse.
4. sembra ormai assodato che se da un lato, come ho detto in precedenza, dato un plaintext P ed un chipertext C $\exists! K : DES_K(P) = C$, benchè la dimostrazione formale ancora non sia stata trovata, dall'altro solo pochi bits della chiave influenzino veramente la creazione del CipherText partendo dal PlainText e dunque esistano delle cosiddette "Backdoors".

Per ottenere dei confronti, rispetto ad altri solver, si è pensato di lavorare principalmente sui *DIMACS Parity problems,* in quanto:

- ogni istanza contiene un elevato numero di equivalenze, traducibili in clausole XOR
- rappresentano da anni un tipico "*benchmark*", su cui si sono confrontati in molti
- le istanze da 8 e 16 bit sono molto più semplici da risolvere rispetto al DES, consentendo di avere risultati "rapidi"

Proprietà Richieste al Solver

Per quanto detto, un solver siffatto doveva godere almeno delle seguenti proprietà:

- Risoluzione ottimizzata delle clausole della logica affine (le clausole XOR), poichè molto frequenti in alcuni problemi reali, e poichè la risoluzione di esse richiede tempi polinomiali e non esponenziali.
- Possibilità di generare soluzioni per modelli, mostrando esplicitamente le dipendenze tra le variabili.
- Possibilità di generare tutte le soluzioni (qualora il primo modello scoperto non fosse sufficiente).
- Orientamento alle istanze del SAT Problem complicate e legate al mondo reale

1.2 Risultati

Il prodotto di questo lavoro è un SAT Solver (il Tame) sicuramente particolare rispetto ad altri, come esso, basati sul DPLL, proprio per cercare di adattarsi ai requisiti precedentemente esposti.

Tame

In questo paragrafo vengono elencate le peculiarità del solver, rimandando a più avanti nel testo per una loro descrizione approfondita.

[2] si può dimostrare che l'applicazione di un numero di iterazioni pari non apporta vantaggi rispetto alla applicazione singola

- **Orientamento al Trattamento della Logica Affine**: Il Solver , è stato dotato di un meccanismo di riduzione delle clausole XOR attraverso il meccanismo della "Triangolazione di Gauss". Questo calcolo viene eseguito durante tutta la computazione, tenendo costantemente separate le clausole XOR da quelle OR.
- **Trattamento Esplicito delle Assegnazioni**: Per incrementare la velocità, viene usata oltre alla "classica" Unit Rule, anche una semplificazione in base alle assegnazioni tra variabili presenti, con lo scopo di "estromettere" le variabili definite in funzione di altre dal calcolo, tramite un meccanismo analogo alla riduzione di Gauss, ma applicabile a tutti gli insiemi di clausole.
- **Clauses-Set Multipli**: Per via delle precedenti proprietà, è stato necessario definire 4 insiemi fondamentali di clausole:
  - clausole OR
  - clausole XOR
  - assegnazioni
  - risolvendi delle clausole XOR (tramite Gauss)
- **Sincronizzazioni fra i Set**: Poichè si voleva che le varie regole di inferenza fossero applicate in modo parallelo su tutti gli insiemi facendo propagare le semplificazioni da un insieme ad un altro, grossa importanza ha rivestito la "comunicazione" fra i set, con particolare enfasi nel passaggio delle clausole da insiemi a più lenta computazione (es. insieme delle clausole OR), verso altri più "pregiati".
- **13 Regole d'Inferenza**: Dato il numero e le peculiarità dei set istanziati, nonchè la necessità di comunicazione fra insiemi, è stato necessario definire ben tredici regole d'inferenza distinte, per la realizzazione completa del procedimento di calcolo.
- **Heuristic Problem Driven**: Si è cercato, nei limiti del possibile, nel solo caso del DES, di inserire delle euristiche ispirate alla struttura del problema, che facessero da "filtro aggiuntivo", rispetto a quelle base.
- **Utilizzo diffuso di LookAhead**: Per cercare di limitare le visite, a nodi dell'albero computazionale, che portassero a contraddizioni, si è fatto largo uso della tecnica dell lookahead. Nonostante sia stata lasciata facoltà, all'utilizzatore del solver, di decidere se usare lookahead ad uno o due livelli, gli esperimenti hanno ampiamente dimostrato la convenienza nell'usare la profondità maggiore, quanto l'aumento di efficienza, nel solo caso del DES, eliminando l'uso della lookahead durante la normale computazione.
- **Semplificazione Iniziale**: Prima di entrare nella ricerca della soluzione tramite splitting, la formula viene a lungo semplificata. Questa fase può durare a lungo, anche più della metà del tempo complessivo di computazione, ma porta ad una riduzione di qualche ordine di grandezza del numero di nodi visitati.

1.2.1 Esperimenti

Le verifiche sperimentali di comparazione con i due solver: *Posit* (cfr. [Freeman, 1995]), particolarmente veloce nei problemi di piccole dimensioni e *Simo* (cfr. [Giunchiglia et al, 2000]), ingegnerizzato dalla Intel ed alla base dei *model checker industrial* **Thunder** della Intel (cfr. [Copty et al, 2001]) e **NuSMV** dell'IRST e Carnegie Mellon University, sono state condotte sui DIMACS Parity Problems, in modo da avere un utile confronto.
Empiricamente si è dimostrato che:

- su problemi di piccole dimensioni (e.g. par8): nonostante le euristiche più semplici, il Posit è più veloce, risultato da additare principalmente alla semplificazione iniziale; a media distanza si colloca il Simo, che non effettua questa operazione, ma fornito di euristiche migliori e una struttura dati altamente efficiente; e da ultimo il tame, rallentato soprattutto a causa del frequente impiego della *lookahead*, anche nella semplificazione iniziale

- su problemi di medie dimensioni (e.g. par16): Posit rimane sempre più veloce, ma le differenze fra i tre si accorciano notevolmente

- su problemi più difficili (e.g. i *parity32*), grazie anche all'estensivo impiego della funzione lookahead, il tame guadagna terreno anche nei confronti del Simo, sicuramente a causa della semplificazione iniziale (che di fatto impiega quasi metà del tempo totale di computazione) visitando un numero di nodi esiguo rispetto ai concorrenti. Viceversa il Posit non riesce a risolvere in due giorni.

Altri esperimenti sono stati effettuati sul DES ed hanno invece dimostrato che:

- conviene non invocare la funzione lookahead() durante la normale computazione poichè non vengono scoperte contraddizioni sufficienti a giustificarne il costo computazionale

- conviene eseguire lo split() selettivo, su un solo tipo di variabile, ed in particolare sulle variabili 'X', riducendo il tempo di computazione, in media, del 15%

Capitolo 2

Il Problema

In questo capitolo viene inquadrato lo scenario da affrontare, ovvero la progettazione di un SAT solver volto alla applicazione su problemi contenenti logica affine, con particolare riferimento ai DIMACS Parity Problems ed alla criptoanalisi logica.

2.1 SAT Solvers

In questo paragrafo viene introdotto il concetto di SAT Solver e, senza la minima pretesa di esaurire l'argomento, viene illustrata una rapida classificazione dei principali *solvers* noti in letteratura, con lo scopo di mostrare le motivazioni che sono alla base della scelta di derivare l'algoritmo, in particolare, dal DPLL.

2.1.1 Constraints Satisfaction Problem

Il CSP è una categoria speciale di problemi, in cui occorre soddisfare alcune proprietà di vincolo, addizionali rispetto ai requisiti basilari dei problemi in generale. In un CSP occorrono, dunque, i seguenti soggetti:

- **variabili**: $\{x_1, ..., x_n\}$ entità presenti nella istanza di cui si vuole determinare il valore. Ognuna di esse appartiene ad un dominio (*discreto* o *continuo*) noto a priori. Una assegnazione è un vettore di n valori ognuno assegnato ad una variabile (secondo dominio di appartenenza)
- **domini**: $\{D_1, ..., D_n\}$ ognuno di essi contiene i possibili valori cui ogni variabile può essere istanziata. Possono essere di natura discreta o continua
- **constraints**: $\{C_1, ..., C_m\}$ sono i vincoli tra le variabili presenti nel problema, da rispettare nel ritrovamento dela soluzione. Da un punto di vista matematico sono delle vere e proprie relazioni tra elementi (elementi del dominii), ovvero $C_i \subset (D_1 \times D_2 \times ... \times D_n)$. Possono essere:
  - *absolute* inviolabili, il mancato soddisfacimento invalida la soluzione potenziale
  - *preference* la cui soddisfazione in una soluzione la rende preferenziale
- **goal**: determinare se esiste una o più assegnazioni per le variaribili tale che tutti i constarints siano soddisfatti.

Cenni alla Classificazione delle Principali Strategie di Risoluzione

I CSP sono estremamente comuni (basti pensare alla colorazione dei grafi, il problema delle n regine, la risoluzione di un sistema di equazioni o la verifica di soddisfacibilità di una formula booleana). Alcuni di questi sono soddisfacibili con algoritmi "diretti" che possono giungere a soluzione in tempo polinomiale, altri (la maggior parte) sono NP-completi e dunque non ammettono una soluzione in un tempo polinomiale; per questi generalmente si adottano algoritmi basati sulla ricerca nello spazio delle soluzioni.

In pratica vengono usati due generi di approcci di base[1] alla ricerca della soluzione:

- **constrained algorithm**: tentano di soddisfare tutte le condizioni poste, magari effettuando le assegnazioni una ad una, variabile per variabile. Generalmente si basano su una sorta di **backtracking**, quindi una sorta di visita di un albero delle soluzioni, opportunamente velocizzato attraverso:
  - *heuristics* funzioni che, dipendentemente dal problema, calcolano, in modo approssimato, la sequenza di assegnazioni che renda minima la ricerca della soluzione e dunque il costo computazionale
  - *constraint propagation* una sorta di ricerca in avanti (lookahead), con lo scopo di rivelare assegnazioni fallaci, prima ancora di effettuarle
- **unconstrained algorithm**: algoritmi "informati" che conoscono la struttura del problema. Generalmente si basano su una **local search** della soluzione, attraverso la ricerca di assegnazioni plausibili, misurando la "distanza" dalla soluzione attraverso:
  - *heuristics*[2] funzioni, generalmente ideata a partire da un tipo di problema più semplice di quello dato, che calcola la vicinanza di una particolare assegnazione alla soluzione, sulla base della quantità e qualità delle condizioni non soddisfatte.
  - *path cost* funzione che calcola il costo computazionale del passaggio da una assegnazione ad un'altra

2.1.2 SAT Problem

Il SAT (SATisfiability problem), sebbene sia stato formulato precedentemente al CSP, può esserne inquadrato come un caso particolare, ovvero il caso in cui il dominio delle variabili è unico ed è quello della Logica Booleana.

Costituisce uno dei più importanti problemi nel campo della Intelligenza Artificiale.

Molti problemi della "vita reale" vengono risolti quotidianamente grazie a questo meccanismo, come ad esempio la verifica dei circuiti elettronici, verifiche formali etc. D'altronde la traduzione in CNF, e dunque in logica booleana, nel caso di problemi complessi (magari NP-completi) se, da un lato, magari può comportare l'aumento del numero delle variabili e dei vincoli presenti, dall'altro significa doversi confrontare con un CSP :

[1] Le "contaminazioni" con elementi di un genere nell'altro sono all'ordine del giorno, nel caso dei moderni algoritmi di risoluzione del CSP ed in alcuni casi rappresentano una scelta vincente nel miglioramento delle performance nella strategia di ricerca.

[2] Occorre osservare esplicitamente la diversa funzionalità, ai fini teorici, del concetto di euristica, per gli algoritmi constrained e gli unconstrained: nel primo caso sono delle funzioni che hanno lo scopo di velocizzare la computazione, con una scelta sbagliata il calcolo rallenta, anche di diversi ordini di grandezza, senza tuttavia lenire la possibilità teorica di giungere ad una soluzione; nel secondo caso si tratta di un componente indispensabile dell'algoritmo, una definizione sbagliata può portare all'impossibilità di risolvere il problema.

- i cui domini sono il più semplice possibile (Z_2)
- i cui valori delle variabili possono essere rappresentati dalle particelle di informazione elementari degli elaboratori: i bit

2.1.3 Cenni alle Principali Strategie di Risoluzione del SAT Problem

Con la esclusione di:

- 2-SAT: problemi in CNF che prevedono solo clausole di lunghezza massima 2
- Horn-SAT: problemi in CNF che contengono solo clausole di Horn (in cui al massimo un solo letterale occorre non negato per ogni clausola)

per i quali esistono algoritmi con tempo di risoluzione polinomiale, per tutti le altre categorie si tratta di problemi NP-completi, per la soluzione dei quali, generalmente, si ricorre ad algoritmi di ricerca nello spazio delle soluzioni. Analogamente al CSP abbiamo la divisione tra procedure di calcolo *constrained* e *unconstrained*.

Constrained Algorithm for SAT Problem

Generalmente le procedure si basano sul rimpiazzamento ricorsivo della formula corrente con una o più formule la cui soluzione implica la soluzione della formula originale. Solitamente, vengono utilizzate, ad ogni passo dell'algoritmo, le seguenti regole di controllo, per bloccare la computazione:

1. se $\exists C \in Clauses : C = \emptyset \Rightarrow$ la formula non ha soluzione
2. se $Clauses = \emptyset \Rightarrow$ la formula ha soluzione

Resolution (Robinson, 1965) si basa sul principio di risoluzione, proposto a livello teorico da Baker nel 1937 e ripreso successivamente da Robinson, ovvero:

$(\alpha \vee \beta) \wedge (\neg\alpha \vee \gamma) \Rightarrow (\beta \vee \gamma)$ con $\beta \neq \gamma$

Per ogni coppia di clausole si generano tutti i possibili risolventi. Il tempo di computazione, generalmente tende ad un tempo esponenziale.

Sono state effettuate alcune ottimizzazioni e modifiche nelle varie realizzazioni dell'algoritmo, tra cui:

1. Subsumption se $\exists C, D \in Clauses : C \subset D \rightarrow$ delete (D)
2. Pure Literals se $\exists a \in literals : \nexists C \in Clauses : \neg a \in C \rightarrow \forall D \in Clauses : a \in D, delete(D)$
3. Davis Putnam (1960) per ogni variabile, dopo aver effettuato le necessarie semplificazioni, calcola tutti i possibili risolventi di quella variabile, cancellando tutte le clausole che la menzionano. Ad ogni step questo genera un sotto-problema con una variabile in meno ma con un numero di clausole che potenzialmente aumenta al quadrato.

Generalmente però, si genera una vera e propria "esplosione" della occupazione della memoria, con conseguente calo di efficienza.

La *resolution* viene applicata soprattutto nei problemi di Logica del Primo Ordine.

Backtracking si basa sullo *splitting*, ovvero, durante ogni iterazione, si seleziona una variabile e si generano due sotto-formule assegnando i due possibili valori, true e false, alla variabile selezionata. La principale differenza tra un algoritmo e l'altro, di questo genere è rappresentata dal modo con cui viene scelta la variabile per lo split, ovvero dalla cosiddetta *Branching Strategy*, che determina, più di ogni altro fattore, il costo computazionale dell'algoritmo e dalle regole di inferenza adottate. Il flusso di controllo di un algoritmo per il SAT, basato sul backtracking, viene spesso rappresentato da un albero binario, la cui radice corrisponde alla formula originale, ogni nodo rappresenta una sub-formula ed ogni foglia corrisponde ad una formula le cui variabili sono completamente assegnate, ovvero è possibile dare una soluzione.

Anche sul backtracking sono state effettuate ottimizzazioni e/o modifiche, tra cui:

1. Backjumping: quando viene generato un bottom (insoddisfacibilità) l'algoritmo cerca la variabile la cui assegnazione ha generato l'inconsistenza e cambia l'assegnazione solo di essa, saltando vari livelli irrilevanti sull'albero di computazione.
2. Lookahead: filtro che "taglia" i rami dell'albero che genererebbero inconsistenza prima di raggiungerli. E' anch possibile usare un lookahead a "2 livelli", ovvero qualora, durante il preprocessamento per analizzare l'eventuale generazione di inconsistenza, non si generassero bottom in entrambe le assegnazioni per una variabile, aggiunge le clausole nuove, comuni ai due rami, eventualmente generatesi.
3. Search Rearrangement: al momento della estensione della soluzione parziale (ovvero alla scelta della variabile sulla quale eseguire lo split) si sceglie la assegnazione di variabile che massimizza i vincoli della formula. In questo modo vengono esplorati prima i nodi che comportano un numero minore di successori.
4. Learning: aggiunta di determinati vincoli aggiuntivi tra le variabili, sotto forma di clausole, desunti durante l'esplorazione dello spazio delle soluzioni.

Unconstrained Algorithm for SAT Problem

La maggior parte delle procedure di questo tipo si basa su una *local search* nello spazio delle assegnazioni, valorizzando in modo completo tutte le variabili ogni volta, e stimando la bontà della scelta operata, mediante l'utilizzo di opportune:

1. **funzioni-obiettivo** che calcolano la "lontananza" dalla soluzione attraverso il calcolo del numero delle clausole OR non soddisfatte o, nel caso di procedimento sulla formula inversa, il numero di clausole AND soddisfatte.
2. **funzioni di costo** che calcolano il costo computazionale, in termini del numero di variabili di cui modificare il valore, per spostarsi da una possibile soluzione ad un'altra.

il calcolo, quindi, procede da un "punto" dello spazio di ricerca ad un'altro, alla ricerca della minimizzazione della funzione-obiettivo. Una sorta di ricerca di "ottimo locale".

Sebbene questo tipo di algoritmi sia efficiente, principalmente per le due ragioni:

1. effettua all'inizio una completa assegnazioni di valori di verità per tutte le variabili occorrenti, limitando l'attenzione ad un singolo "punto" nello spazio delle soluzioni

2. si muove da quel punto solo dopo aver testato l'effettivo miglioramento che si avrebbe muovendosi verso un altro, effettuando delle "riparazioni" attraverso la funzione-obiettivo

purtuttavia, in genere, può accadere che:

- nel "vagabondare" tra le soluzioni si capiti in un minimo locale, dal quale non si riesca più ad uscire; per evitare la qual cosa solitamente viene incluso nell'algoritmo un meccanismo di ripartenza "random"
- la funzione-obiettivo (ovvero la euristica) non sia stata congegnata in modo adeguato, anche perchè profondamente dipendente dal problema specifico, con il risultato di non riuscire ad "avvicinarsi" più di tanto alla soluzione

L'effetto finale è che possa venir meno la completezza (la capacità di scoprire tutte le soluzioni) onde su formule per le quali esistano poche soluzioni (e.g. il DES a 16 giri per il quale dovrebbe esistere una ed una sola soluzione) la ricerca potrebbe fallire.

Scelta dell'algoritmo

In considerazione della analisi fatta, sono stati scartati gli algoritmi di tipo "local search", poichè non garantivano la completezza ed anche le procedure basate sulla regola di risoluzione per non rischiare di consumare eccessiva memoria.

La scelta quindi è ricaduta sugli algoritmi basati sul meccanismo del backtracking e tra essi si è optato per il DPLL, per alcune peculiarità, che lo rendono preferibile rispetto a molti altri analoghi sistemi.

2.1.4 DPLL

Introdotto nel 1962 da Martin Davis, G. Logemann, Donald W. Loveland, è la procedura per SAT, che vanta il più alto numero di implementazioni, modifiche, miglioramenti, ottimizzazioni poichè gode di alcune fondamentali proprietà, non presenti in altri algoritmi:

- **efficienza spaziale**: basandosi sul *Depth First Search*, necessita delle sole risorse di memoria occorrenti ad allocare un branch alla volta
- **esiguo numero di regole**: generalmente, le regole di inferenza presenti sono in quantità esigua
- **adattabilità d'implementazione** : per via della linearità concettuale dell'algoritmo originale, è possibile inserire o adattare gli elementi procedurali, alle proprie esigenze
- **completezza**: realizza una ricerca completa nello spazio delle soluzioni

Tutti i più efficienti solver basati su ricerche sistematiche nello spazio delle soluzioni sono basati sul DPLL, avendo inserito, come prima modifica sostanziale, un'adeguata euristica per la scelta della prossima variabile su cui eseguire lo split. Solo per esempio citiamo alcuni che rappresentano l'attuale stato dell'arte:

1. Sato (cfr. [Zhang, 1997]) progettato per essere efficiente nei problemi del mondo reale, attraverso il meccanismo del "learning" e un'euristica basata sul rilevamento delle clausole di Horn (che come ricordiamo sono risolvibili in tempo polinomiale)

2. Satz (cfr. [LI and Anbulagan, 1997]) molto efficace sui *random generated tests*
3. RelSat (cfr. [Bayardo and Schrag, 1997]) progettato per essere efficiente nei problemi reali, attraverso il meccanismo del "learning", usato specie durante i calcoli dell'euristica.
4. EqSatz (cfr. [Li, 2000]) derivato dal Satz, con l'aggiunta della risoluzione delle clausole XOR contenute nel problema, all'inizio del processamento dello stesso. Molto efficace, ovviamente, su problemi in cui compaiono molte equivalenze fra proposizioni, come i *parity problems*.
5. Posit (cfr. [Freeman, 1995]) pensato fondamentalmente per i *random generated tests*, ma la cui grossa utilità sta nel aver tracciato delle linee guida imprescindibili per chiunque si accinga a progettare un solver derivato dal DPLL
6. Simo (cfr. [Giunchiglia et al, 2000]) che riassume le parti migliori dei i solver precedentemente citati, coniando un solver veramente efficace e fornendo la possibilità di confrontare gli algoritmi di cui sopra su una implementazione unica, quindi in modo oggettivo. Ingegnerizzato dalla Intel Corporation, alla base degli *Industrial Model Checker* Tunder e NuSMV
7. Zchaff (cfr. [Moskewicz et al, 2001], [Zhang et al, 2002]) che migliora circa del doppio le performance finora ottenute, intervenendo sul meccanismo del learning.

Algoritmo Originale e Basi Teoriche

L'algoritmo originale si basa su tre regole della logica proposizionale:

1. Unit Clause Rule: se $\exists C \in Clauses : C = \{a\} \Rightarrow a = \top, propagate(a \leftarrow \top)$ in C, la cui dimostrazione è evidente.
2. Pure Literal Rule: se $\exists a \in proposition : \nexists C \in Clauses : \neg a \in C \Rightarrow \forall C \in Clauses$, if $a \in C \rightarrow$ delete (C) e $a = \top$. Infatti si può dimostrare che, posto:

   X un insieme di clausole

   A un pure literal

   $X^1 = X \setminus Y : Y = \{C \in X : A \in C\} \Rightarrow X^1$ possiede un modello $\Leftrightarrow X$ ne possiede uno.
3. Split: siano

   $X =$un insieme di clausole della logica proposizionale

   $A =$un letterale

   onde possono succedere 2 cose (non mutuamente escludentisi, in generale):

   $X \cup \{A\}$ ha un modello

   oppure

   $X \cup \{\neg A\}$ ha un modello

Queste regole sono quindi state combinate, producendo la seguente procedura:

```
bool
DPLL (Clauses-Set C) {
```

```
  if (C is empty) return true;
  else
    if (exists c in C: c is empty) return false;
  else {
    Pure_Literals (C);
    Unit_Clauses (C);
    proposition a;
    Pick (a from proposition);
    return DPLL (C[a=false]);
    return DPLL (C[a=true]);
  }

}
```

Impoved DPLLs: le Euristiche ed altre Tecniche

Nell'algoritmo originale, la funzione *pick()* prendeva la variabile su cui eseguire lo *split*, senza particolare criterio, praticamente la prima variabile non assegnata. D'altronde è possibile sostituire la pick() con qualsiasi altra funzione si desideri, senza alterare la validità generale dell'algoritmo; questo rappresenta una grossa libertà d'intervento sulla procedura, in pratica.

La maggior parte dei miglioramenti del DPLL, quindi, sono stati volti, principalmente alla ridefinizione, più o meno efficace, della funzione *pick()*, di scelta della variabile, attraverso le Euristiche.Sebbene di queste parleremo più diffusamente nel capitolo dedicato alla architettura del solver da me sviluppato, voglio introdurne una classificazione, in base al tipo di calcolo effettuato:

1. **Lexicographic Measurement**: sono le euristiche che sono state utilizzate per prime storicamente. Basano la loro strategia su misure di frequenza dei letterali nelle clausole e sulla lunghezza di queste ultime. Le più utilizzate sono:

   - **MOMS** (Maximum Occurrence in Minimum clause Size), il cui acronimo si deve a Pretolani (cfr. [Freeman, 1995]), si tratta probabilmente della meno accurata, ma della più rapida da calcolare.
   - **Jeroslow-Wang** (cfr. [Jeroslow and Wang]) la più accurata delle Lexicographic, cerca di dare una misura dei vincoli in cui è implicata ogni proposizione

2. **LookAhead-Based** (cfr. [LI, 1999]): sicuramente più pesanti da un punto di vista computazionale, generano misure ben più attendibili, dal momento che effettuano un split di prova su ogni variabile, calcolando l'effettiva semplificazione della formula, ed eventuali incongruenze che derivano da alcune assegnazioni. Si può anche utilizzare il cosiddetto "Double Level LookAhead" che prevede, inoltre, l'inserimento delle clausole comuni ai due rami di split.

Altri tecniche sperimentate sono:

- **BackJumping** (*Dependency-Directed-BackTracking*), come si evince dal nome, viene effettuato il BackTracking solo sulla assegnazione che ha causato la contraddizione. Ha tuttavia lo svantaggio di obbligare l'algoritmo ad occupare memoria per tenere le informazioni di dipendenza.

- **Learning** vengono aggiunte alla formula delle condizioni aggiuntive, sotto forma di clausole, che rappresentano la negazione di alcune situazioni, ricorrenti nel calcolo, di insoddisfacibilità del problema dato. Va prestata particolare attenzione nell'impedire la crescita a dismisura del numero di clausole, generalmente realizzato tramite il meccanismo di **forgetting**.

- **Partitioning** (cfr. [Park and Gelder, 1995]) attraverso la rappresentazione duale della CNF, mediante ipergrafi (ovvero, anzichè le clausole come insiemi di letterali, i letterali come insiemi di clausole a cui appartengono), ricerca un sottoinsieme minimo di letterali che possa dividere in due sottoinsiemi le clasusole, e dopo aver iterato questo procedimento un pò di volte, la risoluzione dei sottoproblemi, esponenzialmente più semplici. Purtroppo il partizionamento di un ipergrafo è anch'esso un problema molto complesso e dunque non è detto che questo procedimento veramente porti grandi vantaggi, specie nei problemi di piccole dimensioni.

- **Deferring Splitting** poichè negli split solo una parte delle clausole varia, durante l'applicazione del lookahead questo può rallentare di molto la computazione, onde con questa tecnica viene "ritagliata" la parte comune ai due rami dalla formula.

2.1.5 Valutazione della Efficienza

Come detto in precedenza, il SAT problem appartiene alla categoria dei problemi NP-Completi, la cui teoria si basa sulla complessità del caso peggiore, che non è significativa per capire l'efficienza dell'algoritmo in esame. Poichè ogni procedura ricerca in tutto lo spazio delle soluzioni, per delineare il comportamento di un SAT algorithm, si è pensato di utilizzare metodologie fondate sulla teoria di complessità del caso medio, ovvero sulla probabilità di distribuzione, legata alle "dimensioni" di opportune formule di input, generate in modo stocastico, in funzione del Solver che sarebbe stato utilizzato per il calcolo.

Benchè questo abbia portato alla ideazione di modelli di elevata difficoltà e aderenza ai problemi del mondo reale, come *k-SAT* ([Franco and Paull, 1983]), ovvero "Fixed Clause Lenght", ha reso manifesta la inadeguatezza di questo metodo, anche per via di fenomeni, come la "Phase Transiction" ([Hogg et al, 1996]), osservati empiricamente ma non teorizzati.

La via sperimentale sembra, dunque, il miglior metodo per analizzare le effettive capacità dei vari algoritmi. Inoltre permette di portare le questioni algoritmiche più vicino al problema originale per il quale è stata ideata la procedura e testare le assunzioni teoriche circa la convevienza delle scelte implementative effettuate.

E' nata un'area del "Propositional Reasoning" dedicata allo sviluppo di generatori automatici di problemi, alla loro conveniente codifica, nonchè ai metodi di comparazione delle prestazioni degli algoritmi di risoluzione.

Il Discrete MAthematics and theoretical Computer Science ha avuto un grosso ruolo in questo senso, fra le altre cose: organizzando sfide ([Sito DIMACS, 2002]), creando benchmark, proponendo formati di codifica ([DIMACS Format]).

2.2 DIMACS Parity Problems

Le istanze di questo problema sono la versione proposizionale del "Parity Learning Problem". I problemi sono auto riducibili, ovvero ogni istanza è riducibile in termini di un insieme di sotto-istanze random del problema dato.

Ogni problema è contenuto in file nominati *parx-y.cnf*, dove x è il numero di bit del vettore che entra nella funzione di parità, mentre y è un indice di istanza. Esistono 3 classi di problemi, per 8, 16, 32 bit rispettivamente e per ciascuna di esse 5 istanze, generalmente ordinate in ordine di difficoltà crescente (numero di soluzioni decrescente). A fianco esistono dei problemi "compressi": *parx-y-c.cnf*, che derivano da quelli precedentemente esposti previa semplificazione (compressione del numero di variabili proposizionali), che porta generalmente a dimezzare, per ogni istanza, il numero totale delle proposizioni e delle clausole.

Solo recentemente sono state risolte le istanze a 32 bit (cfr. [Li, 2000], [Warners and van Maaren, 1999]), con la escusione delle istanze numero 5, che sono rimaste ancora insolute. Un solver che riesca a risolvere i problemi della classe a 16 bit già si colloca al livelli dello "stato dell'arte".

2.2.1 Descrizione Generale

Il problema generale sta nell'identificare una funzione boolenana sconosciuta, sapendo :

- un certo numero di campioni (possibilmente sbagliati) di ingresso ed uscita della funzione
- la classe di appartenenza della funzione (in questo caso tutte le funzioni di parità)

La soluzione consiste nel soddisfare un limite inferiore al numero di errori compiuti sui campioni.

Ovvero, dati m vettori di lunghezza n, che rappresentano i campioni di ingresso

$$X_1, ..., X_m$$

m bit che rappresentano i campioni di uscita (potenzialmente sbagliati):

$$y_1, ..., y_m$$

ed un tolleranza all'errore

$$0 \leq E < 1$$

occorre trovare su quali bit di ingresso viene calcolata la parità, attraverso la valutazione di n bit $a_1, ..., a_n$ ($a_i = 1$ significa che il bit i partecipa al valore dela funzione).

Creazione dei Problemi

I problemi possono essere facilmente istanziati in questo modo:

- generazione di X_i, a_i
- calcolo della parità
- assegnare E
- rovinare $m \times E$ campioni

Sperimentalmente si è dimostrato che per $m = 2 \times n$ e $E \cong \frac{1}{4}$, con n abbastanza grande per avere meno modelli, i problemi diventano complicati.

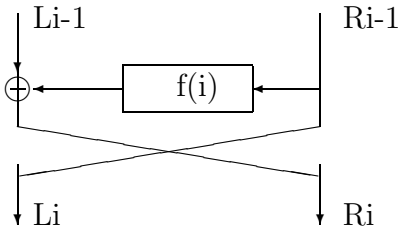

Figura 2.1: Blocco Elementare iterazione i-esima

2.3 Applicazione alla Criptoanalisi del DES

Come detto in precedenza, uno dei casi presi in esame per l'applicazione del solver alla Logica Affine, sono le istanze di criptoanalisi del DES.

In questo paragrafo, dunque, farò riferimento alla traduzione del DES in formule della Logica Affine non già per descriverla (per la qual cosa, si rimanda a [Massacci and Marraro, 2000]), quanto per rendere un'idea del numero e del tipo degli elementi presenti nelle istanze (e.g. rapporto del numero delle clausole XOR rispetto alle tradizionali OR, numero di proposizioni presenti, etc.).

2.3.1 Struttura del DES e sua Codifica in Logica Proposizionale

Nel seguito verrà descritta la struttura del Data Encryption Standard, analizzando numero e tipo di clausole generate da ciascun blocco funzionale.

Initial Permutation

La permutazione non è funzione del blocco di testo da cifrare, nè della chiave, onde, per evitare l'esplosione del numero di clausole, è stata tradotta in una permutazione corrispondente delle variabili proposizionali che rappresentavano il plaintext (ovvero le variabili Px, y, z), in modo del tutto trasparente.

Blocco Funzionale di Iterazione i-esima

Poichè il DES appartiene all'insieme dei Feistel Ciphers, vengono effettuate varie iterazioni (16 nel DES standard) di modifica, sul plaintext originale, ognuna delle quali provvede a che la metà più a sinistra del risultato del calcolo del blocco sia posta in XOR con una funzione (la $f(i)$) dipendente dalla chiave. Ne risultano, quindi 2 equazioni, ognuna da applicare su 32 bit:

$$\begin{cases} L_i = R_{(i-1)} \\ R_i = L_{(i-1)} \oplus f(i) \end{cases}$$

ovvero:

$$\begin{cases} R_i = R_{(i-2)} \oplus f(i) \\ f(i) = f(R_{(i-1)}, K_i) \end{cases}$$

Blocco Funzione $f(i)$

Questa funzione viene calcolata da un blocco interno e, come abbiamo anticipato, dipende dalla chiave. A meno delle permutazioni, che vengono risolte con una permutazione delle variabili, il suo schema è il seguente per il quale ottieniamo le seguenti equazioni:

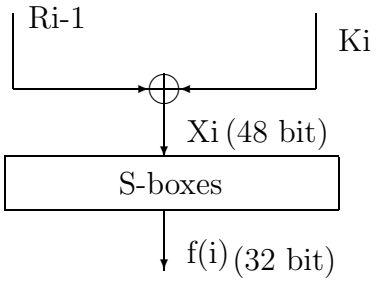

Figura 2.2: Blocco di Calcolo funzione f(i)

$$\begin{cases} S_i = \bigcup M_i \\ M_i = \bigcap X_i, C_i \\ X_i = R_{(i-1)} \oplus K_i \end{cases}$$

dove C_i sono i mintermini dei blocchi non lineari S. Con S_i sono state indicate le variabili che escono dalle S-boxes, ovvero il valore che assume $f(i)$.

Blocco Funzionale di Scelta delle K_i

Le K_i altro non sono che delle "scelte" tra le variabili della chiave. Poichè la scelta delle K_i consiste fondamentalmente dell'applicazione di una schema di permutazione e filtri, anch'essa è stata codificata senza bisogno di creare clausole, in modo trasparente. Nel seguito indicheremo le K_i a meno della permutazione corrente con K_0.

Inverse Initial Permutation

La permutazione non è funzione del blocco di testo da cifrare, nè della chiave, onde, per evitare l'esplosione del numero di clausole, è stata tradotta in una permutazione corrispondente delle variabili proposizionali che rappresentano il ciphertext (ovvero le variabili Cx, y, z).

Definizione delle Equivalenze e delle Variabili per ogni iterazione

Per ogni iterazione avremo dunque le seguenti clausole (espresse in forma di equivalenze):

1. dalla codifica del blocco funzionale di iterzione i-esima otteniamo le equazioni:

   - 32 identità (che non servono!) $L_i = R_{(i-1)}$
   - 32 XOR $R_i = R_{(i-2)} \oplus S_i$[3]

   e le nuove variabili proposizionali:

   - 32 R_i (calcolo iterativo sul plaintext)
   - 32 S_i (uscita dalle S-boxes, ovvero funzione $f(i)$)

[3]Le R_0 e le L_0 sono state indicate come P_0, che sta per PlainText, le R_{16} e le R_{15} sono state indicate con C_0, che sta per CipherText.

2. dalla codifica della funzione $f(i)$ otteniamo le equazioni:

   - 32 OR $S_i = \bigcup M_i$
   - 48 XOR $X_i = R_{(i-1)} \oplus K_0$

   e le nuove variabili proposizionali:

   - 48 X_i (XOR tra la chiave e le variabili di iterazione sul plaintext)
   - 552 M_i (mintermini delle S-boxes)

3. dalla codifica delle S-boxes

   - 552 AND $M_i = \bigcap X_i, C_i$

e quindi in definitiva, per ogni iterazione, abbiamo:

- 80 nuove equazioni XOR
- 32 nuove equazioni OR (da più di 20 elementi in media ciascuna)
- 552 nuove equazioni AND (da 4 elementi in media ciascuna)
- 696 nuove variabili

2.3.2 Numero Totale di Equazioni e Variabili in Funzione delle Iterazioni e dei Blocchi Analizzati

Indicando con b il numero dei blocchi presi inconsiderazione e con r il numero delle iterazioni richieste vengono generate:

- $(80 * r) * b$ equazioni XOR
- $(552 * r) * b$ equazioni AND
- $(32 * r) * b$ equazioni OR
- $(696 * r) * b$ variabili

Semplificazione in Caso di Attacco con PlainText Conosciuto

Nel caso in cui venga simulato un attacco con PlainText conosciuto[4] (ovvero in cui si conoscano entrambi ciphertext e plaintext) la formula si semplifica in modo sensibile. Infatti:

- Spariscono le variabili P_0 e C_0
- Sostituendo i valori di P_0 e C_0 si generano identità e costanti nelle clausole XOR, di modo che il loro numero diminuisca a :
  - $(r-8)$XOR ($\times 32$) del tipo: $R_i = R_{(i-2)} \oplus S_i$ con $5 \leq i \leq (r-4)$

[4]Caso assai più frequente, nella pratica, di quanto si possa pensare, infatti i messaggi di inizio transazione contengono solitamente stringhe facilmente deducibili, come ragione sociale, etc.

  - 2 XOR ($\times 32$) del tipo: $R_i = \pm S_{(i-2)} \oplus S_i$ con $i = 3, 4$
  - 2 XOR ($\times 32$) del tipo: $\pm S_i = R_{(i-4)} \oplus S_{(i-2)}$ con $i = (r-1), r$
  - $(r-6)$ XOR ($\times 48$) del tipo: $X_i = R_{(i-1)} \oplus K_0$ con $4 \leq i \leq (r-3)$
  - 2 XOR ($\times 48$) del tipo: $X_i = \pm S_{(i-1)} \oplus K_0$ con $i = 2, 3$
  - 2 XOR ($\times 48$) del tipo: $X_i = \pm S_{(i+1)} \oplus K_0$ con $i = (r-2), (r-1)$

- Spariscono le variabili X_1 e X_r ($\times 48$), con opportuni cambiamenti anche nelle formule AND e OR
- Nel caso di numero di round inferiore a 3, non tutto il plaintext viene posto in XOR con la chiave, e questo si ripercuote tramite genrazione di sole equazioni AND

Per quanto esposto, possiamo quindi concludere che, in caso di attacco con testo in chiaro conosciuto

1. per $r = 1, 2$ otteniamo
   (a) $552 \times r$ equazioni di tipo AND
   (b) 56 variabili K_0, $552 \times r$ variabili M_i,
2. per $3 \leq r \leq 4$ otteniamo
   (a) $48 \times (r-2)$ equazioni XOR
   (b) $552 \times r$ equazioni AND
   (c) $32 \times r$ equazioni OR
   (d) 56 variabili K_0, $552 \times r$ variabili M_i, $48 \times (r-2)$ variabili X_i, $32 \times (r-2)$ variabili S_i, $32 \times (r-2)$ variabili R_i
3. per $5 \leq r \leq 16$ otteniamo
   (a) $(48 \times (r-2) + 32 \times (r-4))$ equazioni XOR
   (b) $552 \times r$ equazioni AND
   (c) $32 \times r$ equazioni OR
   (d) 56 variabili K_0, $552 \times r$ variabili M_i, $48 \times (r-2)$ variabili X_i, $32 \times (r-2)$ variabili S_i, $32 \times (r-2)$ variabili R_i

Osservo esplicitamente che le equazioni, per essere date in input ad un solver, necessitano di una "traduzione" in CNF, o analoga, come vedremo nel prossimo paragrafo, che farà lievitare ulteriormente il numero delle clausole.

Risulta evidente, quindi, che già utilizzando un numero di round uguale a 3 si ottengano problemi di una certa difficoltà, paragonabili ai DIMACS *parity32*. Con il numero di round uguale a 4 si ottengono istanze di gran lunga più complicate del *parity32*.

2.3.3 Trasformazione delle Formulae

I solver, generalmente, lavorano sulla forma normale a clausole, ovvero ogni vincolo (*constraint*) viene espresso, sia in input, sia durante la fase di calcolo, in clausole. Anche per solver, come il mio, che lavorino anche su logica affine, occorre trasformare le equazioni in clausole proprie (anche se affini) della logica proposizionale. Vediamo dunque come effettuare questa traduzione, soffermendoci su ogni tipo di equazione.

Equazioni XOR

Per ogni equazione si ottiene una clausola XOR, con esattamente gli stessi letterali, infatti:
$L = A + B + ... + Z$
diviene:
$\neg L + A + B + ... + Z$

Equazioni OR

Da 1 equazione OR con n letterali a secondo membro, otteniamo esattamente n+1 equazioni, infatti sia:

$$L = A_1 \vee A_2 \vee ... \vee A_n$$

ovvero:

$$\neg L \oplus (A_1 \vee A_2 \vee ... \vee A_n)$$

trasformando con lo xor, sapendo che, in generale:

$$A \oplus B \Leftrightarrow (A \vee B) \wedge (\neg A \vee \neg B)$$

otteniamo:

$$\neg L \vee (A_1 \vee A_2 \vee ... \vee A_n) \wedge (L \vee \neg(A_1 \vee A_2 \vee ... \vee A_n))$$

applicando le leggi di De Morgan:

$$\neg L \vee (A_1 \vee A_2 \vee ... \vee A_n) \wedge (L \vee \neg A_1) \wedge (L \vee \neg A_2) \wedge ... \wedge (L \vee \neg A_n)$$

poichè ogni equazione OR contiene ca. 22 letterali, dopo la virgola, possiamo affermare che ad ogni equazione OR corrispondono:

1. 1 clausole OR da 23 letterali ca.
2. ca. 22 clausole OR da esattamente 2 letterali

Equazioni AND

Per esse vale un discorso analogo a quanto fatto nel paragrafo precedente, con la differenza che i segni sono cambiati, per via delle leggi di De Morgan, e che la lunghezza delle euqazioni è minore rispetto alle OR (ca. 4 letterali dopo la virgola), onde otteniamo, per ogni equazione AND:

1. 1 clausola OR da 5 letterali ca.
2. ca. 4 clausole OR da esattamente 2 letterali

Capitolo 3

L'algoritmo

L'algoritmo proposto deriva, nella sua struttura centrale, interamente dal sistema di regole di inferenza proposto in [Baumgartner and Massacci, 2001], con opportune modifiche ed aggiunte però, volte essenzialmente ad ottimizzare alcuni momenti di computazione cruciali ed inserire utili tecniche implementative.

La particolarità del lavoro svolto riguarda soprattutto le strutture dati utilizzate e le influenze reciproche, in un certo senso potremmo dire le "sincronizzazioni" tra gli insiemi coinvolti. Infatti, l'idea fondamentale era quella di mettere a comune i risultati delle riduzioni/semplificazioni su ogni entità, cercando di accelerare la computazione, creando al contempo inevitabili necessità di comunicazione tra gli insiemi.

Ne è derivato un solver sicuramente più complesso della maggior parte degli altri. Sono presenti, infatti, ben 4 insiemi distinti (contro l'unico insieme che si usa normalmente negli algoritmi che derivano dal DPLL) di tipo "logico" a cui se ne aggiungono altri, in modo dinamico, che effettuano fondamentalmente funzioni di "buffer", onde ridurre l'impatto di alcune funzioni di frequente utilizzo.

Per le Euristiche si è proceduto ideando nuove funzioni di priorità, eventualmente adattando strategie, presenti in letteratura, allo "spezzettamento" delle informazioni di stato nei vari insiemi.

Vediamo più in dettaglio la struttura "logica" dell'algoritmo, soffermendoci sulla sua realizzazione in un prossimo capitolo.

3.1 Base

Il nucleo deriva dal DPLL, con l'aggiunta del trattamento esplicito delle clausole XOR, tramite un adattamento della regola di eliminazione di Gauss, ovvero eseguendo i calcoli semplificativi sulle formule XOR, sfruttandone la doppia natura di:

1. espressioni della logica booleana
2. equazioni del campo $(Z_2, +, *)$.

La caratteristica fondamentale, da un punto del tempo di computazione, risiede nella richiesta di minori risorse, infatti la risoluzione di Gauss richiede un tempo polinomiale e non esponenziale come il processamento della regola del *cut*, nel DPLL classico.

Un'altra prerogativa è quella di fare un intenso utilizzo della funzione *lookahead()* ([Harrison, 1996], [Groote and Warn, 2000]) non solo nel calcolo euristico, ma anche nella normale computazione, alla ricerca di possibili contraddizioni e vincoli addizionali.

Tutte le regole di inferenza che agiscono senza effettuare assegnazioni arbitrarie (ovvero tutte le regole ad eccezione della *split()*) sono state raccolte nella funzione *calculate()*, dove vengono eseguite secondo il corretto ordine di priorità (si veda il paragrafo, in questo stesso capitolo ad essa dedicato).

E' stata definita, per comodità, in modo da riunificare la parte di calcolo che non fa uso di assegnazioni arbitrarie, la funzione *compute()*, che non fa altro che invocare la funzione *calculate()* e la *lookahead()*. Nella procedura *restart()*, che riveste il ruolo di struttura esterna dell'algoritmo vengono richiamate implicitamente la *calculate()* e la *lookahead()*, quando viene invocata la *compute()*, per cercare di semplificare al massimo la formula e la *split()*, per ottenere un progresso nella computazione grazie ad una assegnazione "arbitraria".

La procedura termina qualora si giunga ad una descrizione funzionale del modello o, se sia stato scelto di ricercare tutte le soluzioni, alla scoperta dell'ultimo modello.

All'inizio, prima che venga avviata la procedura di *restart()*, viene processata la formula originale, come letta da input, alla ricerca di tutte le semplificazioni possibili da effettuare, tramite una funzione molto simile alla *compute()*, che, quindi, contempla sia riduzione tramite regole di inferenza (*calculate()*), sia ricerca di contraddizioni possibili tramite *lookahead()*. Viene effettuato anche un controllo, qualora la formula originale non sia soddisfacibile.

3.1.1 La Struttura Esterna: *restart*()

Come detto in precedenza, questa funzione si occupa di creare la base di tutto l'algoritmo. Essa prende in ingresso due formule, una detta "di riferimento", che altro non è che la formula originale, precedentemente semplificata più possibile, usata in sola lettura, l'altra detta "di calcolo", utilizzata anche in modifica, onde effettuare i calcoli. La funzione ha preso il nome di restart, in quanto, qualora si dovesse raggiungere una contraddizione durante la semplificazione, a seguito di qualche assegnazione, sulla formula "di calcolo", viene effettuato un backtracking sulle assegnazioni delle variabili e riassegnata la formula "di calcolo" con la formula "di riferimento" (che per quanto detto in precedenza non contiene contraddizioni esplicite). Ovvero l'algoritmo assume la seguente struttura:

```
restart (formula reference, formula tide, variable-set VB) {
  if (tide has a contraddiction)
    eliminate all the not backtrack variables from VB;
    invert the sign of the the last var in VB;
    tide = reference;
    add the assignment on last var of VB in tide;
    compute (tide);
  }
  else {
    var a = Split();
    if (there isn't var to Split())
      return solution;
    else {
      insert a in VB;
      add a  in tide;
```

```
      compute (tide);
    }
  }
}
```

3.1.2 Compute

Come accennato ad inizio paragrafo, si è ritenuto comodo dichiarare una funzione *compute()* che racchiudesse tutte le funzionalità di calcolo non contenenti assegnazioni arbitrarie di variabili, la cui definizione ad alto livello è:

```
compute (formula F) {
  calculate(F);
  lookahead (F);
}
```

Sono stati aggiunti poi dei controlli al suo interno, onde gestire opportune opzioni da riga di comando, per abilitare o meno l'utilizzo della lookahead.

Esiste una variante della *compute()* utilizzata unicamente durante la procedura di semplificazione, contenente anche istruzionidi salvataggio delle assegnazioni trovate.

Vediamo in dettaglio nei prossimi sottoparagrafi la definizione delle due funzioni chiamate dalla *compute()*.

3.1.3 Lookahead

Questa procedura serve per calcolare l'effetto di uno split, senza eseguirlo sulla formula corrente, senza dover quindi, in caso negativo, dover poi eseguire un backtrack. Le assegnazioni vengono, quindi, calcolate su una formula a parte.

Originariamente questa metodologia era stata introdotta per scovare le variabili, la cui assegnazione, in un senso o nell'altro, causava una immediata contraddizione ("*one level lookahead*"). D'altronde, soprattutto all'inizio della computazione, questa procedura può essere utilizzata anche per ricercare le clausole comuni ai due rami di assegnazione della variabile corrente, onde ricercare le clausole comuni ed aggiungerle come vincoli addizionali alle clausole originariamente contenute, effettuando anche la sussunzione tra le clausole (che vedremo in dettaglio in un paragrafo successivo di questo capitolo dedicato alle ottimizzazioni) per non far aumentare eccessivamente il numero di clausole ("*two level lookahead*").

Tutte le euristiche più accurate per DPLL includono questa funzione, sebbene rappresenti una sforzo computazionale ben maggiore delle misure lessicografiche.

In questo solver è si è deciso di utilizzare la lookahead non soltanto come strumento di calcolo semplificativo, adottando una versione a due livelli. Questa scelta ha significato una ingente perdita di velocità nei problemi di piccola entità, ma ha ripagato nella computazione di istanze più complicate, non risolvibili da procedure di ricerca che non ne fanno utilizzo assiduo.

Vediamone la struttura di funzionamento ad alto livello

```
lookahead (formula F) {
  formula Fneg, Fpos;
  forall (var v in F) {
    add the assignment (v=false) in Fneg;
    add the assignment (v=true) in Fpos;
    if (Fpos has a contraddiction){
      F = Fneg;
      go on;
    }
    else
      if (Fneg has a contraddiction) {
F = Fpos;
go on;
      }
      else F = F set_union (Fneg set_intersect Fpos);
      go on;
  }
}
```

3.1.4 Vantaggi dell'Algoritmo

1. Approccio coerente per il trattamento delle clausole OR e XOR, con ottimizzazione delle procedure, qualora l'input sia ristretto ad un tipo di clausole.

2. Pesante interazione delle parti coinvolte, con lo scopo di massimizzare la semplificazione, passando le clausole da un insieme all'altro

3. Possibilità di stoppare il procedimento, visualizzando una descrizione funzionale del modello, piuttosto che una completamente specificata.

4. Possibilità di eseguire lo split non solo sulla base dell'assegnazione di un valore costante ad una variabile, ma anche dell'assegnazione tra 2 costanti.

3.1.5 Svantaggi dell'Algoritmo

1. Per via della presenza dele clausole XOR, è stato necessario eliminare la "pure literal rule", che in alcune semplici istanze rappresenta un incremento di velocità notevole

2. Per via della struttura piuttosto complessa, l'algoritmo è piuttosto lento nel risolvere semplici problemi.

3.2 Calculate

Calculate Questa sezione si occupa di definire il modo e le motivazioni con cui è stato ideato il modulo che esegue, per ogni nodo dell'albero computazionale, i calcoli semplificativi che non richiedono assunzione di assegnazione di variabili .

3.2.1 Insiemi Coinvolti

La Calculate opera su una Formula F, onde gli insiemi sono quelli definiti nell'oggetto `formula`. In più viene acceduto un insieme di assegnazioni temporanee, in modo da guadagnare in efficienza nell'applicazione della Simp() e delle altre regole da essa internamente richiamate.

Assegnazioni Correnti (Ass)

$Atomo = Letterale$

Provengono dalla applicazione delle regole:

- Unit()
- Eqv()
- Split()

Viene popolata dalla funzione Simp(), attraverso l'insieme l'insieme delle assegnazioni temporanee XQ, introdotto per ottimizzare le regole: Red(), Unit(), Gauss() e la scoperta di bottom.

Assegnazioni Temporanee (XQue)

Come detto in precedenza, contiene le assegnazioni generate nel ciclo corrente e dunque le uniche utili nel momento. Per mantenere la idempotenza e la funzionalità, questo insieme viene semplificato ad ogni passo di computazione (tramite la SimpQue()).

Insieme delle OR-clause (Or)

Proviene dai vincoli sulle variabili proposizionali espressi in forma di congiunzione[1].

Non puo' aumentare. Puo' solo essere ridotto, da:

- RedOr()
- SimpOr()
- UnitOr()
- EqvOr(), Or3Xor()
- Split()

[1] Nel caso particolare del DES, deriva dalle equazioni OR:
$S_i = \bigcup_j C_{i,j}$
e dalle equazioni AND (descrizione delle S-box):
$C_i = \bigcap_j M_{i,j}, K_{i,j}$

Insieme delle XOR-clause (Xor)

Proviene dai vincli sulle variabili proposizionali espressi in forma di equivalenza o disgiunzione[2]: Puo' essere ridotto:

- RedXor()
- SimpXor()
- UnitXor()
- EqvXor()

Puo' essere modificato/ridotto da:

- Gauss()

Selezione delle XOR-clause "definizioni"¨ (S)

Proviene dalla applicazione della Select(). Puo' essere ridotto dalla SimpXor().

3.2.2 Regole Utilizzate

1. Regola di Riduzione
   (a) $RedXor : Xor \rightarrow Xor$
   (b) $RedOr : Or \rightarrow Or$

l'unica operazione "interna".

1. Regole di Selezione - Spostamento[3]
   (a) $UnitXor : Xor \rightarrow Ass$
   (b) $UnitOr : Or \rightarrow Ass$
   (c) $EqvXor : (Xor, S) \rightarrow Ass$
   (d) $EqvOr : (Or, Or) \rightarrow Ass$
   (e) $Select : Xor \rightarrow S$
   (f) $Or3Xor : (Or, Or, Or) \rightarrow Xor$
2. Regole di Azione - Calcolo
   (a) $Gauss : (Xor, S) \rightarrow Xor$
   (b) $Simp : (Formula, XQue) \rightarrow (Formula)$
   (c) $Split : Formula \rightarrow (Formula, Formula)$

Le Xor Clause selezionate (gli elementi di S) vengono ridottie semplificati assieme alle Xor Clause "normali".

[2]Nel caso particolare del DES, deriva dalla equazioni delle iterazioni:
$R_{i=}R_{(i-2)} \oplus S_i$ (32 per ogni giro)
e dalla equazioni della funzione $f(i)$:
$X_i = R_{(i-1)} \oplus K_{0,i}$(48 per ogni giro)
$\Rightarrow$ 1056 per 16 cicli (considerando le semplificazioni)

[3]Si potrebbe pensare anche di aggiungere la regola:
nul : nessuna operazione applicabile

3.2.3 Flusso di Applicazione delle regole

l'esecuzione di una delle regole di selezione può implicare lo spostamento di una clausola od una variabile in un determinato insieme e dunque la necessita' di eseguire subito la corrispondente regola d'Azione. In altre parole esiste una sorta di "dipendenza", secondo la tabella:

Selezione	Azione	Riduzione
Unit()	Simp()	Red()
Select()	Gauss()	Red()
Eqv()	Simp()	Red()
nul	Split()	Red()

Le altre regole di spostamento o riduzione implicano altre regole di selezione, secodo la tabella:

Sel/Red	Selezione	Azione
Red()	Unit()	Simp()
Or3Xor()	Select()	Gauss()

Alcune regole d'Azione fanno pero' dipendere alcune regole di Selezione, creando l'opportunità di veri e propri cicli". Vediamo più in dettaglio regola per regola.

3.2.4 Analisi delle Regole

Qui le regole si intendono applicate agli insiemi di clausole, avendo "cablato" all'interno eventuali cicli. Per ogni regola sono elencate le voci:

- per (a quali insiemi si applica)
- definizione (le operazioni eseguite)
- precedenze (prima e dopo di quali altre regole conviene che sia applicata)
- controllo (se nell'implementare la precedenza, la regola debba essere eseguita solo se si verfica un determinato evento)
- ciclo (eventuali cicli con altre regole)
- euristica (solo per Select() e Split())
- implementazione (come si pensa di implementare la regola, in linea di massima)

Red()

per: XOR, OR

definizione: riduce le clausole togliendo:

- costanti
- letterali

precedenze: Va usata il piu' spesso possibile!!

utile solo dopo l'applicazione di tutte le regole che non effettuano una selezione; prima delle regole che eseguono uno spostamento verso Ass:

$$\left.\begin{array}{l} Gauss() \\ Simp() \\ Split() \end{array}\right\} \Rightarrow Red() \Rightarrow \left\{\begin{array}{l} \\ Unit() \\ Eqv() \end{array}\right.$$

controllo no

ciclo: no.

implementazione: scandisce, una per una, tutte le clausole, in modo ricorsivo, negli insiemi.

Simp()

per: XOR, OR

definizione: applica le assegnazioni in Ass alle clausole Xor e Or. Restituisce i gruppi di clausole che risultano semplificate.

precedenze: Va usato il piu' spesso possibile, dopo una regola che determini uno spostamento di clausole verso Ass; prima delle regole che sfruttano le modificate condizion sintattiche:

$$\left.\begin{array}{l} Eqv() \\ Unit() \end{array}\right\} \Rightarrow Simp() \Rightarrow \left\{\begin{array}{l} Red() \\ Or2Xor() \\ Or3Xor() \end{array}\right.$$

controllo: deve verificarsi l'aggiunta di un nuovo elemento in Ass

ciclo: no

implementazione: si esegue la Simp() che scandisce Ass dall'inizio alla fine

Select()

per: XOR

definizione: seleziona una xor-clause ed un letterale ad essa appartenente:

- inserisce un nuovo record nella lista dei $sel(X) = S$, del tipo: $A ::= B \oplus C \oplus ...$

In pratica e' come selezionare una clausola\left.\begin{array}{l}Red()\\Or3Xor()\\Simp()\\Gauss\end{arra
precedenze: ogni regola di riduzione ha la precedenza, in modo da selezionare definizioni più semplici. In particolare devono avere la precedenza: $RedXor()$, $Gauss()$.

ciclo: si

controlli: controllare che le XOR-clause non siano state processate tutte ($||X|| \neq 0$). Inoltre, supponendo che si voglia selezionare la xor-clause x, definendo $atom(x)$ gli atomi conteuti in x, con $atomsel(X)$ gli atomi di X per cui si abbia una definizione in S, deve risultare: $atom(x) \cap atomsel(X) = \emptyset$ e $atom(x) \cap assignedvar(A) = \emptyset$

euristica: statica per la clausola (seleziona la prima disponibile che rispecchi i requisiti) ed analoga al MOMS per la scelta della variabile da definire

implementazione: per massimizzare le selezioni, soddisfacendo i controlli, ogni nuova selezione deve essere effettuata dopo aver eseguito tutte le riduzioni di Gauss, le riduzioni etc. Scandisce tutte le clausole; appena trova una selezione possibile effettua lo spostamento ed interrompe il ciclo, restituendo il letterale appena definito.

Gauss()

per: XOR

definizione:

precedenze: Select()⇒Gauss()⇒Red()

ciclo: no

controlli: che la Select() sia stata applicata con successo

implementazione: scandisce tutte le clausole per sostituire alla variabile segnalata dalla Select() la Xor-definizione.

Unit()

per: XOR, OR

definizione: cerca le unit-clauses, le toglie dall'insieme delle caluses e le include in *Ass*. Responsabile anche della ricerca delle *empty-clause* (su C e X).

precedenze: dopo averlo eseguito la prima volta, bisogna eseguirlo ogni qualvolta si creano altre potenziali unit-clause.

$$\left.\begin{array}{l} Red() \\ Gauss() \\ Simp() \end{array}\right\} \Rightarrow Unit() \Rightarrow Simp()$$

conviene sempre alla fine cercare di semplificare con la Simp().

ciclo: sì dentro la $Simp()$

controllo: no

implementazione: scandisce tutte le clausole alla ricerca di una unit-clause

Or2Xor()

per: OR

definizione: cerca coppie di Clausole OR da tradurre in una clausola XOR binaria

precedenze: va eseguita ogni qualvolta abbiamo una modifica delle sole clausole OR e prima di operare sulle formule Xor:

$$\left.\begin{array}{l} Red() \\ Simp() \end{array}\right\} \Rightarrow Or2Xor() \Rightarrow \left\{\begin{array}{l} Eqv() \\ Select() \end{array}\right.$$

controllo: che ci sia stata una effettiva modifica di qualche Or-clause a 2 letterali.

ciclo: no

implementazione: scandendo 2 volte la lista delle clausole OR

Or3Xor()

per: OR

definizione: cerca quartetti di clausole OR di 3 letterali da tradurre in una clausola XOR ternaria

precedenze:va eseguita ogni qualvolta abbiamo una modifica delle sole clausole OR e prima di operare sulle formule Xor:

$$\left.\begin{array}{l} Red() \\ Simp() \end{array}\right\} \Rightarrow Or3Xor() \Rightarrow Select() \Rightarrow Gauss()$$

controllo: che sia stata una effettiva modifica a qualche Xor-clause a 3 letterali

ciclo: no

implementazione: scandendo 4 volte la lista delle clausole OR

Eqv()

per: Xor

definizione: sposta le Xor-definizioni binarie da Xor a Ass.

precedenze: dopo le modifiche alle clausole Xor

$$\left.\begin{array}{l} Red() \\ Simp() \\ Or2Xor() \\ Gauss() \end{array}\right\} \Rightarrow Eqv() \Rightarrow Simp()$$

controllo: che si siano generate nuove xor-clause binarie

ciclo: sì

implementazione: scandisce tutta la lista delle Xor-clause alla ricerca di binary.

Split()

per: Or

definizione: sceglie una variabile non assegnata per creare 2 sotto alberi computazionali, assegnando alternativamente l'uno e l'altro valore.

precedenze: quando non può essere eseguita nessun altra regola

controllo: vedi precedenze

ciclo: no

implementazione: scandisce tutte le clausole e tutte le variabili per trovare la variabile "giusta"

euristica: probabilmente basata su *lookahead*.

3.2.5 Flusso delle clausole e delle proposizioni

In generale quello che ci si aspetta dall'algoritmo è di spostare rapidamente quante più variabili possibile nell'insieme delle assegnazioni, "svuotando" gli altri insiemi di clausole. Per questo ci prefiggiamo 4 obbiettivi principali:

1. ridurre il numero delle clausole OR $\Rightarrow$ $EqvOr()$, $Or3Xor()$. In modo da:
   - ridurre il rischio di $Split()$
   - aumentare potenza del $Gauss()$
2. aumentare gli atomi valutati $\Rightarrow$ $Unit()$. In modo da:
   - ottenere il valore degli atomi
   - togliere clausole dalla computazione
   - leggere la solzione da Ass
3. ridurre il numero delle clausole XOR $\Rightarrow$ $EqvXor()$
4. riconoscere subito le contraddizioni, dopo il primo $Split()$ $\Rightarrow$ $Red()$, $Unit()$
   - tramite generazione di clausole vuote, rivelata dalla Unit()

3.2.6 Metodi di Applicazione delle Regole

Essendoci una palese interdipendenza tra le regole, è possibile intravedere 3 metodi fondamentali di interazione:

guidata: la regola viene richiamata su una clausola specifica. E' il caso delle regole che agiscono su un solo parametro di input (Red(), Unit()). La loro applicazione risulta la più efficiente, in quanto agiscono su un singolo elemento, che ha appena subito una modifica e dunque potenzialmente potrebbe dare buoni risultati dalla applicazione della regola. La loro complessità generalmente è $O(1)$.

forzata: la regola viene chiamata in seguito ad un non meglio definito cambiamento sulla intera formula. Non è però dato sapere bene, su quali clausole la modifica vada ad impattare ed è dunque necessario eseguire una scansione di tutto l'insieme. E' il caso delle regole che agiscono su almeno 2 parametri di input (Simp(), Gauss()), effettuando una ricerca sull'intero insieme. La loro complessità generalmente è $O(n)$.

obbligata: la regola viene chiamata in quanto l'algoritmo la prevede nel suo percorso, ma non è detto che dia dei risultati utili. E' il caso delle regole che ricercano all'interno degli insiemi elementi che contengano particolari proprietà (Eqv(), Or3Xor()). La loro complessità generalmente è $O(n \log n)$

3.2.7 Definizione dell'algoritmo

In base a quanto visto finora, con riferimento particolare agli obiettivi, si riscontrano le seguenti conclusioni:

Ordine di priorità

Le regole da applicare assumono questo ordine di priorità:

1. RedOr(), RedXor()
2. SimpOr(), SimpXor()
3. UnitOr(), UnitXor()
4. EqvOr(), EqvXor()
5. Or3Xor()
6. (Select(), Gauss()) (da applicarsi però in modo esaustivo)
7. Split()

Proprietà delle Assegnazioni/Definizioni

- tutte le riduzioni e semplificazioni, derivano dal rinvenimento di nuove assegnazioni (tramite Eqv() o Unit()) o definizioni nelle XOR, tramite Select()
- solo le nuove assegnazioni/definizioni sono utili al proseguo del calcolo
- è possibile tenere dette assegnazioni in un insieme separato, prima di inserirle fra quelle già utilizzate, in modo da poter anche prevenire il venir meno delle proprietà di idempotenza e funzionalità

$\Rightarrow$ buffer delle nuove assegnazioni $XorQue$

Cicli di Applicazione delle Regole

Si generano i seguenti cicli:

- $Eqv \rightarrow Simp \rightarrow Red \rightarrow Unit \rightarrow Eqv \rightarrow Simp...$
- $Split \rightarrow Simp \rightarrow Red \rightarrow Unit \rightarrow Eqv \rightarrow Simp...$
- $Select \rightarrow Gauss \rightarrow Red \rightarrow Unit \rightarrow Eqv \rightarrow Simp...$

Tutti questi cicli sono compatibili con la definizione dell'insieme temporaneo di nuove assegnazioni di cui al punto precedente

Utilizzo Mirato

- è conveniente massimizzare il numero delle regole che vengono applicate in modo "guidato".

⇒ $Red()$ e $Unit()$, vengono richiamate solamente all'interno delle $SimpOr()$ e $SimpXor()$.

- è conveniente utilizzare solo le nuove assegnazioni generate

⇒ i cicli sono basati sul processamento del buffer $XorQue$ e sono definiti all'interno delle funzioni:

- $Simp()$
- $Split()$
- $Gauss()$

Implementazione Finale della Calculate()

Si è quindi pensato di sistemare le regole in questo modo:

- inserire le 2 regole che effettuano azioni solo in base alle proprietà della singola clausola ($RedOr()$, $RedXor()$, $UnitOr()$, $UnitXor()$) all'interno delle funzioni di semplificazione ($Simp()$, $SimpOr()$, $SimpXor()$, $SimpQue()$).
- costruire un ciclo per le $Eqv()$:

  $do\{$

  $EqvOr();$

  $EqvXor();$

  $Simp();$

  $\}while(change);$
- inserire la $Or3Xor()$ isolata
- costruire un ciclo per la $Gauss()$:

  $do\{$

  $a = Select();$

  $Gauss(a);$

  $Simp();$

  $\}while(change);$
- racchiudendo il tutto dentro un ciclo:

  $do\{$

  $\}while(change);$

  ed inserendo gli opportuni controlli su eventuali bottom verificatisi.

3.3 Euristiche

L'euristica (dal greco $\epsilon\upsilon\rho\iota\zeta\omega$ che significa trovo, scopro), nelle scienze ipotetico-deduttive, è un procedimento non rigoroso (approsimativo, intuitivo, analogico) che permette di conseguire un risultato da verificare poi in modo rigoroso. A seconda di quale sia il tipo di risultato conseguito, l'accezione di questa parola cambia. Nell'informatica, essa ha assunto i significati di:

- metodologia per l'invenzione di tecniche di risoluzione dei problemi
- processo di risoluzione senza garanzia di esattezza, nè ottimalità
- regola per generare buone soluzioni senza servirsi della ricerca sistematica
- tecnica per migliorare il comportamento di un algoritmo nel caso medio

Nei procedimenti basati sulla ricerca sistematica, il significato che assume è proprio l'ultimo, nel senso di funzione che provveda un'accurata stima dei costi di risoluzione. Nel DPLL, e derivati, viene utilizzata, quindi, principalmente per scegliere le variabili sulle quali, di volta in volta, effettuare lo *splitting*.

L'euristica ottima dovrebbe garantire la scelta delle proposizioni, tale da causare l'esplorazione del numero minimo di nodi dell'albero computazionale. Sfortunatamente questo è un problema ancora più complicato del SAT Problem.

Hooker e Vinary (cfr. [Hooker and Vinary, 1994]) concepirono la *Simplification Hypotesis for SAT*: una "Euristica è tanto migliore, quanti più semplici, da analizzare, sono i sottoproblemi da essa generati, se le altre cose rimangono uguali".

In pratica, una buona euristica deve possedere le due seguenti proprietà;

- richiedere poco tempo per il calcolo
- fornire un miglior punteggio alle scelte che generino sottoproblemi di minore entità

3.3.1 Criteri Euristci per il SAT Problem

Nel caso del DPLL, ci sono fondamentalmente tre proprietà da tenere presenti, nella scelta di una buona euristica.Considerando una formula F, per ogni variabile v su cui eseguire lo split, considerando i due rami generati $BCP(F[l \leftarrow \top])$ e $BCP(F[v \leftarrow \bot])$ la scelta di v deve consentire di:

1. visitare meno alberi e dunque agevolare la scelta per lo split successivo, ovvero $BCP(F[l \leftarrow \top])$ e $BCP(F[v \leftarrow \bot])$ devono contenere il maggior numero possibile di clausole binarie
2. avvicinarsi maggiormente alla soluzione e dunque $BCP(F[l \leftarrow \top])$ e $BCP(F[v \leftarrow \bot])$ devono contenere il maggior numero di proposizioni assegnate
3. ottenere un albero bilanciato, generando sottoproblemi di analoga dimensione, ovvero il numero di proposizioni assegnate nei due casi, di cui al punto 2., devono essere circa uguali.

3.3.2 Struttura Generica di una Euristica

Nel caso del SAT Problem, la generica struttura di una euristica è composta da:

- **Branching Strategy**: descrive tutte le regole, legate a opportuni parametri, tramite le quali scegliere la proposizione più adeguata
- **Priority Function**: traduce in una funzione che calcola, tramite una formula matematica, la priorità da associare alla proposizione in esame in dipendenza dalle regole descritte
- **Branching Order**: determina se la variabile debba essere prima negata o asserita.

3.3.3 Criteri Euristici per il SAT Problem

3.3.4 Punti di Intervento

Le euristiche, in questo algoritmo, comprendente anche il trattamento della logica affine, tramite un vero e proprio "riavvio" della computazione, sono da porsi nei seguenti punti:

Split() come in ogni algoritmo derivato dal DPLL, dove lo stato è rappresentato dalla formula corrente (comprendente sia le clausole che le assegnazioni) e dalle variabile assegnate da split() precedenti.

Reorder() come si è visto, questo algoritmo implementa una variante del DPLL, tramite Restart(). Ad ogni riavvio è utile riordinare le clausole. In questo caso, lo stato è rappresentato dalla formula originale (comprendente, al solito, clausole ed assegnazioni) e dalle variabile assegnate da split() precedenti, da riordinare, eventualmente cambinadone il segno

Select() per il trattamento delle clausole XOR, la scelta della prossima variabile da definire. Lo stato è rappresentato in questo caso dalla formula corrente (considerando le sole clausole XOR e le assegnazioni)

3.3.5 Tipologie

La definizione della branching strategy parte quindi dall'analisi delle caratteristiche del problema e del modo di analizzarlo. In particolare abbiamo due tipi di euristiche:

Problem Indipendent Heuristics

Sono le euristiche tradizionali negli algoritmi aricerca sistematica, non rivolte a problemi specifici o a classi specifiche di problemi. Viceversa mirano a migliorare il comportamento dell'algoritmo di risoluzione, in base ai meccanismi interni di funzionemanto dello stesso. Rivestono, ovviamente, maggiore importanza, in quanto mirano a creare dei "Solver" quanto più potenti e versatili possibile.

Problem Driven Heuristics

Sono euristiche specifiche, che scaturiscono dall'analisi di quelle proprietà particolari del problema in esame, che possono essere sfruttate per generare predizioni o migliorare la velocità di

ricerca. Può essere opportuno usufruirne, solo in aggiunta alle euristiche tradizionali, nei casi particolarmente intricati, come questo in esame.

Nell'algoritmo proposto, ho cercato di utilizzare alcune proprietà del DES, in modo da derivare alcune regole della Branching Strategy, che potessero essere usate all'interno delle euristiche più generali, quasi a modo di filtro ulteriore (cercare tutte le proposizioni che rispecchino le regole dell'euristica generale ed in più le regole dell'euristica specifica).

3.3.6 Euristiche Relative al Gauss-DPLL

Diversamente da altri algoritmi derivati, direttamente od indirettamente, dal DPLL, nei quali generalmente si ha a che fare con il solo insieme delle clausole OR, in questo ci troviamo ad avere 4 insiemi fondamentali:

- Assegnazioni
- Selezioni
- Xor Clausole
- Or Clausole

elencati in ordine decrescente di influenza nella velocizzazione nel calcolo del modello (funzionale o unico) è proprio quello delle Assegnazioni, seguito dall'insieme delle clausole XOR.

E' conveniente, quindi, prendere in considerazione (nel caso di una euristica basata su BCP) i seguenti parametri:

$a(v)$: numero delle assegnazioni (o incremento rispetto a prima)

$s(v)$: numero delle clausole OR da esattamente 2 letterali (o incremento)

$c(v)$: numero delle clausole OR (o incremento)

$x(v)$: numero delle clausole XOR (o incremento)

da mettere in correlazione tramite coefficienti.

Ho proposto cinque euristiche, caratterizzate, fondamentalmente dalla semplicità di calcolo dei parametri; infatti, dato l'ingente utilizzo di lookahead durante la computazione, l'euristica riveste sempre meno interesse, man mano che il problema cresce.

Maximum Size of Assignment Set

Il calcolo più semplice di tutti, si basa sull'aumento del solo parametro $a(v)$. Nella pratica sperimentale si è visto corrispondere all'euristica "*first open proposition*"

Minimum Medium Or Clause Lenght

Dato il grosso dispendio di spazio e di energie computazionali per il trattamento delle clausole Or, questa euristica sceglie di splittare prima sulle variabili le quali generino clausole di dimensione minore.

Maximum Medium Occurrence in Xor Clauses

Cerca di privilegiare l'azione della riduzione di Gauss, cercando di splittare sulle proposizioni che generino il maggior numero di occorrenze medie di letterali nelle clausole Xor.

Minimum Medium Dimension of the Xor Clause

Cerca di privilegiare gli splitting la cui assegnazione causa:

- il maggior numero di assegnazioni possibili
- il maggior numero di riduzioni di gauss possibili

attraverso la misura delle dimensioni dell'insieme delle clausole Xor

Maximal Added Constraint

Questa euristica è la più accurata. Tenta di riprodurre una ragionamento ispirato alla euristica di Jeroslow-Wang (cfr. [Jeroslow and Wang]), cercando di stimare l'impatto dell'aumento dei vincoli. Il ragionamento che si è fatto è il seguente. Ogni tipo di clausola (assegnazione, Or etc.) introduce un numero di vincoli dipendente dal numero di letterali in essa contenuti, ma diverso da tipo a tipo, inversamente proporzionale al numero di soluzioni che ammette, ovvero, ponendo n il numero di letterali presenti:

- assegnazioni: 1 vincolo, 1 soluzione
- clausole or: 1 vincolo, $2^n - 1$ soluzioni
- clausole xor: n vincoli, $2^{(n-1)}$ soluzioni
- selezioni: $2 \times n$ vincoli, $2^{(n-2)}$ soluzioni

Trascurando i vincoli generati dalla presenza di un letterale in più clausole anche di tipi diversi), calcolando i constraint medi per ogni insieme, sulla base della dimensione media delle clausole in essa presenti e della dimensione stessa degli insiemi e normalizzando rispetto ai vincoli generati dall'insieme delle clausole Or:
$H(v) = a(v) \times 2^{(n_o)} + x(v) \times 2^{(n_o - n_x + 1)} + s(v) \times 2^{(n_o - n_s + 2)} + c(v)$
dove:

- n_o dimensione media delle clausole Or
- n_x dimensione media delle clausole Xor
- n_s dimensione media delle Select

3.4 Ottimizzazioni

In questo paragrafo sono descritti gli interventi di modifica all'algoritmo di base, volti al miglioramento delle prestazioni computazionali, non legati ai particolari implementativi, per i quali si rimanda ad un capitolo successivo ed inerenti la struttura portante della procedura. Le modifiche apportate per incrementare l'efficienza delle singole regole regole d'inferenza sono state descritte precedentemente.

3.4.1 Semplificazione Iniziale

Subito dopo l'acquisizione della formula, viene effettuata una lunga operazione di calcolo, volta a ridurre il più possibile gli insiemi delle clausole. Tale semplificazione viene eseguita tramite una procedura che alterna computazione diretta, tramite le regole di inferenza, alla ricerca di nuove clausole ed assegnazioni, tramite la ben nota funzione di Lookahead a due livelli.

Questa operazione, che risulta molto onerosa (arrivando a superare di ben oltre la metà del tempo totale di calcolo, nel caso di istanze di input complesse), consente tuttavia di ridurre notevolmente la durata complessiva della ricerca, mediante la scoperta di assegnazioni contraddittorie.

3.4.2 Lookahead a due Livelli durante la Fase di Computazione

Come già detto in precedenza, si è optato per l'utilizzo di questa procedura anche durante la normale computazione, non solo per il calcolo delle Priorità Euristiche. Con questa funzione è infatti possibile:

- scoprire assegnazioni che portano in contraddizione
- aggiungere nei vari insiemi, la parte comune ai due rami delle assegnazioni sulla variabile corrente

3.4.3 Sussunzione Esplicita delle Clausole OR

Poichè si era riscontrato, sperimentalmente, un ingente aumento, ad ogni iterazione del lookahead, del numero delle clausole Or, al momento dell'inserimento delle clausole comuni ai due rami della valutazione nella formula originaria, è stata aggiunta questa procedura.

Siano 2 insiemi di clausole Or C e D, supponiamo di dover effettuare l'inserimento di ogni clausola d nell'insieme C, per ogni inserimento deve essere effettuato il controllo:

```
for_all (c in C) {
  if (d is a subset of c) {
    delete c from C;
    insert d in C;
  }
}
```

Questa procedura non era molto efficiente, infatti posto n_d il numero delle clausole in D e n_c il numero delle calusole in C la complessità era $O(n_c \times n_d)$. Sfruttando le proprietà di ordinamento degli insiemi di calusole proprie della implementazione e considerando che, qualora $d \subset c \Rightarrow min(d) \geq min(c)$, si è modificato la procedura nel modo seguente:

```
fast_subsumpt (orclauset C, orclauset D) {
 for_all (d in D) {
  for_all (c in C) {
   if (min (d) < min (c))
    break;
   else
```

```
    if (d is subset of c) {
     delete c from C;
     insert d in D;
    }
   }
 }
}
```

3.4.4 Insieme delle Occorrenze delle Clausole

Per aumentare la velocità di esecuzione di funzioni quali la lookahead e le funzioni di priorità per il calcolo delle euristiche, è stao introdotto, per ogni formula, un insieme contenente il numero di occorrenze di ogni variabile, per ogni tipo di clausola, aggiornato ad ogni operazione eseguita sugli insiemi della fomula stessa.

3.4.5 Definizione della Calculate

La ottimizzazione che ha richiesto maggiore tempo è stata la definizione della struttura della funzione calculate() (definita nei dettagli in un paragrafo precedente).

Esperimenti

28 maggio 2002

In questo capitolo vengono presentate le verifiche sperimentali effettuate.

I tests sono stati eseguiti su un calcolatore non molto potente[1], per questo i tempi di calcolo potranno risultare più lunghi di quelli esposti nei siti di benchmark (e.g. il sito SAT-EX http://www.satex.org).

Gli esperimenti hanno richiesto molto tempo, anche perchè si è preferito avviarne non più di uno alla volta, in modo da dedicare le risorse di sistema e presentare lo stesso scenario di azione per tutti i programmi.

3.5 Confronto sui DIMACS Parity Problems

Si è ritenuto opportuno confrontare i risultati ottenuti dal tame, con due importanti solutori, basati anch'essi sulla pocedura DPLL:

- Posit ([Freeman, 1995]): questo solver è stato scelto per le alte performance ottenute sui problemi di piccole dimensioni, nonchè sui random tests e poichè presenta, come si è cercato di realizzare anche sul tame, sebbene in altro modo, un meccanismo di semplificazione iniziale davvero efficiente.
- Simo ([Giunchiglia et al, 2000]): questo solver è stato scelto per via della sua alta efficienza (è stato infatti ingegnerizzato dalla Intel Corporation ed è alla base di importanti Industrial Model Checker, come il Thunder), soprattutto per quanto riguarda le strutture dati ed anche perchè "incorpora" le migliori tecniche ed accortezze implementative relative al DPLL: lookahead, conflict-directed-backjumping, learning.

Le istanze di test di confronto prescelte sono stati i DIMACS Parity Problem, per testare la efficienza dell'algoritmo proposto su un tipo di problema che includesse l'utilizzo di logica affine.

3.5.1 Risultati

Vengono qui riportati i risultati degli esperimenti svolti sui DIMACS Parity Problems, considerando i tempi di computazione, la memoria impiegata e, laddove possibile, il numero di nodi visitati ed il numero di variabili eliminate o assegnate durante la semplificaione della formula iniziale.

In ognuna delle tabelle le istanze sono state riportate unificate per tipologia. Ad esempio con "Par8-x*.cnf" si intendono tutte e dieci le istanze (le cinque di base più le cinque compresse) del DIMACS Parity Problem su 8 bit.

Tempi di Computazione.

In questa sezione sono riportati i tempi di computazione. Per il tame ho incluso una colonna aggiuntiva, significativa per le prime due classi di problemi, per evidenziare il tempo di effettivo calcolo, scorporando il tempo necessario al parsing[2] dall'input.

[1] Si tratta di un portatile con processore AMD K6 400 MHz, 188 MB di RAM e S.O. Linux Slackware 8.0

[2] Il processo di parsing del tame è notevolemente più lento rispetto agli altri. Ciò, a mio avviso è dovuto a:

- maggiore complessità del formato
- maggior numero di caratteri da leggere
- utilizzo di prodotti esterni (Bison++ e Flex++) per la produzione del codice di scansione della formula di input

Osservo esplicitamente che, dato che tame accetta in ingresso un formato, cosiddetto "Human Readable", diverso dallo standard DIMACS cnf, è stato dunque necessario utilizzare degli script di conversione.

	Posit 1.0	Simo 2.0	Tame 1.0	Tame 1.0 (no parser)
Par8-x*.cnf	2 sec.	4 sec.	1 min.	9 sec.
Par16-x*.cnf	1 min	15 min	75 min	66 min
Par32-1*.cnf	> 2 giorni	10 ore	6,5 ore	6,5 ore
Par32-2*.cnf	> 2 giorni	11 ore	7 ore	7 ore

nonostante i risultati veramente drammatici del tame ottenuti sulle istanze da 8 e 16 bit, dipendenti soprattutto dall'elevato uso di lookahead, spesso senza risultati immediati, durante la computazione, la situazione si è poi invertita nell'affrontare le istanze a 32 bit.

Risulta evidente il vantaggio di avere una semplificazione iniziale, anche dal confronto tra Posit e Simo, ed è stata appunto per merito di essa se il Tame è riuscito a superare il Simo sulle istanze da 32 bit.

Osservo esplicitamente, che le semplificazione effettuata da Tame, mano mano che aumenta la dimensione del problema, aumenta la durata percentualmente relativa alla durata totale della computazione (nel caso del parity 32 la semplificazione è durata circa un terzo di tutto il tempo totale).

Memoria Impiegata

In questa sezione è riportata la quantità media[3] di risorse di memoria impiegate, durante la risoluzione del DIMACS Parity32. E' da osservare che la quantità è rimasta pressocchè invariata durante la computazione, apparte qualche piccola oscillazione. Ricordo che il Posit non è riuscito a risolvere i problemi dopo due giorni di computazione

	Posit 1.0	Simo 2.0	Tame 1.0
Parity32	1,6 MB	4,3 MB	5,5 MB

Ho voluto riportare questa tabella per vedere eventuali ripercussioni, considerata la struttura dei solver molto diversa. Il dato macroscopico è che nessun solver consuma, almeno per questo tipo di problemi, un quantitativo di risorse tali da bloccare il sistema.

Risulta evidente la maggiore dimensione occorrente agli algoritmi che effettuano lookahead (Simo e Tame), per via delle copie "di calcolo" della formula originale da allocare.

Le strutture dati del tame, che ricordiamo prevede più insiemi di clausole rispetto ai solver "standard" basati su DPLL, benchè non ottimizzate come quelle del Simo, non richiedono, tuttavia, un eccessivo ammontare di memoria: la differenza[4] è dovuta, praticamente, alla dimensione dell'eseguibile (tame: 900 KB ca, Simo: 90 KB ca.), che deriva proprio dalla presenza della logica di controllo per la gestione di tutti gli insiemi in esso presenti.

Nodi Visitati

In questa sezione sono riportati, per la classe dei Parity16, il numero di nodi visitati dagli algoritmi che formiscono questo dato

[3] media temporale sulla singola istanza e fra le istanze. Questo dato ha senso in quanto i consumi di memoria erano praticamente costanti.

[4] sia Simo che Tame sono scritti in C++, mentre Posit è in C.

	Posit 1.0	Tame 1.0
par16-1	3783	132
par16-1-c	4448	2
par16-2	138	2
par16-2-c	221	4
par16-3	1744	71
par16-3-c	5894	3
par16-4	931	3
par16-4-c	2447	129
par16-5	1373	4
par16-5-c	801	9

risulta evidente il netto abbattimento del numero di nodi visitati, da parte del tame, benche esso sia scorrelato statisticamente dal numero di visite effettuate dal Posit, pe via del metodo completamente diverso di processare la formula in input

Variabili Assegnate/Eliminate nella Semplificazione Iniziale

Posit e Tame effettuano una semplificazione iniziale della formula, in modo di cercare di limitare il tempo impiegato nella computazione successiva.

	Posit: Var. Elimin.	Tame: Var. Assegn.
par16-1	553	408
par16-1-c	4	0
par16-2	519	383
par16-2-c	4	0
par16-3	533	412
par16-3-c	4	0
par16-4	541	396
par16-4-c	4	0
par16-5	525	388
par16-5-c	4	0

Il funzionamento della semplificazione iniziale è molto differente tra i due solver, potremmo dire contrario, infatti Posit tenta di espellere variabili, attraverso un meccanismo derivato da una restrizione "ad hoc" della Elimination Rule presente nella originale DP (cfr. [Freeman, 1995]par. 7.9), mente Tame va alla ricerca di assegnazioni possibili, su variabili diverse rispetto a quelle trattate da Posit, per mezzo delle contraddizioni trovate dai calcoli della lookahead). La "espulsione" delle variabili, nel Tame, interviene in un altro ambito: nella definizione della soluzione.

Ciononostante, i dati mostrano comunque una correlazione tra i risultati ottenuti tra i due solver, che sta a simboleggiare l'effettiva utilità di un processamento iniziale (ciò che purtroppo manca al Simo). Evidentemente il modo con cui vengono formulati alcuni problemi nasconde intrinsecamente una ridondanza (non è un caso che sulle istanze "compressed" non ci siano state praticamente semplificazioni da fare) che conviene eliminare, in modo da semplificare la ricerca successiva delle soluzioni.

3.6 Esperimenti sulla Criptoanalisi del DES

I tests sono stati eseguiti eclusivamente sul solver qui presentato, onde verificare il diverso comportamento a seconda delle opzioni impartite all'avvio della computazione. Le istanze sono state generate a partire da dieci chiavi, simulando un attacco a plaintext conosciuto, fatto passare per una, due e tre iterazioni del DES.

3.6.1 Risultati

Lookahead durante la Computazione

Si è valutata la convenienza, in funzione dei tempi medi totali, di eseguire la funzione lookahead durante la normale computazione in funzione del tempo medio complessivo

	Tame	Tame (no lookahead)
des 1 round 1 block	15 m. 9 s.	8 m. 2 s.
des 2 round 1 block	30 m. 15 s.	15 m. 32 s.
des 3 round 1 block	12 h. ca.	8 h. 13 m.

risulta evidente, almeno nelle istanze meno grandi del des, la convenienza di non inserire la funzione lookahead all'interno della computazione. La maggior parte del tempo speso nel calcolo dei lookahead non porta al ritrovamento di contraddizioni e, inoltre, le clausole aggiunte alla formula originale, comuni ai due rami dello split sono in numero esiguo (dell'ordine di 5-6 clausole OR, per ogni applicazione della lookahead, 1 XOR ogni 20 applicazioni e 1 Assegnazione ogni 35 applicazioni ca.), contrariamente a quanto accadeva con i DIMACS Parity Problems.

Questo comportamento ha portato ad inserire l'opzione per disabilitare il ricorso alla funzione lookahead durante il calcolo. I calcoli seguenti sono stati effettuati con questa opzione.

Split Selettivo

Si è valutata la convenienza, in funzione del tempo medio totale necessario al calcolo, di eseguire lo split in modo selettivo, solo su variabili che "hanno un determinato nome", ovvero rappresentano uscite di particolari blocchi funzionali all'interno della procedura di criptazione del DES. Nelle colonne sono indicate tra parentesi le variabili sulle quali si è eseguito lo split.

	Tame	Tame (K)	Tame (S)	Tame (X)	Tame (M)	Tame (R)
des 1 round 1 block	8 m. 2 s.	5 m.40 s.	5 m. 9 s.	3 m.6 s.	6 m. 13 s.	6 m. 4 s.
des 2 round 1 block	15 m. 32 s.	10 m. 14s.	9 m. 36 s.	7 m. 49 s.	12 m. 39 s.	11 m. 21 s.
des 3 round 1 block	8 h. 13 m.	6 h. 23 m.	5 h. 49 m.	5 h. 12 m.	7 h. 31 m.	7 h. 5 m.

la tabella evidenzia l'utilità di eseguire uno split selettivo, specialmente nei confronti delle variabili 'X', 'S', ed 'K'. Questo comportamento era comunque prevedibile, per il des da 3 round, infatti le clausole XOR contengono esattamente tre variabili scelte tra quelle precedentemente menzionate e dunque ad ogni assegnazione di una di quelle variabili si generano nuove assegnazioni. Questo comportamento viene intercettato dalla funzione euristica, ma con ovvio dispendio di tempo in cumputazione.

Le istanze da uno e due round contengono solo le clausole OR che derivano dalle equazioni AND. Questo giustifica l'andamento pseudo-linerare nel passaggio dalla prima alla seconda classe di istanze. I tempi di calcolo assai più onerosi nelle istanze della terza classe sono dovuti alla computazione delle clausole OR, più che ai calcoli riduttivi sulle XOR, come risulta dai log dell'applicazione

Capitolo 4

Architettura del Software

Questo software, come precedentemente illustrato, nasce come realizzazione pratica di un SAT Solver, basato sul DPLL, con l'aggiunta dell'uso di un adattamento della riduzione di Gauss per il trattamento delle clausole XOR (anzichè tradurle dispendiosamente in clausole OR), descritto in [Baumgartner and Massacci, 2001], applicato ad un caso particolare: la criptoanalisi.
In questo capitolo vengono descritti i principali vincoli di progetto, gli strumenti ed i linguaggi utilizzati, nonchè le interfacce di input e output, l'ambiente operativo e la funzionalità complessiva, onde poter comprendere il funzionamento generale ed, eventualmente, poter operare gli aggiustamenti necessari per un uso qui non previsto.

4.1 Requisiti

In questa sezione sono stati analizzati i principali vincoli di progetto, che hanno costituito le linee guida durante la creazione del software. Questi sono divisi in requisiti generali, ovvero qualità relative al progetto nella sua interezza, con particolare riguardo all'algoritmo centrale, e requisiti specifici, riguardanti principalmente l'acquisizione dei dati criptoanalitici.

4.1.1 Requisiti Generali

Efficienza

La sfida é quella di creare un software capace di portare avanti un attacco criptoanalitico a conoscenza completa (ovvero sia il *cypher text* che il *plain text*) su piattaforme hardware non dedicate e di non elevatissime prestazioni. Si é cercato, quindi, di prestare particolare importanza a questo aspetto, cominciando con lo scegliere un linguaggio particolarmente veloce come il l'ANSI/C++, con l'ausilio della STL. Per la scelta e l'implementazione dei tipi di dati (soprattutto la classe `var` che rappresenta le variabili e dunque viene inclusa nei contenitori) si é cercato di scegliere una forma che preservasse vuoi l'impegno di memoria, vuoi il carico di istruzioni da eseguire per la ricerca, le sostituzioni etc.

Modularità

Per suddividere nel modo migliore i vari componenti, in modo da non perdere di vista la funzionalitá complessiva, ma soprattutto di "disaccoppiare" la struttura fondamentale dell'algoritmo dalle strutture utilizzate, si è fatto largo uso di classi e funzioni che creassero uno "strato intermedio" tra i tipi di basso livello e le strategie computazionali di livello alto. In particolare si è cercato di mascherare alcune funzioni semanticamente pericolose, definite nei contenitori standard della STL.

Riusabilità

Come detto in precedenza, questo software viene applicato alla criptoanalisi, ma implementa un algoritmo che mantiene una validità ben più generale. Ridefinendo alcune macro che fissano i valori ed i tipi minimi (per preservare la efficienza nell'occupazione di memoria) della classe `var` e modificando all'uopo il parser, é possibile riusare il codice in altri ambienti in cui occorrano Or clause e Xor clause insieme.

Ogni classe é stata definita in un file header proprio, corredata da ampia documetazione (vedere capitolo successivo) che ne specifica interfacce, implementazione e metodi.

Portabilità

Si é fatta molta attenzione a non introdurre elementi esterni all' ANSI/C++ '97, di modo che il codice potesse essere portabile. Nonostante il software sia stato scritto per essere compilato su piattaforme Unix (testato in ambienti Linux Slackware 8.0, Linux RedHat 6.2, 7.2 e Sun Solaris 5.7), con *gcc*, che utilizza la implementazione della STL della SGI [**?**], non sono stati utilizzati, benchè sarebbero stati di grosso ausilio, strutture dati proprie della SGI-STL.

4.1.2 Requisiti Specifici

Formato dell'Input

Questo software non include la riduzione in clausole (Or e Xor) del problema da analizzare, che deve essere trattata da un programma esterno. Va quindi progettato un parser che crei la interfaccia di Input nei confronti delle Output generato dal riduttore in clausole

Bison++

E' necessario questo tipo di parser, come vedremo più avanti, per generare l'analizzatore lessicale. Sfortunatamente questo pacchetto solitamente non è istallato sui sistemi normalmente, essendo sufficiente, per la maggior parte degli scopi, l'originale Bison (cfr. [Donnelly and Stallman, 1995]).

4.2 Librerie di Appoggio (STL e LEDA)

Le librerie di appoggio sono state necessarie per usufruire delle implementazioni di tipi di dati ed algoritmi efficienti ed ampiamente documentate a tutto vantaggio della efficienza e della riusabilità, porgendo l'attenzione alla struttura complessiva del progetto. D'altronde un codice C++ moderno non può prescindere da librerie fondamentali, potenti e flessibili come la STL o LEDA.

In particolare servivano una libreria che implementasse in modo efficiente alcuni tipi di dati basici ma indispensabili come set, double ended queue e mappe.

Inizialmente il codice era stato realizzando usufruendo della libreria proprietaria LEDA. Successivamente sia per ragioni di efficienza (principalmente nelle iterazioni), sia per garantire maggiore portabilità (come detto in precedenza la STL standard fa parte dell'ANSI/C++ '97), sia per questioni di licenze, si é trasportato il codice su STL.

La produzione della documentazione del codice tuttavia, resta ancora basata sul modello proposto da LEDA (si veda appendice).

4.2.1 La Standard Template Library

Sicuramente si tratta di una delle funzionalità aggiunte più importanti al C++, la cui inclusione ha presentato sforzi non indifferrenti. Inizialmente è stata sviluppata alla HP, da Alexander Stepanov (cfr. [SGI, 2002]) e Meng Lee, a partire dal 1992.
Come dice il nome stesso si tratta di una libreria sviluppata mediante classi con largo uso di Template, in modo da poter essere istanziata su quasi tutti i tipi di dati definibili dall'utente, che si pone l'obbiettivo di rappresentare uno standard di riferimento ed un punto di partenza per tutti i programatori C++, che fanno largo uso di tipi realizzati tramite classi.
Ha avuto una importanza basilare nella definizione del C++ standard, come lo conosciamo oggi: basti pensare che la introduzione dei template si deve ad essa (si legga a proposito la interessante intervista ad Alexander Stepanov, sempre nel sito della SGI, dedicato alla STL).
Per l'uso della STL, si rimanda al già citato sito della SGI (http://www.sgi.com/tech/stl) ed anche [Meyers, 2001], [Eckel et al, 2000].

4.3 L'analisi sintatica (Bison++ e Flex++)

Benchè il formato di input fosse relativamente semplice, si è pensato di ricorrere ad un parser ed uno scanner per generare l'analizzatore dell'input. La scelta è ricaduta su Bison++ (e conseguentemente Flex++) poichè era necessario che il parser:

- maneggiasse correttamente oggetti C++
- generasse codice rientrante

Poichè Bison genera codice impossibile da compilare in C++, anzichè in C, si è optato per una soluzione alternativa. Le possibilità, usufruendo di software di dominio pubblico erano 2:

1. *Bison2.6A,* lo script *mkparserclass* della suite "Andrew" della Università Carnegie Mellon e successivamente *Flex++* (incluso nel Flex per C)
2. Bison++ e Flex++

la prima soluzione aveva il vantaggio di poter usare la stessa sintassi per scrivere il codice del parser, ma lo svantaggio di essere più macchinoso e necessitare comunque la istallazione di un parser non presente nel sistema; la seconda aveva il vantaggio di essere sicuramente maggiormente diffuso e documentato (si veda ad esempio [Brokken and Kubat, 1997]), oltre ad essere più lineare nell'utilizzo, presentando la difficoltà della sintassi leggermente differente rispetto al software da cui deriva.
Alla fine si è optato, come detto in precedenza, per la seconda soluzione.

4.3.1 Formato di Input

L'input è in un formato denominato "Human Readable Format", che, come si intuisce dal nome, rende le formule facilmente intelleggibili, lasciando separate le equazioni provenienti da clausole Or, Xor, And ma rendendo notevolmente più onerosa la sua analisi.
Ogni riga contiene una equazione diversa. Il formato generico di una riga è:
<LITERAL>=<OPERATOR>(<LITERAL_LIST>)
dove:
<LITERAL> ::= [NOT(]<NAME><BIT_NUMBER>,<CYCLE_NUMBER>,<BLOCK_NUMBER>[)]
<OPERATOR>::= XOR | OR | AND
<LITERAL_LIST> ::= <LITERAL> | <LITERAL>;<LITERAL_LIST>

4.4 Funzionalità complessiva

Il software è stato sviluppato e testato in ambienti Unix, anche se, con opportune modifiche, dovrebbe poter girare su molte altre piattaforme, poichè ogni componente o programma esterno da esso utilizzato (non ultimo lostesso Bison++) è stato sviluppato o portato su un gran numero di piattaforme.

La interfaccia si presenta, unicamente, a linea di comando. Il meccanismo di comunicazione dei dati di I/O e messaggistica avviene per mezzo della console sulla quale si lavora o tramite file.

4.4.1 Input

Ovviamente la prima operazione che deve effettuare il programma è l'inserimento delle equazioni da input. Per default il programma si aspetta input da tastiera, ma specificando l'opzione -I <FILE_INPUT> è possibile far leggere dal file specificato.

Nel caso il file di input contenga un errore, il parser smette di leggere l'input, uscendo e facendo processare al programma, quindi, le sole clausole precedenti l'errore sintattico.

4.4.2 Output

Le soluzioni, insieme ad altre (poche) informazioni riguardanti la durata della computazione etc., viene inviata per default allo stdout della console utilizzata. E' possibile specificare un file di output, tramite l'opzione -O <OUTPUT_FILE>; il file in questione viene aperto in append, onde consentire:

- di inserire tutte le soluzioni per la stessa istanza
- di inserire soluzioni anche per più istanze (cioè usare sempre lo stesso file di output per una stssa campagna di esperimenti)

4.4.3 Storing

Il programma salva parte dello stato computazionale, rappresentato dall'insieme delle variabili assegnate, in un file, che se non specificato esplicitamente tramite l'opzione -S <STORE_FILE>, ha come nome fisso "./tame.store".

Questo salvataggio è stato implementato pensando di realizzare la possibilità di far ripartire la computazione da esso, nel caso di interruzione forzata del calcolo, dovuta a cause esterne.

4.4.4 Logging

E' prevista una fuzionalità di logging, a livelli, per il controllo del funzionamento dell'applicativo, soprattutto in relazione alla efficacia delle opzioni di calcolo specificate. Sono previsti 10 livelli di logging (da 0 a 9), pure se, nel caso in cui si stiano facendo dei confronti di velocità o si stiano processando istanze complesse, si consiglia di limitarne l'utilizzo, in quanto riduce di parecchio la velocità computazionale, per via del grosso numero di chiamate di sistema effettuate (con il livello massimo il tempo necessario alla computazione è ca. 40 volte il tempo necessario con il livello minimo).

4.4.5 Specificazione dei 2 Livelli di LookAhead

E' possibile scegliere esplicitamente se usare, durante il procedere del calcolo, la funzione lookahead a uno (default) o due livelli. Sperimentalmente, infatti, si è verificato che in presenza di istanze più semplici conviene un lookahead meno completo.

4.4.6 Esclusione della Riduzione di Gauss

E' possibile escludere il processamento delle clausole XOR tramite la regola di riduzione di Gauss. E' però opportuno che la formula in ingresso non contenga clausole XOR, altrimenti l'algoritmo non è completo.

4.4.7 Scelta delle Euristiche

E' possibile scegliere esplicitamente l'euristica da utilizzare tra le cinque proposte ed illustrate precedentemente. E' anche relativamente facile aggiungerne altre, modificando opportunamente i file "./include/heuristics.h" e "./src/heurisitcs.C".

Capitolo 5

Realizzazione: Strutture Dati e Algoritmi

In questo capitolo vengono descritte le fondamentali strutture dati che è stato necessario realizzare, in particolare dando enfasi:

1. alla interfaccia, nella speranza di un riuso futuro
2. alle motivazioni che sono dietro alle scelte implementative, con particolare riguardo alla scelta delle classi della STL sulle quali ogni nuovo oggetto è basato.

La documentazione relativa ad ogni oggetto è generata da script in PERL, che processano i commenti dei file header (vedi appendice A).

5.1 Formulae (formula)

1. Definition

Una istanza F del tipo *formula* é un insieme contenente gli elementi:

- Or-clause Set
- Xor-clause Set
- Assignment Set
- Selected-Xor Set
- Variables Occurrences Vector

#include < *../../include/formula.h* >

2. Creation

formula F; crea una istanza di F e la inizializza con insiemi vuoti.

3. Operations

bool	F.bottom( )	Restituisce true se in C o X é presente una clausola contraddittoria, false altrimenti.
void	F.unset_changed( )	Setta il flag che indica che ci sono stati cambiamenti a false.
void	F.set_changed( )	Setta il flag che indica che ci sono stati cambiamenti a true.
bool	F.changed( )	Legge il flag che indica che ci sono stati cambiamenti.
void	F.set_lookahead(*int* i)	Valorizza il contatore del numero di lookahead con l'ultimo valore.
int	F.get_lookahead( )	restituisce il numero di lookahead effettuati l'ultima volta.
void	F.set_calculate(*int* i)	Valorizza il contatore del numero di calculate con l'ultimo valore.
int	F.get_calculate( )	Restituisce il numero di calculate effettuati l'ultima volta.
void	F.set_restart(*int* i)	Valorizza il contatore del numero di restart effettuat con i.
int	F.get_restart( )	Restituisce il numero di restart effettuati.
int	F.get_solution_number( )	Restituisce il numero di soluzioni finora trovate.
bool	F.resync_vset( )	Risincronizza l'insieme $F.V$ in base alle variabili effettivamente contenute nelle clausole di F con le occorrenze.
bool	F.occur(*var* a)	restituisce se $abs(a)$ ricorre in qualche formula di F.
bool	F.empty( )	restituisce se gli insiemi di F sono tutti vuoti.
int	F.getOrPosOccurrence(*const var* a)	restituisce le occorrenze positive di a nelle Or clausole.
int	F.getOrNegOccurrence(*const var* a)	restituisce le occorrenze negative di a nelle Or clausole.
int	F.getXorOccurrence(*const var* a)	restituisce le occorrenze di a nelle Xor clausole.
int	F.getSelOccurrence(*const var* a)	Restituisce le occorrenze di a nelle *Select Xor*.
int	F.getAssOccurrence(*const var* a)	restituisce le occorrenze negative di a nelle Xor clausole.

void	F.AddOr(*orclause* c)	Aggiunge una clausola Or nella formula, aggiornando le occorrenze delle variabili.
void	F.AddXor(*xorclause* x)	Aggiunge una clausola Xor nella formula, aggiornando le occorrenze delle variabili.
void	F.AddSel(*const var* v, *xorclause* x)	Aggiunge una selezione Xor nella formula, aggiornando le occorrenze delle variabili.
void	F.AddAssign(*var* A, *var* B)	aggiunge in $F.Ass$ una nuova assegnazione, $abs(A) := \pm B$, eventualmente eliminando da $F.S$ la definizione di A.
void	F.DelOr(*orclause* c)	Elimina una clausola Or nella formula, aggiornando le occorrenze delle variabili.
void	F.DelOr(*orclauset*::*iterator itc*)	Elimina la clausola Or nella formula, cui si riferisce l'iteratore, aggiornando le occorrenze delle variabili.
void	F.DelXor(*xorclause* x)	Elimina una clausola Xor nella formula, aggiornando le occorrenze delle variabili.
void	F.DelXor(*xorclauset*::*iterator itx*)	Elimina la clausola Xor nella formula, cui si riferisce l'iteratore, aggiornando le occorrenze delle variabili.
void	F.DelSel(*selectxor*::*iterator its*)	Elimina la selezione Xor nella formula, cui si riferisce l'iteratore, aggiornando le occorrenze delle variabili.
formula&	$F = G$	Assegna G a F e restituisce F.
bool	F != G	Ritorna se le 2 formule sono diverse.
formula &	F += G	Aggiunge a F gli elementi di $G.C$, $G.X$, $G.S$ non presenti in F.
formula	F & G	Restituisce l'intersezione insiemistica di G e F.
ostream & *ostream*& *os*	$\ll F$	Stampa su *os* il contenuto della formula, nell'ordine: *Orclauset X* *Xorclauset C* *SelectXor S* *Assignmap A* *Variables Occurrences Vector V*
void	F.PrintSolution(*ofstream*& *of*)	Stampa la soluzione su *of*, se aperto, altrimenti a video.

void *F*.PrintInsolution(*ofstream*& *of*)

Stampa un messaggio di insoddisfacibilitá della formula su *of*, se aperto, altrimenti a video.

Non-Member Functions

4. Implementation

Le Formule sono implementate tramite sets di Clause e sets di Assign. I tempi di esecuzione delle operazioni dipendono dalla implementazione dei sets.

5.2 Assegnazioni Arbitrarie di Variabili (lassign)

1. Definition

Una istanza del tipo di dati *lassign* rappresenta un insieme, gestito a coda-pila di assegnazioni arbitrarie di letterali a valori costanti. Ogni assegnazione è contrassegnata dall'etichetta *toggable* a seconda se sia ancora invertibile o meno. Infatti se da questa assegnazione dovesse provenire una contraddizione deve essere negata.

#include < *../../include/lassign.h* >

2. Creation

void	*B*.add(*var LHS*, *var RHS* = *TRUE*, *bool toggle* = *true*)	inserisce una nuova assegnazione, del tipo $LHS := RHS$. Per default $RHS = TRUE$, l'assegnazione é arbitraria.
bool	*B*.Find(*var v*)	Restituisce se esiste una assegnazione (toggable o certa) per v.
void	*B*.AddCostantAssignment(*assignmap Ass*)	Aggiunge tutte le assegnazioni costanti che si trovano in Ass, provenienti dai calcoli semplificativi, etichettandole come NON backtrackable.
void	*B*.Reorder( )	Cancella le variabili, ultime inserite in ordine cronologico, che non hanno settato il flag di Backtrack (ovvero sono state provate in entrambi i valori). Inverte il segno della ultima variabile "backtrackable".
ostream &	*ostream*& *os* $\ll$ *B*	Stampa su *os* il contenuto di B variabile per variabile, nel formato: <*var1*> -¿ <*const_value1*> [not] toggable ... <*varn*> -¿ <*const_valuen*> [not] toggable.

5.3 Assegnazioni Base di Variabili (bassign)

1. Definition

Una istanza del tipo di dati *bassign* rappresenta una assegnazione di letterali a valori costanti.

#include < *../../include/bassign.h* >

2. Creation

bassign BA(*bassign bb*);
Costruttore di copia

3. Operations

void	*BA*.set_label( )	Setta il flag di backtrack.
void	*BA*.unset_label( )	Resetta il flag di backtrack.
bool	*BA*.label( )	Restituisce se la assegnazione é di backtrack oppure no.
bassign&	inverse( )	Restituisce la assegnazione cambiata di segno.
bassign & BA = const bb		Assegnazione.
ostream & ostream& os $\ll$ *BA*		Stampa su *os* il contenuto di *BA*, nel formato: <*var1*> -¿ <*const_value1*>

5.4 Orclausets (orclauset)

1. Definition

Una istanza C del tipo di dati *orclauset* rappresenta un insieme di clausole Or della logica booleana.

#include < *../../include/orclauset.h* >

2. Creation

orclauset C; crea una istanza di C e la inizializza con insiemi vuoti.

3. Operations

void	C.set_bottom( )	Setta il flag che indica la presenza di una contraddizione.
void	C.unset_bottom( )	Resetta il flag che indica la presenza di una contraddizione.
bool	C.bottom( )	Restituisce true se in C é presente una clausola contraddittoria, false altrimenti.

Updating Member Function

void	C.add(*orclause* c)	inserisce la clausola c nell'insieme di clausola C.
void	C += D	Assegna a C l'unione insiemistica $C \cup D$, effettuando la sussunzione delle clausole contenute in C, ovvero: $\forall d \in D$: if $\exists c \in C : d \subset c, C.del(c), C.add(d)$ if $\exists c \in C : d = c$ skip to the next d if $\nexists c \in C : d \subset c, C.add(d)$
orclauset	C & D	Restituisce l'insieme intersezione insiemistica $C \cap D$

Output Extern Function

ostream & *ostream*& *os* $\ll$ C	Stampa su *os* le clausole presenti nella xorclauset C, separate da un accapo. Utilizza *operator* $\ll$ (*ostream* & *os*, *orclause* x) per stampare ogni elemento.

4. Implementation

Le Orclausets sono implementate tramite set, instanziata al tipo *orclause*. Le funzioni *add*( ), *bottom*( ), *set_bottom*( ) hanno complessitá $O(1)$; le altre funzioni sono $O(n)$.

5.5 Orclauses (orclause)

1. Definition

Una istanza C del tipo di dati *orclause* rappresenta una clausola Or della logica booleana, ovvero un insieme di variabili booleane *var*.

#include < *../../include/orclause.h* >

2. Creation

orclause C; crea una istanza di C e la inizializza con insiemi vuoti.

3. Operations

void	C.add(*var* x)	inserisce la variabile x nella clausola C.
bool	is_complement(*orclause* D)	Ritorna se le 2 clausole, che devono essere binarie, sono complementari, ovvero se data $C = \{a, b\}$ sia effettivamente $D = \{not(a), not(b)\}$
bool	$C == D$	Ritorna se le 2 clausole sono uguali.
ostream & *ostream*& *os* $\ll$ x		Stampa su *os* le variabili presenti nella xorclause, nel formato: {<*var1*>,<*var2*>, ... ,<*varn*>}. Utilizza *operator* $\ll$ (*ostream* & *os*, *const var* & v) per stampare ogni elemento.

4. Implementation

Le Orclauses sono implementate tramite set, instanziata al tipo *var*. La definizione della funzione *add* si è resa necessaria per analogia con la classe *xorclause*. Le funzioni *add*( ), *is_complement*( ) hanno complessitá $O(1)$; gli operatori $<$, $==$, $\ll$ hanno complesitá $O(n)$, dove n è la lunghezza della clausola.

5.6 Xorclausets (xorclauset)

1. Definition

Una istanza X del tipo di dati *xorclauset* rappresenta un insieme di clausole Xor della logica booleana.

#include < *../../include/xorclauset.h* >

2. Creation

xorclauset C; crea una istanza di C e la inizializza con insiemi vuoti.

3. Operations

void	C.set_bottom( )	Setta il flag che indica la presenza di una contraddizione.
void	C.unset_bottom( )	Resetta il flag che indica la presenza di una contraddizione.
bool	C.bottom( )	Restituisce true se in C é presente una clausola contraddittoria, false altrimenti.

Updating Member Function

void	C.add(*xorclause* x)	inserisce la clausola x nell'insieme di clausola C.
void	$C \mathrel{+}= D$	Assegna a C l'unione insiemistica $C \cup D$
xorclauset C & D		Restituisce l'insieme intersezione insiemistica $C \cap D$

Output Extern Function

ostream & *ostream*& *os* $\ll$ X	Stampa su *os* le clausole presenti nella xorclauset C, separate da un accapo. Utilizza *operator* $\ll$ (*ostream* & *os*, *xorclause* x) per stampare ogni elemento.

4. Implementation

Le Xorclauses sono implementate tramite set, instanziata al tipo *xorclause*. Le funzioni *add*( ), *bottom*( ), *set_bottom*( ) hanno complessitá $O(1)$; le altre funzioni sono $O(n)$.

5.7 Xorclauses (xorclause)

1. Definition

Una istanza X del tipo di dati *xorclause* rappresenta una clausola Xor della logica booleana.

#include < *../../include/xorclause.h* >

2. Creation

xorclause X; crea una istanza di X e la inizializza con insiemi vuoti.

3. Operations

void	X.add(*var* x)	inserisce la variabile x nella clausola X, tenendo conto se la variabile x sia giá presente (asserita o negata) all'interno della *xorclause*.
xorclause	not(*xorclause* Y)	Restituisce la xclause cambiata di segno, ovvero: elimina la costante *True* se presente altrimenti nega la prima variabile.
bool	$X == D$	Ritorna se le 2 clausole sono uguali.
ostream & *ostream*& *os* $\ll$ *const x*		Stampa su *os* le variabili presenti nella xorclause, nel formato: {<*var1*>,<*var2*>, ... , <*varn*>}. Utilizza *operator* $\ll$ (*ostream* & *os*, *const var* & *v*) per stampare ogni elemento.

4. Implementation

Le Xorclauses sono implementate tramite set, instanziata al tipo *var*. La definizione della funzione *add* si è resa necessaria poichè si è dovuta ridefinire la funzione di inserimento, in relazione alla presenza o meno della variabile da inserire all'interno della *xorclause*. Infatti, per non trattare le *xorclause* come multiinsiemi, nell'inserimento si deve eseguire uno *xor* e non un *or*, ovvero si possono presentare i 3 casi, per l'inserimento della variabile x nella xorclause X:
$x \notin X$ e $not(x) \notin X$, onde $x \oplus X \equiv x \vee X$
$x \subset X$ e $not(x) \notin X$, onde $x \oplus X \equiv X - x$
$not(x) \subset X$ e $x \notin X$, onde $x \oplus X \equiv True \vee X$

Complessitá delle funzioni

La funzione add ha complessitá $O(\log n)$, la funzione *not*( ) e gli operatori $<$, $==$, $\ll$ hannocomplessitá $O(n)$, dove n è il numero dei letterali di X.

5.8 Mappa delle Xorclause selezionate (selectxor)

1. Definition

Una istanza del tipo di dati *selectxor* rappresenta un insieme di xorclause selezionate, ciascuna avente come chiave la variabile di cui costituisce la "definizione".

#include < *../../include/selectxor.h* >

2. Creation

selectxor S; crea una nuova istanza S del tipo *selectxor* vuota.

3. Operations

int	S.status( )	restituisce il possibile stato di errore dovuto ad una improvvida espansione di S: 0 tutto ok 1 variabile non presente nella *xorclause* da aggiungere. 2 *xorclause* troppo corta. 3 variabile giá definita.
void	S.unset_bottom( )	Resetta la condizione di contraddizione.
bool	S.add(*var* v, *xorclause* x)	Inserisce nella mappa di selezione, la nuova xorclause x. che $\forall$ xorclause $y : v, y \notin S$. *Precondition*: $\pm v \in x$; $x.size() \geq 3$; $\forall$ xorclause $y : v, y \notin S$. In tutti questi casi vengono settati i flag di errore, da leggere con: *int S.status* ( ).
bool	S.yet_defined(*var* v)	ritorna se la variabile v ha giá una definizione in *selectxor*.
void	D	
selectxor	D	
ostream & *ostream*& *os* $\ll$ S		Stampa su *os* le selezioni contenute in *selectxor*, nel formato: <*Var1*> :: = <*Xorclause1*> ... <*Varn*> :: = <*Xorclausen*>.

4. Implementation

Le *selectxor* sono implementate tramite gli standard STL associative container:
template <*class Key*, *class T*, *less*, *allocator*<*T*> *class map*; dove:
class Key è istanziata al tipo *var*
class T è istanziata al tipo *xorclause*

less è la funzione "standard" di ordinamento delle *var*: *bool operator* $<$ (*const var* & *a*, *const var b*)
allocator <*var*> è l'allocatore predefinito dalla STL. La variabile da definire viene imagazzinata con il suo valore assoluto
la definizione viene immagazzinata previo eliminazione della variabile definita e con l'opportuna negazione di una dei letterali in essa contenuti.
In questo modo è possibile eseguire una ricerca piú veloce dei letterali contenuti in un *xorclause* che possiedono una definizione in S, ricercandone il valore assoluto.

Si è scelto di implementare *map* <*var*, *xorclause*> in modo che:

1. il fatto che v sia anche la chiave della singola selection garamtisce che non si inseriscano 2 assegnazioni distinte per lo stesso Atomo.
2. La ricerca del valore da sostituire ad ogni ad ogni occorrenza di v viene ottiizzata dagli algoritmi interni alla STL, in quanto le *map*<*Key*, *T*> sono ordinate in base ai valori della chiave e della funzione *operator* $<$ (*Key*, *Key*). Inoltre poichè le "definizioni" vengono memorizzate per il valore assoluto di v garantisce di poter fare una sola iterazione nella ricerca della variabile nella *xorclause* (e non 2, una per ogni segno!)
3. Non è stato necessario definire un tipo base: *class select* : *public pair* <*var*, *xorcluase*> poichè questa operazione è effettuata internamente alla *map* <*var*, *xorclause*>.

.

Iterazioni Sono possibili tutte le iterazioni implementate nelle *map*<*Key*, *T*>, in particolare:

- con i tipi *selectxor* :: *iterator* e *selectxor* :: *const_iterator* definiti in cicli (i.e. *while*, *for*)
- con l'operatore *operator* []

.

Complessitá La funzione *status*( ) ha complessitá $O(1)$; *add*( ), *yet_defined*( ), $O(\log n)$, l'operatore $\ll$, $O(n)$.

5.9 Mappa di assegnazioni di Variabili (assignmap)

1. Definition

Una istanza del tipo di dati *assignmap* rappresenta un insieme di assegnazioni: $Atomo = Letterale$.

#include < *../../include/assignmap.h* >

2. Creation

assignmap Ass; crea una nuova istanza *Ass* del tipo *assignmap* vuota. Resetta le condizioni di errore.

3. Operations

var	*Ass*.add(*var Atom*, *var Lit*)	Inserisce nella lista di assegnazioni, la nuova assegnazione $Atom = Lit$. Nel caso *Atom* sia negativo viene cambiato segno ai 2 letterali. Restituisce *Atom*.
bool	*Ass*.yet_defined(*var v*)	ritorna se la variabile v ha giá una definizione in *assignmap*.
void	$Ass \mathrel{+}= B$	Assegna a *Ass* l'unione insiemistica $Ass \cup B$
assignmap Ass.-(assignmap B)		Restituisce la differenza $A - B$.
assignmap Ass & B		Restituisce l'intersezione insiemistica $Ass \cap B$.
ostream & ostream& os $\ll$ *Ass*		Stampa su *os* le assegnazioni contenute in *assignmap*, nel formato: <*Atom1*> :: = <*Lit1*> ... <*Atomn*> :: = <*Litn*>.

Funzioni esterne

bool store_Ass(*assignmap*& *Ass*, *assignmap*& *Bss*)

Sposta tutto il contenuto di *Ass* in *Bss*.
Precondition:
$Ass \neq \emptyset$
Bss ottenuta solo per inserimenti successivi di elementi di *Ass*.
Restituisce se è stato effettuato lo spostamento.

4. Implementation

Le *assignmap* sono implementate tramite gli standard STL associative container:
template <*class Key*, *class T*, *less*, *allocator*<*T*> *class map*; dove:

class Key è istanziata al tipo *var*
class T è istanziata al tipo *var*
less è la funzione "standard" di ordinamento delle *var*: *bool operator* $<$ (*const var* & *a*, *const var b*)
allocator <*var*> è l'allocatore predefinito dalla STL.

Le assegnazioni sono implementate tramite *map* <*var*, *var*> in modo che:

1. il fatto che *Atom* sia anche la chiave della singola assegnazione garamtisce che non si inseriscano 2 assegnazioni distinte per lo stesso Atomo.
2. La ricerca del valore da sostituire ad ogni ad ogni occorrenza di *Atom* viene ottiizzata dagli algoritmi interni alla STL, in quanto le *map*<*Key*, *T*> sono ordinate in base ai valori della chiave e della funzione *operator* $<$ (*Key*, *Key*).
3. Non è stato necessario definire un tipo base: *class assign* : *public pair* <*var*, *var*> poichè questa operazione è effettuata internamente alla *map* <*var*, *var*>.

.

Iterazioni Sono possibili tutte le iterazioni implementate nelle *map*<*Key*, *T*>, in particolare:

- con i tipi *assignmap*:: *iterator* e *assignmap*:: *const_iterator* definiti in cicli (i.e. *while*, *for*)
- con l'operatore *operator* []

Sostituzioni Qualora un Atomo sia da "ridefinire" sostituendo cioè il valore della chiave con un altro (i.e. quando si effettua una sostituzione, in seguito ad una nuova assegnazione), per preservare l'ordinamento della mappa, viene prima eliminata la assegnazione da modificare e poi reinserita giá aggiornata.

5.10 Set of Unassigned Boolean Variables (vset)

1. Definition

Una istanza V del tipo di dati *vset* rappresenta l'insieme delle variabili, della logica booleana, con il numero di occorrenze positive e negative nelle clausole, ma non ancora assegnate. La sua definizione si é resa necessaria per semplificarne la ricerca in funzioni come *lookahead*( ) e *split*( ).

#include < *../../include/vset.h* >

2. Creation

vset V; crea una istanza di V e la inizializza all'insieme vuoto.

3. Operations

void	V.addor(*const var* a)	incrementa il numero di occorrenze della variabile a, tenendo conto del segno, nell'insieme V. Nel caso a non sia presente, la inserisce con occorrenza unitaria. Per *Red*( ), *Simp*( ).
void	V.addxor(*const var* a)	incrementa il numero di occorrenze della variabile a, non tenendo conto del segno, nell'insieme *XorClause*. Nel caso a non sia presente, la inserisce con occorrenza unitaria. Per *Red*( ), *Simp*( ).
void	V.addsel(*const var* a)	incrementa il numero di occorrenze della variabile a, non tenendo conto del segno, nell'insieme *Select Xor*. Nel caso a non sia giá presente, la inserisce con occorrenza unitaria.
void	V.addass(*const var* a)	incrementa il numero di occorrenze della variabile a, nelle assegnazioni.
void	V.subor(*const var* a)	decrementa il numero di occorrenze della variabile a nell'insieme delle orclausole. Nel caso a non sia presente esce. Nel caso il numero di occorrenze di a sia esattamente 1, cancella a. Per *Red*( ), *Simp*( ).
void	V.subxor(*const var* a)	decrementa il numero di occorrenze della variabile a nell'insieme delle xorclausole. Nel caso a non sia presente esce. Nel caso che, dopo il decremento, il numero di occorrenze di a sia nullo sia nelle orclause che nelle xorclause, cancella a. Per *Red*( ), *Simp*( ).
void	V.subsel(*const var* a)	decrementa il numero di occorrenze della variabile a nell'insieme delle select xor. Nel caso a non sia presente esce.
void	V.subass(*const var* a)	decrementa il numero di occorrenze di a nelle assegnazioni.

void	V.del(*var* a)	azzera le occorrenze della variabile a dall'insieme V.
void	V.flush( )	resetta le occorrenze di ogni variabile.
ostream & *ostream*& *os* $\ll$ V		stampa le variabili contenute con le loro occorrenze.

4. Implementation

I *vset* sono implementati come map <*var*, *occurrence*>. Le operazioni *addor*( ), *addxor*( ) prendono $O(1)$. Le operazioni *subor*( ), *subxor*( ), *del*( ) prendono $O(\log n)$ Le operazioni *flush*( ) prendono $O(n)$ dove n é il numero di elementi di V.

5.11 Variabili (var)

1. Definition

Una istanza v del tipo di dati *var* rappresenta una variabile booleana dotata di segno. È stata ideata per rappresentare la codificazione "human readable" nella riduzione in formule dell'algoritmo del DES. Per questo sono stati definiti i seguenti campi:

- nome variabile
- posizione occupata dal bit nel vettore
- numero ciclo
- numero blocco

#include < *../../include/var.h* >

2. Creation

var v; crea una instanza v of type *var* and la inizializza con il valore "T" (True).

var v(*bool costant*); crea una instanza v of type *var* and la inizializza con il valore costante contenuto in *costant*.

var v(*bool sign*, *char name*, *int bit_number*, *int cycle_number*, *int block_number*);
crea una instanza v of type *var* and la inizializzza con i valori passati come parametri.
Precondition:
sign = $1, 0$;
name = lettera dell'alfabeto;
bit_number = $[MIN\_BIT - MAX\_BIT]$;
cycle_number = $[MIN\_CYCLE - MAX\_CYCLE]$;
block_number = $[MIN\_BLOCK - MAX\_BLOCK]$;

var v(*var* w); crea una istanza v del tipo *var* e la inizializza con il valore di w (copy constructor).

3. Operations

bool	$v == w$	ritorna *true* se $var = w$, *false* altrimenti.
var &	$v = w$	assegna a v il valore di w
var	not(*var* w)	restituisce w cambiato di segno.
var	abs(*var* w)	restituisce il valore assoluto di w.

bool	$a < b$	restituisce 0 se a è maggiore od uguale a b effettuando la comparazione campo per campo nell'ordine: • sign • block • cycle • bit • name Altrimenti restituisce 1.
bool	v.sign( )	restituisce il segno di v nel senso della variabile.
char	v.name( )	restituisce il nome di v nel senso della variabile.
int	v.bit( )	restituisce il numero di bit di v nel senso della variabile.
int	v.cycle( )	restituisce il numero di ciclo di v nel senso della variabile.
int	v.block( )	restituisce il numero di blocco di v nel senso della variabile.
bool	v.is_costant( )	restituisce se la variabile rappresenta una costante (*true* o *false*).
bool	v.is_negative( )	restituisce *true* se la variabile ha segno negativo, *false* altrimenti.

Output Functions

ostream & *ostream*& *os* $\ll$ v	stampa su *os* i valori della variabile. Utilizza un formato analogo a quello dell'input: $[not](<name><bit>, <cycle>, <block>)$

4. Implementation

Le variabili vengono rappresentate da una struct contenente:

1. *char* nome della variabile
2. *bit_t* numero di posizione della variabile nell'array assume i valori da MIN_BIT a MAX_BIT
3. *cycle_t* numero di ciclo assume i valori da MIN_CYCLE a MAX_CYCLE
4. *block_t* numero di blocco assume i valori da MIN_BLOCK a MAX_BLOCK

Per ognuno di detti tipi (con l'esclusione del tipo rappresentante il nome) In questo modo diventa possibile usare codifiche differenti o anche aggiungerne di nuove.

Complessitá computazionale delle funzioni

Ogni funzione ha complessitá $O(1)$.

Indicazioni per la modifica degli intervalli di valori

Poichè *bit_t* e *block_t* assumono in generale solo valori positivi, la loro negazione è stata utilizzata per rappresentare:

- *bit_t* → segno negativo della variabile
- *block_t* → flag in uso nella classe derivata (*lvar*)

Onde non è possibile definire MIN_BIT o $MIN\_BLOCK < 1$. Non modificare le definizioni giá esistenti, ma aggiungerne di nuove, modificando invece il Makefile generale.

Indicazioni per la definizione dei tipi

Questa classe è stata pensata in seno alla criptoanalisi, ed in particolare ad una sua applicazione sul DES. Per questo comprende un campo carattere indicante il nome (la cui modifica richiede la revisione dell'intera classe). I campi rappresentanti in indice hanno invece il tipo ridefinibile, modificando semplicemente la *typedef* corrispondente. Occorre tuttavia osservare che la modifica deve essere fatta coerentemente con i valori limite impostati ($MAX\_\ldots$).

Definizioni per il DES

I corretti valori sono caricati in fase di compilazione, definendo nel Makefile:
$TAME = TAME\_ON\_DES$
I corretti tipi sono definiti nell' header file:
typedef short bit_t
typedef unsigned char cycle_t
typedef char block_t.

5.12 Xorqueues (xorque)

1. Creation

#include < *../../include/xorque.h* >

xorque XQ; crea una istanza di *XQ* e la inizializza all'insieme vuoto.

2. Operations

bool	*XQ*.bottom( )	restituisce se é sorta una contraddizione.
void	*XQ*.set_bottom( )	Setta la condizione di errore.
void	*XQ*.unset_bottom( )	resetta la condizione di errore.
void	*XQ*.Add(*var a*, *var b*)	inserisce la clausola x nello stack come ultimo elemento.
void	*XQ*.Push(*xorclause x*)	inserisce la clausola x nello stack come ultimo elemento.
xorclause	*XQ*.Top( )	ritorna il riferimento al primo elemento.
void	*XQ*.Pop( )	cancella il primo elemento.
void	*XQ*.AddAllArbitraryAssignment(*lassign VB*)	Inserisce tutte le assegnazioni di *VB* in *XQ*.
void	*XQ*.AddAllAssignment(*assignmap Ass*)	Inserisce tutte le assegnazioni di *Ass* in *XQ*.

Output Friend Function

ostream & *ostream*& *os* $\ll$ *XQ*	Stampa su *os* le clausole presenti nella XorQue *XQ*, separate da un accapo. Utilizza *operator* $\ll$ (*ostream* & *os*, *xorclause x*) per stampare ogni elemento.

3. Implementation

Le XorQue sono implementate tramite deque, instanziata al tipo *xorclause*. Tutte le funzioni hanno complessitá $O(1)$, con l'eccezione di $\ll$ che é $O(n)$.

5.13 Structural Functions (Algorithmic Guidelines for Formulas)

bool init(*formula*& *F*, *char* ∗ *infile*)

Inizializza la formula —— e l'insieme delle variabili v che vi sono racchiuse, analizzando il file di Input descritto dalla stringa *infile*.

bool evaluate(*formula*& *G*, *formula F*, *var v*)

Assegna a G il valore della formula ——, imponendo $v = true$. Dove v é inteso con segno. v é aggiunta tra le variabili assegnate in $G.Ass$.

bool calculate(*formula*& *F*) Esegue su —— tutti i calcoli semplificativi, che non implicano assunzione arbitraria di valori per alcuna variabile. Ritorna *true* se non state generate clausole vuote nella computazione; *false* altrimenti. Setta il flag di cambiamento di F in caso di semplificazione.

bool lookahead(*formula*& *F*, *lassign*& *VB*)

Esegue il calcolo del lookahead a 2 livelli sulla formula ——:

- controllo che non ci siano variabili che rendono falsa dopo un solo passo la formula
- inserimento delle clausole comuni ai 2 casi di assegnazione della variabile nel caso non si renda falsa in nessuno dei 2 casi opposti la formula
- controllo che non venga generato un bottom() per entrambi i valori assunti

Restituisce $false$ nel caso si generi bottom() per entrambe le assunzioni, $true$ altrimenti. Setta il flag di cambiamento di F.

bool compute(*formula*& *F*, *lassign*& *VB*)

Funzione ad alto livello che esegue su —— le operazioni che non necessitano di split():

- calcolo sulla formula
- lookahead a 2 livelli

. Restituisce *true* se il calcolo ha avuto esito positivo, *false* se si sono generati degli errori.

bool .computeone(*formula*& *F*, *lassign*& *VB*)

Come la funzione precedente, in piú aggiunge le assegnazioni che provengono dalla normale computazione (e non solo dalla *lookahead*( ) in *VB*. Va usata all'inizio, quando si semplifica la formula di riferimento.

bool restart(*formula*& *F*, *formula G*, *lassign*& *VB*)

Calcolo con split() sulla formula G, tenendo come riferimento la formula F. Finché possibile esegue la riduzione della formula G, senza effettuare branch. Ove necessario esegue il branch attraverso la *split*( ). In caso di sopraggiunta contraddizione, riassegna G con la formula di riferimento F riassegnando le variabili di branch, avendone preventivamente efettuato un riordine.

5.14 Rules (Calculating Rules for Formulas)

1. Definition

Qui sono raccolte le funzioni utilizzate per semplificare le formule. Tutte le funzioni effettuano $side - effect$.

#include < *../../include/rules.h* >

2. Operations

bool RedOr(*orclause*& *c*) Riduce la clausola c, eliminando le ridondanze, effettuando "side-effect". Esegue le seguenti azioni:

- elimina le costanti *TRUE*: $T \vee C : C \neq \emptyset \rightarrow T$
- elimina le costanti *FALSE*: $F \vee C \rightarrow C$
- elimina le ridondanze di segno opposto: $L \vee \neg L \vee C \rightarrow T$

Restituisce se la clausola é stata ridotta. Osserviamo esplicitamente che le ridondanze (solo di segno concorde) sono eliminate automaticamente poichè le clausole sono implementate come *set*.

bool RedXor(*xorclause*& *x*) Riduce la clausola x, eliminando le ridondanze, effettuando "side-effect". Esegue le seguenti azioni:

- elimina le costanti *TRUE*: $T \oplus L \oplus C \rightarrow \neg L \oplus C$
- elimina le costanti *FALSE*: $F \oplus C \rightarrow C$
- elimina le ridondanze di segno opposto: $L \oplus \neg L \oplus C \rightarrow T \oplus C$

Restituisce *true* se la clausola é stata ridotta. Osserviamo esplicitamente che le ridondanze (anche di segno discorde) sono eliminate automaticamente poichè le clausole *xor* sono implementate come *set*.
È tuttavia necessario che si usi la funzione *add*( ) ogni qualvolta si voglia inserire un nuovo elemento.

int SimpOr(*formula*& *F*, *const var A*, *const var B*, *xorque*& *XQ*)

semplifica l'insieme di clausole $F.C$ in base alla assegnazione: $A := B$, sostituendo ogni occorrenza di A con B. Restituisce il numero delle clausole semplificate.

int SimpXor(*formula*& *F*, *const var A*, *const var B*, *xorque*& *XQ*)

semplifica l'insiemi di clausole $F.X$ in base alla assegnazione: $A := B$, sostituendo ogni occorrenza di A con B. Restituisce il numero delle clausole semplificate.

int SimpQue(*xorque*& XQ, *const var* A, *const var* B)

semplifica il buffer delle assegnazioni, eseguendo anche la riduzione.

bool Simp(*formula*& F, *xorque*& XQ)

semplifica XQ, $F.C$, $F.X$, $F.S$, in base alle assegnazioni contenute in XQ. Restituisce *true* se non ci sono state contraddizioni, *false* altrimenti.

int SimpSel(*formula*& F, *const var* A, *const var* B)

semplifica l'insieme di clausole $F.S$ in base alla assegnazione: $A := B$, sostituendo ogni occorrenza di A con B. Restituisce il numero di clausole semplificate.

int SimpSel(*formula*& F, *const var* A, *const var* B, *selectxor*::*iterator* IT)

come la funzione precedente, senza semplificare la selezione puntata da IT.

bool UnitOr(*orclause*& x, *xorque*& XQ)

Ricerca all'interno di $F.C$ ogni clausola contenente un solo letterale ($L = \pm A$).

- se $L = False$ setta il flag di bottom() su F.C e restituisce *false*
- se $A \neq T$ inserisce $A := \pm True$ nello stack delle assegnazioni $F.Ass$, cancella la clausola da $F.C$, restituisce *true*
- se $L = T$ cancella la clausola da $F.C$ e restituisce *true*

Restituisce il numero di $unit-clauses$ trovate. Nel caso sia trovato un bottom restituisce 0.

bool UnitXor(*xorclause*& x, *xorque*& XQ)

Restituisce se x é una clausola contenente un solo letterale L

- se $L = False$ setta il flag di bottom() su XQ e restituisce *false*
- se $A \neq T$ inserisce $A := \pm True$ in XQ e restituisce *true*
- se $L = T$ restituisce *true*

int Or3Xor(*formula*& F) Sposta le clausole di 3 letterali da $F.C$ a $F.X$. Ovvero: $\forall c \in F.C : c = a \vee b \vee c$ if $\exists c1, c2, c3 \in C$ con $c1 = a \vee \neg b \vee \neg c,\ c2 = \neg a \vee b \vee \neg c,\ c3 = \neg a \vee \neg b \vee c$ cancella $c, c1, c2, c3$ da $F.C$ ed inserisce $x = a \oplus b \oplus c$ in $F.X$ Restituisce il numero di nuove clausole XOR trovate.

var Select(*formula*& *F*)

Seleziona una clausola x di $F.X$ ed un letterale A in essa presente per prenderla come definizione di A. Per eliminare cicli nella riduzione di gauss $\forall B \in x, x$ non ha una definizione in $F.S$ e $A \neq \pm T$. Restituisce A se la ricerca ha avuto successo, *False* altrimenti. Va usato nelle clausole con almeno 3 letterali.

bool Unselect(*formula*& *F*, *var A*)

deseleziona da $F.S$ la definizione di A, se esiste. Ritorna se ha avuto successo.

int Gauss(*formula*& *F*, *var A*, *xorque*& *XQ*)

Riduce le clausole dell'insieme $F.X$, $F.S$ sulla base della *var A* e della sua definizione x in $F.S$. Calcola il risolvente di Gauss rispetto a x, riducendolo e controllando se sia divenuto una clausola con un unico letterale. Cancella la premessa di x. Inserisce il risolvente in x. Restituisce il numero di clausole *XOR* ridotte.

int EqvOr(*formula*& *F*, *xorque*& *XQ*)

Sposta le clausole di 2 letterali da $F.C$ ad XQ. Qualora vengano trovate 2 clausole del tipo:
$C1 = \{A, B\}$ e $C2 = \{\neg A, \neg B\}$.
Cancella le variabili A e B dall'insieme $F.V$ delle proposizioni di F. Restituisce il numero di nuove assegnazioni trovate.

int EqvXor(*formula*& *F*, *xorque*& *XQ*)

Sposta le clausole di 2 letterali da $F.X$ ad XQ. Per ogni clausola $X = \{A, B\}$ trovata cancella le variabili A e B dall'insieme $F.V$ delle proposizioni di F. Restituisce il numero di nuove assegnazioni trovate.

var Split(*formula F*, *lassign*& *VB*)

Funzione che impone una condizione su una della variabili della formula F, effettuando la scelta in base alla function object euristica *Priority*. Restituisce:

- la variabile assegnata, se il calcolo ha avuto esito positivo
- la costante *FALSE*, se l'insieme delle clausole *OR* é vuoto

int RedUnit(*formula*& *F*, *xorque*& *XQ*)

Esegue le riduzioni necessarie sulle clausole di F e la ricerca di eventuali unit clause, che vengono inserite in XQ.

5.15 Priority Functions (Branching Strategies for Heuristics)

float MaximumSizeofAssSet(*formula F*)

Calcola la prioritá in base alla dimensione dell'insieme delle assegnazioni della formula F. Split che producono insiemi $F.Ass$ piú grandi portano a prioritá piú elevate. Non molto accurata: nella maggior parte dei casi corrisponde alla "First Open Proposition".

float MinimumMediumOrClauseLenght(*formula F*)

Calcola la prioritá in base alla dimensione media delle clausole *OR*. Split che producono dimensioni medie maggiori portano a pioritá minori.

float MaximalMediumOccorrenceinXor(*formula F*)

Calcola la prioritá in base al numero medio di occorrenze delle proposizioni nelle clausole *XOR*. Split che producono occorrenze medie maggiori portano a prioritá piú elevate.

float MinimumXorClausesMediumDimension(*formula F*)

Calcola la prioritá in base alla dimensione media delle clausole *XOR*. Split che producono dimensioni medie maggiori portano a pioritá minori.

float MaximalAddedConstraints(*formula F*)

Calcola la prioritá in base ad un calcolo approssimativo sul numero di *costraints* che vigono nella formula. Il calcolo tiene conto:

- dei constraint dati da ogni tipo di clausola in funzione della lunghezza
- della lunghezza media delle clausole di ogni insieme

Non vengono tracciati i *costraints* dati dalle interazioni fra letterali presenti in piú di un insieme.

Capitolo 6

Conclusioni

Lo scopo della presente tesi consiste nel dimostrare la convenienza nel trattamento esplicito della Logica Affine, nelle tipologie di SAT Problems in cui occorra, in modo più o meno preponderante, non già procedendo in modo separato (e.g. EqSatz, cfr [Li, 2000]), bensì effettuando più calcoli paralleli e condividendone i risultati.

Il solver realizzato si basa sull'algoritmo descritto in [Baumgartner and Massacci, 2001], dove viene proposto un metodo di calcolo innovativo, per la logica proposizionale, derivato dal DPLL, basato sulla riduzione delle clausole XOR, tramite regole di inferenza, tra le quali un adattamento della risoluzione di Gauss (già utilizzata in EqSatz, cfr [Li, 2000], e Warners e van Maaren, cfr. [Warners and van Maaren, 1999]). Questo ha permesso anche la introduzione, e questo è un concetto nuovo, del supporto per la definizione funzionale di modelli, per mezzo della quale le variabili possono essere espresse in funzione di altre, per mezzo di equivalenze e, quindi implicitamente, di XOR, senza ricercare spasmodicamente una assegnazione di valori di verità, che limita la generalità della soluzione ed ha un costo rilevante sulla computazione.

Dopo la realizzazione ed il test del solver si è potuto procedere a dei test comparativi, in cui si è evidenziata la validità di questo approccio, soprattutto con riferimento alle istanze abbastanza complesse, come i DIMACS Parity32.

Il trattamento esplicito della logica affine ha permesso la computazione di alcuni problemi in minor tempo, ma soprattutto la drastica riduzione del numero di nodi visitati alla ricerca della soluzione.

Inoltre, in genere, grazie alla descrizione funzionale dei modelli, le soluzioni formite sono più generiche e, proprio per questo motivo, meno numerose, qualora si vogliano tutte le soluzioni.

6.1 Sviluppi Futuri

Ho cercato di sviluppare un solver che fosse il più possibile fedele all'algoritmo proposto dai Professori Massacci e Baumgertner, inserendo qualche ottimizzazione in alcuni punti critici. Tuttavia lo scopo essenziale era realizzare un solver funzionante sull'algoritmo base, onde poterne testare le capacità.

Vi sono, quindi, numerosi spazi di miglioramento del solver proposto, fondamentalmente in tre direzioni:

- il bilanciamento di alcune funzionalità interne già presenti nella procedura
- l'introduzione di altri elementi nella struttura di calcolo

- la ottimizzazione, da un punto di vista funzionale, delle strutture dati

purtroppo lo studio di ciascuna di esse non è semplice per via della complessità delle strutture necessarie all'algoritmo, ognuna con le sue proprietà logico-matematiche, rispetto agli altri solver .

Bilanciamento di Funzionalità Interne

Introdurre un meccanismo per limitare l'uso della lookahead in istanze più semplici in cui non occorre richiamare tale funzione così spesso. Parallelamente migliorare le euristiche effettuando uno studio sistematico dei parametri da considerare su tutti i tipi di insiemi di clausole previsti.

Introduzione di altri Elementi

Negli ultimi anni sono state introdotte tecniche altamente inovative come la BackJumping ed il Learning (e.g. [Giunchiglia et al, 2000], [Zhang et al, 2002], [Bayardo and Schrag, 1997]), che se ben usate possono ridurre di parecchio i tempi di computazione (si veda le performnace di Zchaff su http://www.satex.org).

Ottimizzazione delle Strutture Dati

Come si è potuto osservare in fase sperimentale, il programma non richiede eccessivo spazio di memoria, onde è possibile costruire una struttura dei dati "tramata", imitando quanto fatto in Simo(cfr. [Giunchiglia et al, 2000]), ma anche in molti altri solver, in modo da poter risalire facilmente (in tempo logaritmico) a tutte le clausole che contengono un certo letterale, a tutto vantaggio della esecuzione di funzioni fondamentali come la semplificazione o la riduzione di Gauss stessa.

Bibliografia

[Baumgartner and Massacci, 2001] Peter Baumgartner, Fabio Massacci. *The Taming of the (X)OR*, 2001

[Massacci and Marraro, 2000] Fabio Massacci, Laura Marraro. *Logical Cryptoanalysis as a SAT Problem: the Encoding of the Data Encryption Standard*, JAR: Journal of Automated Reasoning, 2000

[FIPS PUB 46-2, 1997] AA. VV. *Data Encryption Standard (DES)*. Federal Information Processing Standard Publications FIPS PUB 46-2, National (U.S.) Bureau of Standards, december 1997

[Matsui, 1993] Mitsuru Matsui. *Linear Cryptanalysis Methods for DES Ciphers*, Advanced in Cryptography (EUROCRYPT93), Lectire Notes in Computer Science, 1993

[Matsui, 1994] Mitsuru Matsui. *The First Experimental Cryptanalysis of the Data Encryption Standard*, Proc. Of Advanced Cryptography (CRYPTO-94), Lecture Notes in Computer Science, Springer & Verlag, 1994

[Biham, 1994] Eli Biham. *On Matsui's Linear Cryptanalysis*, Advanced in Cryptography (EUROCRYPT93), Lectire Notes in Computer Science, 1994

[Chabaud and Vaudenay, 1994] Florent Chabaud, Serge Vaudenay. *Links Between Differential and Linear Cryptanalysis*, Advanced in Cryptography (EUROCRYPT93), Lectire Notes in Computer Science, 1994

[Matsui, 1994b] Mitsuru Matsui. *On Correlation Between the Order of S-boxes and the Strenght of DES*, Advanced in Cryptography (EUROCRYPT93), Lectire Notes in Computer Science, 1994

[McFarland, 1997] Matt McFarland. *Infomation Security Through Encryption*, 1997

[Anderson, 1995] Ross J. Anderson *Computer Science Tripos-Security*, Stream Ciphers, Block Ciphers, Lectures

[Diffie, 1996] Whitfield Diffie. *Public Key Cryptography*, Sun Microsystems, 1996

[Giunchiglia et al, 2000] E.Giunchiglia, M.Maratea, A. Tacchella, D.Zambonin. *Evaluating search heuristics and optimization techniques in propositional satisfiability*, DIST, Università di Genova 2000

[Copty et al, 2001] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, Moshe Y. Vardi, *Benefits of Bounded Model Checking at an Industrial Setting*, Logic and Validation Technology, Intel Corporation, Haifa, Israel

[Moskewicz et al, 2001] Matthew. H. Moskevicz, Conor F. Madigan, Ying Zao, Lintao Zhang, Sharad Malik. *Chaff: Engineering an Efficient SAT Solver*, 2001

[Zhang et al, 2002] Lintao Zhang, Conor F. Madigan, Matthew H. Moskevicz, Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, 2002

[Harrison, 1996] Johm Harrison. Stalmarck's algorithm as a HOL derived rule, TPHOL-96, LNCS, 1996

[Selman et al, 1997] Bart Selman, Henry Kautz, Davis McAllester. *Ten Challenges in Propositional Reasoning and Search*, Proc. IJCAI, 1997

[Kautz and Selman, 1996] Henry Kautz, Bart Selman. Pushing the Envelope: *Planning, Propositional Logic and Stocastic Search*, In Proc. AAAI-96

[Johnson, 1996] Davis S. Johnson. *A Theoretician's Guide to the Experimental Analysis of Algorithms*, 1996

[Groote and Warn, 2000] J. Groote and J. Warners. The Propositional Farmula Checker Heer Hugo, JAR, volume 24, 2000

[Franco and Paull, 1983] John Franco and Marvin Paull. Probabilistic Analysis os the Davis Putnam procedure for solving the Satisfiability, Discrete Applied Mathematics

[Hogg et al, 1996] Tad Hogg, Bernardo A. Huberman, Colin P. Wiliams. Phase Transitionand the Search Problem, Artificial Intelligence 81, 1996

[Sito DIMACS, 2002] AA.VV. http://dimacs.rutgers.edu

[DIMACS Format] AA.VV. Satisfiability Suggested Format, 8 maggio 1993

[Buro and Burning, 1992] M. Buro, H. Kleine Burning. *Report on a SAT Competition*, 1992

[SATLib site, 2002] AA.VV. http://www.satlib.org

[Edwards, 1996] Douglas D. Edwards. *Research Summary*, 4 december 1996

[Dechter and Rish, 1994] Rins Dechter, Irina Rish. *Directional Resolution: the Davis Putnam Procedure, Revisited*, 1994

[Zhang and Stickel, 1995] Hantao Zhang, Mark E. Stickel. *Implementing the Davis-Putnam Algorithm by Tries*, 1995

[Zhang and Stickel, 2000] Hantao Zhang, Mark E. Stickel. *Implementing the Davis-Putnam Method, SAT 2000*, Highlights of Satisfiability Research in the Year 2000. IOS Press, 2000

[Zhang, 1997] Hantao Zhang. *SATO: an Efficient Propositional Prover*, Proc. CADE, 1997

[Freeman, 1995] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*, PhD Thesis, University of Pennsylvania, 1995

[Selman et al, 1992] Bart Selman, Levensque, David G. Mitchell. *A New Method for Solving Hard Satisfiability Problems*, 1992

[LI and Anbulagan, 1997] Chu Min Li, Anbulagan. *Heuristic Based on Unit Propagation for Satisfiability Problems*, Proc. IJCAI, 1997

[Li, 1999] Chu Min Li. *Equivalence Reasoning to Solve a Class Of Hard SAT Problems*, 1999

[LI, 1999] Chu Min Li, *A Constraint Based Approach to Narrow Search Trees for Satisfiability*, IPL, 1999

[Li, 2000] Chu Min Li. *Integrating equivalence reasoning into Davis-Putnam procedure*, Proc. AAAI, 2000

[Warners and van Maaren, 1999] J. Warners, H. van Maaren. *A two Phase Algorithm for Solving a Class of Hard Satisfiability Problems*, Operations Research Letters, 1999

[Jeroslow and Wang] Robert G. Jeroslow, Jinchang Wang. *Solving Propositional Satisfiability Problems*

[Hooker and Vinary, 1994] J.N. Hooker, V, Vinay. Branching Rules for Satisfiability, GSIA Working Paper 1994-09, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania, Aprile 1994

[Schiermeyer, 1996] Ingo Schiermeyer. *Pure Literal Look Ahead: an O(1,497^n) 3-Satisfiability Algorithm*, Siena Workshop on Satisfiability, 1996

[Park and Gelder, 1995] Tai Joon Park, Allen Van Gelder. *Partitioning Methods for Satisfiability Testing on Large Formulas*, 1995

[Cook and Mitchell, 1997] Stephen A. Cook, David G. Mitchell. *Finding Hard Instancies of the Satisfiability Problem: A Survey*, DIMACS Series vol. 35, 1997

[Bayardo and Schrag, 1997] Roberto J. Bayardo Jr, Robert Schrag. *Using CSP Look-Back Techniques Exceptionally SAT Instancies*, Proc. AAAI 1997

[Purdom et al, 1997] Jun Gu, Paul W. Purdom, John Franco, Benjamin W. Wah. *Algorithm for the Satisfiability (SAT) Problem: A Survey*, DIMACS Series vol. 35, 1997

[Rumbaugh and Blama, 1995] James Rumbaugh, Michael Blama. *Object-Oriented Modeling and Design*, Prentice Hall 1995

[Melhorn et al, 1999] Kurt Mehlhorn, Stefan Naher, MIchael Seel, Christian Uhrig. *The LEDA User Manual, version 3.8*

[Meyers, 2001] Scott Meyers. *The Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library,*Addison-Wesley, november 2001

[SGI, 2002] The SGI STL web site. *http://www.sgi.com/tech/stl*

[Boost] The Boost web site. *http://www.boost.org*

[Brokken and Kubat, 1997] Frank B. Brokken, Karel Kubat. *C++ Annotations, version 4.30*, University of Groningen 1997

[Eckel et al, 2000] Bruce Eckel. *Thinking in C++, 2nd edition*, january 2000

[Donnelly and Stallman, 1995] Charles Donnelly, Richard Stallman. *Bison: The YACC-compatible Parser Generator,* november 1995

[Coetmeur, 1993] Alan Coetmeur. *Bison++,*march 1993

[Paxson, 1995] Vern Paxson. *Flex: A fast Scanner Generator,* march 1995

[Knuth, 1983] Donald E. Knuth. *The TEXbook*, Addison-Wesley, june 1983

[Oetiker et al, 2000] Tobias Oetiker, Hubert Partl, Irene Hyna, Elisabeth Schlegl. *The (not so) short Introduction to LATEX 2ε,* august 2000

Appendice A

Organizzazione dei File Sorgenti

./include

Directory contenente le definizioni di tutti i tipi di dati (classi) con i relativi metodi, i principali algoritmi, quali le regole, i *function-object* per la euristica, le funzioni strutturali.

./src

tame.C

File principale contiene:

- funzione `main()` con:
  - argc, argv[] per identificazione file di Input con le clausole
  - getopt() per modificare il metodo computazionale da riga di comando
- definizione formula reference (di riferimento) e tide (di calcolo)
- chiamata funzione `init()` (init.h) per inizializzare le formule
- chiamata della funzione `restart()` per eseguire il calcolo

init.C

File per la inizializzazione delle formule, contiene:

- collegamento con il parser

Deve includere:

- ../include/init.h
- ../parser/...

restart.C

File per la esecuzione dei calcoli. Definisce la funzione:

- *restart()* che chiama le funzioni:
  - *split()* per la scelta della variabile sulla quale eseguire l'assegnazione arbitraria
  - `compute()` per la computazione del nodo corrente

compute.C

Definizione delle funzioni:

- `compute()` che effettua la computazione sul nodo corrente, chiamando a sua volta:
  - `calculate()` che effettua i calcoli relativi alla applicazione delle regole che non prevedono assegnazioni arbitrarie
  - `lookahead()` che effetttua le semplificazioni, tramite assegnazione temporanea di valori a determinate varibili
- `computeone()` come la compute(), ma inizializza il vettore di stato delle variabili assegnate

lookahead.C

definizione delle funzioni:

- lookahead() per il calcolo di previsione con assegnazione ad un passo

formula.C

definizione della classe-contenitore fondamentale. Tutti gli oggetti delle classi in essa istanziate sono definiti integralmente in appositi header file nella directory `../include`.

./src/rules

Directory contenente tutti le funzioni che realizzano le regole

./parser/

Directory contenente i file di definizione del parser da usare in principio:

- parser.y definizione delle regole gramaticali e delle azioni semantiche, per bison++
- lexer.l definizione delle regole sintattiche dello scanner, per flex++
- populate.C definizione delle funzioni per popolare la formula iniziale.

./bin

Directory contenente:

- tame l'eseguibile finale, dopo la compilazione
- MANAUL.pdf il testo di tesi
- cnf2mas.pl script per convertire i problemi dal formato DIMACS al formato da me utilizzato
- runtame.sh script per lanciare gli esperimenti

./obj

Directory contenente i file oggetto durante la compilazione

./lib

Directory contenente i file di libreria durante la compilazione

Appendice B

LEDA tools for Manual Production

In questa appendice si introducono i concetti fondamentali, ma soprattutto i comandi da inserire nei commenti del codice C++ presente negli header file, per fare in modo di generare automaticamente la documentazione delle proprie classi in LaTeX.[1]Tutto il meccanismo si basa sul processamento effettuato in *perl* dal file: *Manual/cmd/ext.pl*. Tutte, o quasi, le informazioni qui riportate le ho desunte analizzando i commenti negli header file della libreria e questo programma, che puó essere invocato nei seguenti modi:

- tramite *lextract* producendo dei file LaTeXda includere successivamente in un documento piú ampio (come questo)
- tramite *Lman* producendo un file LaTeXtemporaneo, compilato *on-the-fly* e visualizzato con *xdvi* (dunque il comando va dato da una finestra di X)
- tramite *Fman* producendo in stdout un file Ascii

Per una trattazione completa sul funzionamento si veda []. Vediamo ora una guida alle direttive maggiormente usate.
Si osserva esplicitamente che per un corretto utilizzo ogni oggetto cui si voglia fornire la documentazione di accesso, deve essere definito in file a sé stante, magari avente lo stesso nome della classe da esso contenuta.

B.1 Concetti Generali

I comandi per la generazione devono essere:

- inclusi all'interno dei commenti "multiriga" (quelli con gli asterischi per intenderci)
- essere raccchiusi, a loro volta, da una coppia di parentesi graffe
- cominciare con un "master command"

B.1.1 Comandi

I comandi sono macro LaTeXdefinite nel file: *Manual/tex/MANUAL.mac.*
Li divido per comodo in due insiemi:

- "master command" che creano un nuovo "blocco"
- "auxiliar command" che forniscono comode utilitá

[1]Esistono dei prodotti Open Source anche migliori di questo, in quanto permettono di esportare anche in altri formati (UML, manpages etc.). Uno dei piú famosi è doxygen.

B.1.2 master command

Si dividono nei tipi:

- comando di intestazione: da posizionare in testa a tutti i comandi seguenti. Comunica la richiesta di generare una sezione riguardante il codice contenuto nel file
- comando di "mansection": crea una specie di sottosezione all'interno del documento, per separare i metodi in costruttori (distruttori) e selettori e suddividere le informazioni aggiuntive sulla classe
- comando di "function": crea una sorta di item contenente la descrizione del costruttore (eventualmente anche le precondizioni) a testo libero. Va collocato la riga subito successiva a quella contenente la dichiarazione o la definizione della funzione.
- "\Mvar", "\Mname": sostituiscono, rispettivamente, il nome della variabile utilizzata per indicare l'oggetto tipo della classe definenda e il nome stesso della classe, stabiliti nella intestazione

B.1.3 auxiliar command

Ne ho usati pochi. Molto comodo il comando "— ... —", che tratta tutti i caratteri che trova al suo interno verbatim, visualizzandoli poi in corsivo. Anche utile "\precond", all'interno della descrizione di un funzione, per elencare le precondizioni ostanti al fruimento delle stessa.

B.1.4 LaTeXcommand

Ovviamente rimane possibile usare all'interno dei vari blocchi i comandi standard, facendo bene attenzione, soprattutto con le direttive riguardanti la formattazione, specie se in prossimitá della fine del blocco, di inserire una riga vuota.

B.2 Master Command

Vediamo i principali comandi, di utilitá generale.

Osservo che nel caso dei comandi "function" vale anche un secondo nome oltre quello che indico con una 'l' in piú alla fine (i.e. "\Marrop" ed anche "\Marropl"); il funzionamento è pero\'o del tutto identico.

Nei titoli dei sotto paragrafi seguenti i nomi dei comandi saranno indicati senza il backslash anteposto.

B.2.1 Intestazione (Manpage)

Ogni dichiarazione di oggetti per la quale si voglia generare la documentazione, deve essere preceduta da questo comando che:

- imposta il Titolo
- imposta "\Mname"
- imposta "\Mvar"

- imposta il nome del tipo *template*, se presente, su cui viene costruita la classe
- apre una sezione con titolo: *¡Titolo¿ (\Mname)*

tutti questi argomenti sono passati tra parentesi graffe. La sua forma generale:

`/*{\Manpage {<Titolo>} {<nome_template>} {Mname}{Mvar} }*/`

Possono essere omessi degli argomenti, partendo dall'ultimo.

B.2.2 Definizione (Mdefinition)

Contiene una descrizione generale e succinta delle caratteristiche e funzionalitá della classe.

Va inserita nella linea immediatamente inferiore a quella contenente la parentesi grafa di inizio dichiarazione dell'oggetto. Contrariamente ad altri mansection, ospita testo libero:

```
/*{\Mdefinition
<testo_libero>
}*/
```

B.2.3 Creazione (Mcreation)

Indica l'inizio della dichiarazione dei costruttori (o distruttori). Solitamente si colloca dopo la dichiarazione dei campi privati, subito dopo la riga contenente la parola chiave *public:*.

Puó contenere la ridefinizione del parametro *Mvar*. La sua forma generale è:

`/*{\Mcreation 3}*/`

Costruttori (Mcreate)

Crea un blocco per le definizione dei costruttori e distruttori.

B.2.4 Operazioni (Moperation)

Indica l'inizio delle dichiarazioni dei selettori (funzioni membro, friend, esterne, operatori). Va posto subito dopo la fine della dichiarazione dei costruttori. Contiene due parametri:

`/*{\Moperation 1.8 5}*/`

Metodo Membro (Mop)

Si riferisce ai metodi membro (quelli che si invocano tramite un oggetto) che non siano degli operatori (*operator...*).

Metodo Membro Operatore (Mbinop)

Si riferisce ai metodi membro operatori, eccezzion fatta per l'operatore *[]*, ovvero tutti quelle funzioni interne per le quali è prevista la notazione infissa.

Metodo Membro Operatore [] (Marrop)

Si riferisce ai metodi membro che eseguono l'overload di `[]`.

Metodo Esterno Operatore (Mbinopfunc)

Si riferisce ai metodi esterni (anche *friend*) che sovraccarichino un operatore, ovvero tutte le funzioni esterne per le quali è prevista la notazione infissa.

Metodo Esterno (Mfunc)

Si riferisce ai metodi esterni (anche *friend*).

B.2.5 Dettagli Implementativi (Mimplemenation)

Contiene una descrizione dettagliata della classe con particolare riferimento a:

- classi su cui si basa
- complessitá tipiche nella esecuzione delle funzioni
- motivazione delle scelte realizzative

Anch'essa contiene testo libero:

/{\Mimplementation*

<testo_libero>

}/*

B.2.6 Testo Libero (Mtext)

Permette di inserire del testo libero (non collegato al codice). Non interrompe la "mansection" entro cui viene inserito. La sua forma generale:

/*{\Mtext

¡testo_libero¿

}*/

È possibile inserire qualsiasi tipo di comando LaTeX.