Università di Pisa
Facoltà Di Scienze Matematiche, Fisiche E Naturali
Corso di Laurea in Informatica

Analisi di euristiche per il calcolo del grado di autosimmetria di funzioni booleane non completamente specificate

Relatore:
Prof.ssa Anna Bernasconi

Candidato:
Alessio Bardelli

Anno Accademico 2019/2020

Indice

Introduzione

I circuiti digitali rappresentano l'implementazione fisica di una funzione booleana. La progettazione di tali circuiti è un processo tramite il quale si passa da una funzione booleana alla descrizione in termini di porte logiche del circuito che la calcola. Una fase estremamente importante di questo processo, nota come *minimizzazione*, si occupa di calcolare forme algebriche ridotte, possibilmente minime, che portino alla realizzazione di circuiti compatti e con alte performance. Purtroppo come spesso accade per molti problemi di ottimizzazione, la *minimizzazione* ricade in una ben nota classe di problemi detta $\mathcal{NP}$-Ardui, per cui non sono tuttora noti algoritmi risolutivi con un tempo di esecuzione polinomiale nella dimensione degli input, richiedendo così l'uso di algoritmi euristici e di approssimazioni per la sua risoluzione.

Fortunatamente, una funzione booleana che rappresenta problemi della vita reale mostra spesso una qualche forma di regolarità esprimibile in termini logico-matematici. Non sempre è facile scoprire il tipo di regolarità di una funzione ma, una volta scoperto quest'ultimo, è possibile utilizzarlo durante il processo di minimizzazione per ottenere un'implementazione migliore del circuito.

In letteratura sono stati proposti numerosi metodi che sfruttano diversi tipi di regolarità per il procedimento di *minimizzazione*. In particolar modo in [1] viene introdotta la notazione di *autosimmetria*, una caratteristica che, se presente, permette di ottenere una forma ridotta della funzione trattata che dipende da meno variabili, rendendo il procedimento di minimizzazione più semplice da un punto di vista computazionale. Viene inoltre mostrato che l'autosimmetria è una caratteristica che accomuna numerose funzioni di interesse pratico, infatti circa il 61% degli output delle funzioni presenti nella suite di benchmark *Espresso* [16] ricadono in questa classe. Ciò suggerisce che una trattazione dettagliata di questa caratteristica può portare a grandi vantaggi.

In [5] viene fornito un insieme di algoritmi per testare l'autosimmetria di una funzione e calcolarne la sua forma ridotta in modo molto efficiente. Tuttavia non viene presa in considerazione la classe delle funzioni booleane note come non completamente specificate, ovvero quelle funzioni che possono assumere un valore detto *condizione di indifferenza* o *don't care*, ad indicare che per un dato input la funzione può assumere indifferentemente valore zero oppure uno. In [1] viene proposta una prima definizione di autosimmetria, che mira a comprendere anche questo tipo di funzioni.

L'obiettivo di questa tesi è quello di proporre e valutare nuove euristiche per il calcolo del grado di autosimmetria di funzioni booleane non completamente specificate.

Capitolo 1

Premesse

In questo capitolo si descrivono le funzioni booleane e, in particolare, si introducono formalmente le funzioni booleane non completamente specificate dando una caratterizzazione più precisa del dominio.

1.1 Funzioni booleane

Le funzioni booleane vengono solitamente definite sullo spazio booleano $\{0,1\}^n$, descritto da n variabili $x_1, x_2, ..., x_n$, dove ogni punto rappresenta un vettore binario di n bit. Una funzione booleana $f : \{0,1\}^n \to \{0,1\}^m$ mappa quindi, ciascuna delle 2^n possibili configurazioni delle variabili in ingresso in una delle 2^m configurazioni di uscita. Esistono diversi modi di specificare come viene effettuata questa corrispondenza. Un modo esplicito di farlo è quello di utilizzare le tabelle di verità, dove si vanno ad enumerare tutte le configurazioni di ingresso, e per ciascuna di esse si specifica la configurazione di uscita. Ad esempio:

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	0
1	0	1
1	1	0

Tabella 1.1 – Tabella di verità di $f : \{0,1\}^2 \to \{0,1\}$.

Nel resto dell'elaborato, per semplicità, verranno considerate solo funzioni a singolo output. È possibile fare questa scelta, poiché ogni funzione ad m output, può essere considerata come m funzioni differenti ad output singolo.

Alternativamente è possibile fissare un insieme di operatori logici di base, come ad esempio { AND($\wedge$), OR($\vee$), NOT($\neg$) }. In questo modo si può fornire la specifica di una funzione booleana, in modo implicito, tramite un'espressione booleana in cui le variabili sono connesse tra loro utilizzando gli operatori logici considerati. Formalmente le espressioni booleane sono generate dalla seguente grammatica:

$$t ::= x \mid 0 \mid 1 \mid \neg x \mid t \vee t \mid t \wedge t$$

dove x varia su un'insieme di variabili booleane. La sintassi concreta dovrebbe includere anche le parentesi per risolvere l'ambiguità di un'espressione. E' interessante notare che esiste una sola tabella di verità per ogni funzione, mentre vi possono essere diverse espressioni booleane che rappresentano la stessa funzione. Difatti due espressioni booleane si dice che sono uguali se e solo se esprimono la stessa funzione booleana.

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

(a)

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

(b)

x_1	$\neg x_1$
0	1
1	0

(c)

Tabella 1.2 – Tabelle di verità per gli operatori AND(a), OR(b) e NOT(c).

1.2 Funzioni non completamente specificate

Molto spesso è utile avere la possibilità di non specificare l'output della funzione per un dato input. A tale scopo si estende il codominio delle funzioni booleane, ottenendo così il seguente insieme: $\{0, 1, *\}$. Il simbolo appena utilizzato per estendere il codominio delle funzioni booleane viene chiamato *condizione di indifferenza* o *don't care* e, indica che per un dato input la funzione può assumere sia il valore 1 che il valore 0.

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	0
1	0	*
1	1	*

Tabella 1.3 – Tabella di verità della funzione non completamente specificata f.

Di particolare interesse per il problema trattato, è il modo in cui può essere partizionato il dominio di una funzione. Sia $f : \{0,1\}^n \to \{0, 1, *\}$, abbiamo le seguenti definizioni:

Definizione 1. L'insieme $f^{\text{on}} = \{x \in \{0,1\}^n \mid f(x) = 1\}$ è detto *On-Set* di f.

Definizione 2. L'insieme $f^{\text{off}} = \{x \in \{0,1\}^n \mid f(x) = 0\}$ è detto *Off-Set* di f.

Definizione 3. L'insieme $f^{\text{dc}} = \{x \in \{0,1\}^n \mid f(x) = *\}$ è detto *Dc-Set* di f.

Da queste definizioni segue che:

Definizione 4. Sia $f : \{0,1\}^n \to \{0, 1, *\}$, f viene detta *non completamente specificata* se e solo se $f^{\text{dc}} \neq \emptyset$. Altrimenti f è detta *completamente specificata.*

Ad esempio la funzione in Tabella 1.3 è non completamente specificata dato che $f^{on} = \{00\}$, $f^{off} = \{01\}$ e $f^{dc} = \{10, 11\} \neq \emptyset$.

Capitolo 2

Rappresentazione di funzioni booleane

Come spesso accade quando si passa dall'analisi di un problema all'implementazione della sua soluzione, occorre trovare quale o quali siano le strutture migliori da utilizzare per manipolare i dati coinvolti. Nonostante le funzioni booleane siano definite su insiemi finiti, il numero di punti della funzione può crescere molto rapidamente e, rendere anche la più banale delle operazioni intrattabile da un punto di vista computazionale. Basti osservare che se la funzione dipende da n variabili il numero dei suoi punti è 2^n.

Nel capitolo precedente sono stati introdotti due possibili metodi per la rappresentazione di funzioni booleane. Rispettivamente: le tabelle di verità e le espressioni booleane. Tuttavia risultano essere entrambi inappropriati ai nostri scopi. Il primo per via del fatto che equivale a memorizzare tutti i 2^n punti della funzione. Il secondo invece rende alcune operazioni particolarmente complicate. Ad esempio, determinare un assegnamento delle variabili che renda vera un'espressione booleana in forma normale congiuntiva (ovvero un AND di OR di variabili e loro negazioni) è un problema $\mathcal{NP}$-Completo [2].

In questo capitolo si introduce una rappresentazione alternativa che generalmente, permette di ridurre sia lo spazio necessario alla memorizzazione della funzione, che il tempo richiesto per effettuare determinate operazioni.

2.1 Alberi di decisione

Iniziamo con la seguente

Definizione 5. Siano x, y_0, y_1 tre espressioni booleane. Si definisce l'operatore *if-then-else* come

$$x \to y_0, y_1 \equiv (x \wedge y_0) \vee (\neg x \wedge y_1)$$

dove x è detta espressione di test.

Quindi $x \to y_0, y_1$ è vera se x e y_0 sono veri oppure se x è falsa e y_1 è vera. Con l'introduzione di questo nuovo operatore è possibile definire una nuova forma normale per le espressioni booleane:

Definizione 6. Un'espressione booleana è detta in forma normale *if-then-else*, o semplicemente INF, se e soltanto se è costruita interamente tramite l'applicazione del suddetto operatore e le costanti $0, 1$. Inoltre tutte le espressioni di test sono composte da sole variabili.

Si indica ora con $t[0/x]$ e $t[1/x]$ le sostituzioni di ogni occorrenza della variabile x nell'espressione booleana t rispettivamente con le costanti 0 e 1. Si definisce a questo punto una importante equivalenza:

Definizione 7 (Espansione di Shannon)**.** Sia t un'espressione booleana e x una variabile che compare al suo interno. La seguente equivalenza prende il nome di *espansione di Shannon* di t rispetto ad x

$$t \equiv x \to t[1/x], t[0/x]$$

.

Come dimostrato in [3] è possibile ottenere, a partire da un'espressione booleana, una espressione INF equivalente tramite l'applicazione ricorsiva dell'espansione di Shannon.

Esempio 1. Si consideri la seguente espressione booleana: $t \equiv (x_1 \iff y_1) \wedge (x_2 \iff y_2)$. Se si applica ricorsivamente l'espansione di Shannon, selezionando in ordine le variabili x_1, y_1, x_2, y_2 per ottenere la INF di t, si ottengono le seguenti espressioni:

$$t = x_1 \to t_1, t_0$$

$$t_0 = y_1 \to 0, t_{00}$$

$$t_1 = y_1 \to t_{11}, 0$$

$$t_{00} = x_2 \to t_{001}, t_{000}$$

$$t_{11} = x_2 \to t_{111}, t_{110}$$

$$t_{000} = y_2 \to 0, 1$$

$$t_{001} = y_2 \to 1, 0$$

$$t_{110} = y_2 \to 0, 1$$

$$t_{111} = y_2 \to 1, 0$$

Le espressioni ottenute nell'esempio precedente possono essere viste come un albero (Figura 2.1). La radice di tale albero corrisponde al nodo relativo alla prima variabile scelta nel processo di espansione. Ciascun nodo intermedio ha due archi: *low-branch* e *high-branch* che, indicano rispettivamente l'assegnazione del valore 0 oppure 1 alla variabile del nodo da cui si diramano. Le foglie, invece, contengono solo valori di verità: 0 oppure 1.

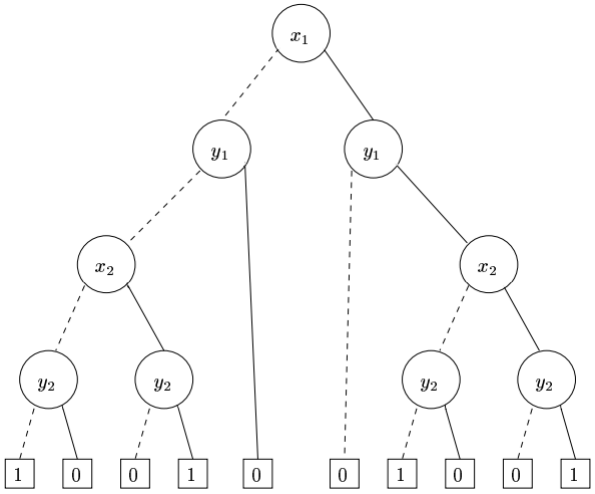


Figura 2.1 – Albero di decisione per $(x_1 \iff y_1) \wedge (x_2 \iff y_2)$

Seppur definito in maniera informale, l'albero appena introdotto prende il nome di *albero di decisione* per una data espressione booleana. Osserviamo che, per come sono stati presentati gli alberi di decisione, questi non risultano essere più adatti ai nostri scopi, di quanto non lo siano le tabelle di verità, difatti lo spazio richiesto per memorizzare una funzione booleana è pressappoco lo stesso.

Nonostante questo, si può notare che alcune delle espressioni da cui abbiamo ricavato l'albero sono identiche. Ad esempio t_{000} è uguale a t_{110}, lo stesso vale per t_{001} e t_{111}. Possiamo quindi sostituire t_{000} a t_{110} nell'espressione t_{11} e t_{001} con t_{111}, ottenendo che anche t_{11} e t_{00} sono uguali. Quindi, possiamo sostituire t_{11} con t_{00} in t_1. In questo modo si ottiene un albero differente (Figura 2.2), con meno nodi intermedi e foglie che, rappresenta sempre la stessa funzione booleana.

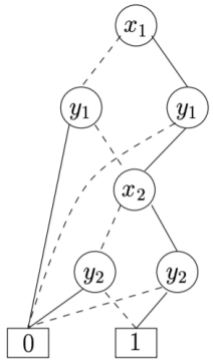


Figura 2.2 – Albero di decisione semplificato per $(x_1 \iff y_1) \wedge (x_2 \iff y_2)$

2.2 Binary decision diagrams

Rimuovendo tutte le espressioni ridondanti da un albero di decisione, come fatto nell'esempio nella sezione precedente, si ottiene quello che prende il nome di *binary decision diagram*, o semplicemente BDD. Quest'ultimo si può formalizzare come segue:

Definizione 8 (BDD)**.** Sia $f : \{0,1\}^n \to \{0,1\}$ e sia $X = \{x_1, ..., x_n\}$ l'insieme delle variabili da cui dipende f. Un BDD per f è un grafo diretto, radicato e aciclico con:

- Uno o due nodi terminali con grado uscente pari a zero ed etichettati come 0 oppure 1.
- Un insieme di nodi u con grado uscente pari a due. I due archi uscenti possono essere ottenuti tramite le funzioni $low(u)$ e $high(u)$. Infine, a ciascun nodo è associata una variabile $var(u) \in X$.

Di particolare interesse sono i BDD con specifiche proprietà. Si introducono di seguito:

Definizione 9 (ROBDD)**.** Un BDD è *ordinato* (OBDD) se su tutti i percorsi attraverso il grafo le variabili rispettano un dato ordine lineare $x_1 < ... < x_n$. Inoltre, un OBDD si dice *ridotto* (ROBDD) se valgono le due seguenti proprietà:

- **Unicità**. Non esistono due nodi distinti etichettati con la stessa variabile e aventi gli stessi archi uscenti.
$$\forall u, v \; low(u) = low(v) \wedge high(u) = high(v) \wedge var(u) = var(v) \Rightarrow u = v$$
- **Non ridondanza**. Non esiste un nodo avente gli stessi successori.
$$\forall \; u \; low(u) \neq high(u)$$

Da ora in poi per semplicità si utilizzerà il termine BDD per riferirsi in realtà ai ROBDD. E' importante notare che l'ordinamento delle variabili scelto durante il processo di costruzione di un BDD riveste un ruolo di grande importanza. Difatti la dimensione finale del BDD dipenderà da tale ordinamento. A tal proposito si consideri nuovamente l'espressione booleana $t \equiv (x_1 \iff y_1) \wedge (x_2 \iff y_2)$. Se come ordine delle variabili si sceglie $x_1 < x_2 < y_1 < y_2$ si ottiene il BDD in Figura 2.3 con 9 nodi interni e non 6 come quello in Figura 2.1, ottenuto tramite l'ordinamento $x_1 < y_1 < x_2 < y_2$. Quindi, se il BDD è di grandi dimensioni, utilizzare un buon ordinamento delle variabili, consente di risparmiare una grande quantità di memoria e, di conseguenza eseguire certe operazioni con più velocità. Purtroppo il problema di determinare quale sia l'ordinamento delle variabili che produce un BDD di dimensione minima ricade nella classe dei problemi $\mathcal{NP}$-Ardui [4]. Tuttavia esistono diverse euristiche per il riordinamento delle variabili che permettono spesso di ottenere dei BDD di dimensioni contenute.

Si conclude dicendo che i BDD presentano alcune caratteristiche che li rendono particolarmente adatti ai nostri scopi. Difatti i BDD sono una forma canonica [3], quindi per ogni funzione $f : \{0,1\}^n \to \{0,1\}$, scelto un ordinamento delle variabili, esiste un unico BDD che la rappresenta. Inoltre, come già accennato, generalmente i BDD permettono di ottenere delle rappresentazioni compatte, in termini di memoria utilizzata, delle funzioni che rappresentano. Ne è un esempio il confronto tra la Figura 2.1 e la Figura 2.2. Infine, molte delle operazioni su funzioni booleane (AND, OR, EXOR, etc...) possono essere efficientemente implementate in termini di manipolazioni di BDD. Quindi il loro costo non dipenderà dal numero di punti delle funzioni, bensì dalla dimensione dei BDD coinvolti, che spesso risulta essere molto ridotta. La Tabella 2.1 illustra il costo computazionale di alcune di queste operazioni [6].

Dunque, queste considerazioni motivano la scelta dei BDD come strutture dati impiegate negli algoritmi descritti nei capitoli successivi.

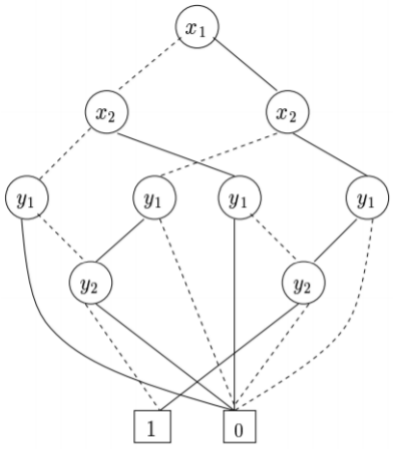


Figura 2.3 – BDD per l'espressione $(x_1 \iff y_1) \wedge (x_2 \iff y_2)$ con ordinamento delle variabili $x_1 < x_2 < y_1 < y_2$.

Operazione	Costo computazionale
$f_1 \vee f_2$	$\mathcal{O}(G_{f_1} \cdot G_{f_2})$
$f_1 \wedge f_2$	$\mathcal{O}(G_{f_1} \cdot G_{f_2})$
$f_1 \oplus f_2$	$\mathcal{O}(G_{f_1} \cdot G_{f_2})$
$f_1 \iff f_2$	$\mathcal{O}(G_{f_1} \cdot G_{f_2})$
$\forall\ x.\ f$	$\mathcal{O}(G_f^2)$
$f_{x_i} oppure f_{\bar{x}_i}$	$\mathcal{O}(G_f)$
$f_{\|x_i = f_2}$	$\mathcal{O}(G_f^2 \cdot G_{f_2})$

Tabella 2.1 – G_{f_i} indica la dimensione del BDD per f_i, ovvero il numero di nodi del relativo grafo.

Capitolo 3

Autosimmetria di funzioni booleane

La progettazione dei circuiti digitali include una fase in cui si effettua la minimizzazione di una funzione booleana. Questo problema generalmente risulta essere $\mathcal{NP}$-Arduo e, tutti gli algoritmi di minimizzazione hanno un tempo di esecuzione esponenziale, nel numero di variabili della funzione considerata. Tuttavia le funzioni trattate derivano sempre da problemi reali e, spesso presentano alcune forme di "regolarità" che possono essere sfruttate per velocizzare il processo di minimizzazione. In questo capitolo, si cerca di dare una caratterizzazione della regolarità di una funzione booleana in termini del suo grado di autosimmetria. La nozione di autosimmetria è stata introdotta in [8] e ulteriormente studiata in [11], [1], [5], [12], [13], dove è stato mostrato come questa regolarità possa essere sfruttata per velocizzare il processo di minimizzazione e, per derivare rappresentazioni logiche più compatte delle funzioni booleane. Pertanto, in tale capitolo si presenteranno i concetti teorici che stanno alla base degli algoritmi che saranno presentati nei capitoli successivi.

3.1 Spazio vettoriale di funzioni booleane

Si inizia definendo l'operatore di disgiunzione esclusiva ($\oplus$) tra due vettori di $\{0,1\}^n$ come:

Definizione 10 ($\oplus$)**.** Si definisce $\oplus : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ come $\omega = \alpha \oplus \beta$ tale che $\omega_i = \alpha_i \oplus \beta_i = \alpha_i + \beta_i \ (mod\ 2)$ con $i = 1, ..., n$.

Ricordiamo dall'algebra che $(\{0,1\}^n, \oplus)$ è uno spazio vettoriale e che, un suo sottospazio vettoriale V è un sottoinsieme di $\{0,1\}^n$, contenente il vettore nullo $\mathbf{0} = 0, ..., 0$ e tale per cui per ogni coppia di punti $v_1, v_2 \in V$ abbiamo che $v_1 \oplus v_2 \in V$. Il sottospazio vettoriale V contiene 2^k vettori, dove k è la dimensione di V ed è generato da una base B contenente k vettori linearmente indipendenti. Infatti, B è un insieme minimo di punti di V tale che ogni altro punto di V è una combinazione di disgiunzioni esclusive di alcuni vettori di B.

Esempio 2. L'insieme $B = \{010, 011\}$ è una base per lo spazio vettoriale $V = \{000, 001, 010, 011\}$. Difatti, possiamo generare ogni vettore di V come una combinazione di disgiunzioni esclusive dei vettori in B: $000 = 010 \oplus 010$ e $001 = 010 \oplus 011$.

Inoltre nel seguito si indica con f l'insieme dei punti in cui la funzione assume valore 1, e con $|f|$ la sua cardinalità. Abbiamo la seguente definizione:

Definizione 11 (Chiusura)**.** Una funzione booleana $f : \{0,1\}^n \to \{0,1\}$ è detta chiusa rispetto ad un vettore $\alpha \in \{0,1\}^n$ se $\forall\ \omega \in \{0,1\}^n$, $\omega \oplus \alpha \in f \Leftrightarrow \omega \in f$.

Esempio 3. La funzione $f = \{001, 010, 100, 111\}$ è chiusa rispetto al vettore 101. Infatti $001 \oplus 101 = 100 \in f$, $010 \oplus 101 = 111 \in f$, e così via.

Ogni funzione f risulta essere chiusa rispetto al vettore nullo $\mathbf{0}$ poiché, per $\omega \in f$ abbiamo che $\mathbf{0} \oplus \omega = \omega$. Inoltre, se f è chiusa rispetto a due vettori differenti $\alpha_1, \alpha_2 \in \{0,1\}^n$, risulta anche essere chiusa rispetto alla loro somma $\alpha_1 \oplus \alpha_2$:

Teorema 1. *Siano $\alpha_1, \alpha_2 \in \{0,1\}^n$ due vettori diversi. Se f è chiusa rispetto a α_1 e α_2 allora risulta anche essere chiusa rispetto a $\alpha_1 \oplus \alpha_2$.*

Dimostrazione. Dire che f è chiusa rispetto a $\alpha_1 \oplus \alpha_2$ significa che $\forall\ \omega \in \{0,1\}^n, \omega \oplus (\alpha_1 \oplus \alpha_2) \in f \Leftrightarrow \omega \in f$. Si procede quindi dimostrando le due implicazioni.

- $\omega \oplus (\alpha_1 \oplus \alpha_2) \in f \Rightarrow \omega \in f$: si riscrive la premessa come $(\omega \oplus \alpha_1) \oplus \alpha_2 = t \oplus \alpha_2 \in f$ con $t = \omega \oplus \alpha_1$. Assumendo vera la premesse e, poiché f è chiusa rispetto a α_2, si ha che $t = \omega \oplus \alpha_1 \in f$. Ma f è chiusa anche rispetto a α_1 e quindi $\omega \in f$.
- $\omega \in f \Rightarrow \omega \oplus (\alpha_1 \oplus \alpha_2) \in f$. Ripetendo il ragionamento precedente si ha che $\omega \oplus \alpha_1 = t \in f$ e quindi $t \oplus \alpha_2 = (\omega \oplus \alpha_1) \oplus \alpha_2 \in f$.

□

Quindi l'insieme di tutti i vettori β tale per cui f è chiusa rispetto a β, è un sottospazio vettoriale di $(\{0,1\}^n, \oplus)$. Dunque otteniamo la definizione:

Definizione 12. L'insieme $L_f = \{\beta \mid f \text{ è chiusa rispetto a } \beta\}$ è lo *spazio vettoriale* di f.

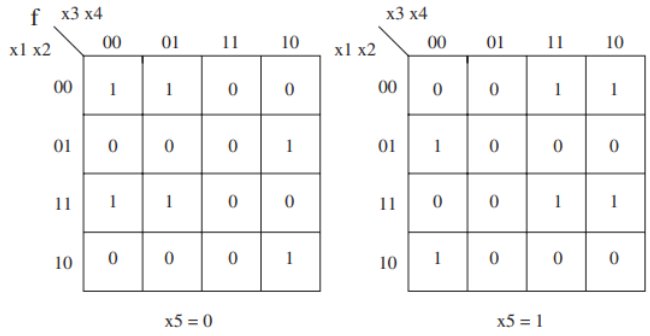


Figura 3.1 – Mappa di Karnaugh per una funzione con 5 variabili

Esempio 4. La funzione f in Figura 3.1 è chiusa rispetto ai vettori in $L_f = \{00000, 00101, 11000, 11101\}$. L_f è un sottospazio vettoriale di $\{0,1\}^5$. Infatti, $\mathbf{0} = 00000 \in L_f, 00101 \oplus 11000 = 11101 \in L_f, 00101 \oplus 11101 = 11000 \in L_f, 11000 \oplus 11101 = 00101 \in L_f$. Il numero di vettori appartenenti a L_f è 2^2 e la dimensione di L_f è 2.

Si procede adesso definendo una base speciale, chiamata canonica, per rappresentare L_f. Consideriamo ora una matrice M di dimensione $2^k \times n$ le cui righe corrispondono ai punti di un generico spazio vettoriale V di dimensione k e le cui colonne corrispondono alle variabili $x_1, x_2, ..., x_n$. Infine numeriamo le righe della matrice M da 0 a $2^k - 1$. Si dice che V è in *ordine binario* se le righe della matrice M sono ordinate come numeri binari crescenti.

Definizione 13 (Base canonica)**.** Sia V uno spazio vettoriale di dimensione k in ordine binario. La *base canonica* B_V di V è l'insieme dei vettori corrispondenti alle righe della matrice M con indici $2^0, 2^1, ..., 2^{k-1}$. Le variabili corrispondenti al primo 1 da sinistra verso destra, di ogni riga della base canonica, sono dette *variabili canoniche* di V. Mentre le altre variabili sono dette *non canoniche*.

Si può facilmente dimostrare che la base canonica è davvero una base vettoriale. Ricordiamo che, per una funzione f, L_f è uno spazio vettoriale. Quindi le variabili canoniche di L_f sono anche chiamate *variabili canoniche di f*. Si osservi inoltre che, le variabili canoniche sono le variabili veramente indipendenti, nello spazio vettoriale considerato, nel senso che possono assumere tutte le possibili combinazioni di valori 0,1. Al contrario, le variabili non canoniche, assumono solo un valore costante oppure sono definite da combinazioni lineari delle variabili canoniche.

Esempio 5. Consideriamo lo spazio vettoriale L_f della funzione f in Figura 3.1. Possiamo organizzare i suoi vettori in una matrice in ordine binario come mostrato nella Tabella 3.1. La base canonica è composta dai vettori che si trovano alle righe 2^0 e 2^1 che sono i vettori 00101 e 11000. Le variabili canoniche di f sono x_1 e x_3. Le variabili rimanenti, quindi x_2, x_4 e x_5, sono non canoniche.

	x_1	x_2	x_3	x_4	x_5
0	0	0	0	0	0
1	0	0	1	0	1
2	1	1	0	0	0
3	1	1	1	0	1

Tabella 3.1

3.1.1 Algoritmo per il calcolo di L_f

Per calcolare in modo efficiente lo spazio vettoriale L_f associato ad una funzione booleana f, si deve riformulare la Definizione 12, come mostrato in [5]

$$L_f = \{\alpha \mid \forall \omega, f(\alpha \oplus \omega) = f(\omega)\}.$$

A partire da questa definizione e, utilizzando i BDD introdotti nel Capitolo 2, si ottiene l'Algoritmo 1 per il calcolo dello spazio vettoriale L_f. Tale algoritmo ha un costo computazionale pari a $\mathcal{O}(|f|+n\cdot G_f^2)$ [5], dove G_f è la dimensione del BDD per f.

Algoritmo 1: VectorSpace

Input: f

1 g_1 = BDD per $f(x_1, x_2, ..., x_n)$;
2 Sia $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ un vettore booleano;
3 g_2 = BDD per $f(\alpha_1 \oplus x_1, \alpha_2 \oplus x_2, ..., \alpha_n \oplus x_n)$;
4 g_3 = BDD per $g_1 \iff g_2$;
5 g_4 = BDD per $\forall\ (x_1, x_2, ..., x_n)\ g_3$;
6 **return** g_4;

3.2 Funzioni k-autosimmetriche

A questo punto siamo pronti per dare una prima caratterizzazione del tipo di regolarità che è oggetto di questo elaborato.

Definizione 14 (k-autosimmetria)**.** Una funzione booleana f è detta *k-autosimmetrica* o, equivalentemente f ha grado di autosimmetria k, con $0 \leq k \leq n$ se, il suo spazio vettoriale L_f ha dimensione k.

Esempio 6. Come osservato nell'Esempio 4, lo spazio vettoriale L_f della funzione f in Figura 3.1 ha dimensione 2. Pertanto la funzione f è 2-autosimmetrica.

Più generalmente si dice che f è *autosimmetrica* se il suo grado k è strettamente maggiore di 0.

La Definizione 14 risulta essere troppo restrittiva poiché non fornisce la nozione di autosimmetria quando la funzione considerata è non completamente specificata. A tal proposito si danno due nuove definizioni di autosimmetria per estendere quest'ultima anche al caso di funzioni booleane non completamente specificate.

Definizione 15 (k-autosimmetria)**.** Data una funzione f non completamente specificata, questa viene detta *k-autosimmetrica* se la funzione completamente specificata g, con $g^{\text{on}} = f^{\text{on}} \cup f^{\text{dc}}$, è *k-autosimmetrica*.

Definizione 16 (k-autosimmetria)**.** Data una funzione f non completamente specificata, questa viene detta *k-autosimmetrica* se la funzione completamente specificata g, con $g^{\text{off}} = f^{\text{off}} \cup f^{\text{dc}}$, è *k-autosimmetrica*.

Nonostante le precedenti due nuove definizioni estendano il concetto di autosimmetria anche al caso di funzioni non completamente specificate, queste ultime risultano essere ancora troppo restrittive. A tal proposito si consideri la funzione espressa in Tabella 3.2. In base alla Definizione 15 la funzione f in Tabella 3.2, non dovrebbe essere autosimmetrica in quanto $g^{\text{on}} = f^{\text{on}} \cup f^{\text{dc}} = \{00, 01, 10\}$ e il suo spazio vettoriale L_g contiene solo il vettore 00. Si osservi inoltre che, anche mettendo tutti i *don't care* a zero, f non risulta autosimmetrica. Tuttavia possiamo partizionare i due elementi del *dc-set* in modo differente e ottenere la funzione h:

$$h^{\text{on}} = f^{\text{on}} \cup \{01\}$$

$$h^{\text{off}} = f^{\text{off}} \cup \{10\}$$

Si osservi innanzitutto che la funzione h ha lo stesso comportamento di f in quanto $f^{\text{on}} \subseteq h^{\text{on}} \subseteq (f^{\text{on}} \cup f^{\text{dc}})$ e, inoltre, è completamente specificata. In accordo con la Definizione 14, h è 1-autosimmetrica, visto che $L_h = \{00, 01\}$.

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	*
1	0	*
1	1	0

Tabella 3.2 – Tabella di verità della funzione 1-autosimmetrica f.

3.2.1 Autosimmetria generalizzata

Dalle considerazioni fatte nella sezione precedente emerge la necessità di fare delle scelte più accurate sul partizionamento del *dc-set*.

Quindi, come prima cosa, è necessario estendere la definizione di spazio vettoriale di una funzione booleana, al caso di funzioni non completamente specificate. Si consideri l'insieme dei vettori $\alpha \in \{0,1\}^n$ che ci permettono di traslare un punto di f^{on} in uno appartenente a $f^{on} \cup f^{dc}$:

Definizione 17. $S = \{\alpha \mid \forall\ \omega \in \{0,1\}^n,\ \omega \in f^{on} \Rightarrow \omega \oplus \alpha \in f^{on} \cup f^{dc}\}$.

Si osservi che l'insieme S contiene gli spazi vettoriali L_0 e L_1 associati rispettivamente alle funzioni completamente specificate f_0 e f_1 che, hanno come *on-set* rispettivamente f^{on} e $f^{\mathrm{on}} \cup f^{\mathrm{dc}}$. In altre parole S contiene gli spazi vettoriali delle funzioni completamente specificate ottenute mettendo rispettivamente tutti i *don't care* a 0 e a 1.

Teorema 2. *Sia* $L_0 = \{\alpha \in \{0,1\}^n \mid \forall\ \omega \in f^{on},\ \omega \oplus \alpha \in f^{on}\}$ *e sia* $L_1 = \{\alpha \in \{0,1\}^n \mid \forall\ \omega \in f^{on} \cup f^{dc},\ \omega \oplus \alpha \in f^{on} \cup f^{dc}\}$. *Allora* $L_0 \subseteq S$ *e* $L_1 \subseteq S$.

Dimostrazione. Sia $\alpha \in L_0$. Allora, per ogni $\omega \in f^{on}$ si ha che $\alpha \oplus \omega \in f^{on}$. Quindi $\alpha \in S$ dato che $f^{on} \subseteq f^{on} \cup f^{dc}$. Ora sia $\alpha \in L_1$. Abbiamo che $\alpha \in S$ poiché in questo caso la proprietà di chiusura vale per tutti gli $\omega \in f^{on} \cup f^{dc}$ e, in particolare per ogni $\omega \in f^{on}$. □

Notiamo tuttavia che $(S, \oplus)$ non è sempre uno spazio vettoriale. Ad esempio, si consideri la funzione in Tabella 3.2, $S = \{00, 01, 10\}$, ma $(10 \oplus 01) = 11 \notin S$, da cui si deduce che S non è uno spazio vettoriale. Ovvio anche perché $|S|$ è dispari, in particolare 3. Per poter sfruttare quanto introdotto nella sezione precedente, è necessario lavorare su uno spazio vettoriale. Un possibile approccio è quello di considerare il più grande spazio vettoriale contenuto in S, ottenendo la seguente definizione:

Definizione 18. Sia f una funzione booleana non completamente specificata, sia $S = \{\alpha \mid \forall\ \omega \in \{0,1\}^n, \omega \in f^{on} \Rightarrow \omega \oplus \alpha \in f^{on} \cup f^{dc}\}$ e sia L_S il più grande spazio vettoriale contenuto in S. Il grado di autosimmetria di f è pari alla dimensione di L_S.

Osserviamo che il più grande spazio vettoriale contenuto in un insieme, non necessariamente è unico a parità di dimensione. Ad esempio, l'insieme $S = \{00, 01, 10\}$ contiene due spazi vettoriali, rispettivamente $\{00, 10\}$ e $\{00, 01\}$, entrambi di dimensione massima pari ad uno. Tuttavia indipendentemente da quale venga calcolato, il grado di autosimmetria rilevato non cambia.

3.2.2 Algoritmo per il calcolo di S

Di seguito si propone l'algoritmo per il calcolo dell'insieme S (Algoritmo 2). Si noti che i parametri u e h corrispondono rispettivamente alla funzione caratteristica di $f^{on} \cup f^{dc}$ e f^{on}.

Teorema 3. *Per ogni funzione booleana non completamente specificata* f, *sia* $f' = f^{on} \cup f^{dc}$, *la procedura Build_S calcola l'insieme* S *in tempo* $\mathcal{O}(|f'| + n \cdot (G_{f'} \cdot G_f)^2)$. *Dove* G_f *e* $G_{f'}$ *sono le dimensioni dei BDD per* f *e* f'.

Dimostrazione. In maniera analoga a quanto mostrato in [5] per l'Algoritmo 1, ordinando le variabili come $\alpha_1, x_1, ..., \alpha_n, x_n$ il BDD per g_3 ha una dimensione proporzionale a $G_{f'}$. In accordo alla Tabella 2.1, il calcolo di g_4 richiede tempo $\mathcal{O}(G_{f'} \cdot G_f)$ e produce un BDD dello stesso ordine di grandezza. Di conseguenza il calcolo di g_5 viene effettuato in tempo $\mathcal{O}(n \cdot (G_{f'} \cdot G_f)^2)$. Infine il tempo necessario alla costruzione di g_1 e g_2 è proporzionale a $|f|$ e $|f'|$. Dato che $f \subseteq f'$, il costo complessivo dell'Algoritmo 2 è $\mathcal{O}(|f'| + n \cdot (G_{f'} \cdot G_f)^2)$. □

Algoritmo 2: Build_S

Input: u, h

Output: BDD per l'insieme S

1 g_1 = BDD per $h(x_1, x_2, ..., x_n)$;
2 g_2 = BDD per $u(x_1, x_2, ..., x_n)$;
3 Sia $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ un vettore booleano;
4 g_3 = BDD per $u(\alpha_1 \oplus x_1, \alpha_2 \oplus x_2, ..., \alpha_n \oplus x_n)$;
5 g_4 = BDD per $g_1 \Rightarrow g_3$;
6 g_5 = BDD per $\forall\ (x_1, x_2, ..., x_n)\ g_4$;
7 **return** g_5;

Capitolo 4

Copertura lineare di un insieme di vettori

Nel capitolo precedente, dopo aver formalmente introdotto il concetto di autosimmetria, è stato proposto un procedimento per calcolare il grado di autosimmetria di una funzione non completamente specificata. Come già discusso questo comporta il calcolo del più grande spazio vettoriale contenuto nell'insieme S. Questo problema, a differenza di quello complementare, ovvero trovare il più piccolo spazio vettoriale che contiene l'insieme S, sembra essere particolarmente complesso dal punto di vista computazionale. Pertanto, in tale capitolo, si esporrà un algoritmo per il calcolo del più piccolo spazio vettoriale, contenente un dato insieme che, sfrutta gli spazi affini. Tale spazio, in algebra lineare prende il nome di *copertura lineare* di un insieme. L'algoritmo sviluppato sarà utilizzato per le euristiche che saranno presentate successivamente nel corso dell'elaborato.

4.1 Spazi affini

Per un vettore $\alpha \in \{0,1\}^n$ e per un sottospazio vettoriale V di $(\{0,1\}^n, \oplus)$, si consideri l'insieme $\alpha \oplus V = \{\alpha \oplus v \mid v \in V\}$. In un certo senso il vettore α è utilizzato per "traslare" un sottospazio vettoriale. Abbiamo:

Definizione 19. Sia V un sottospazio vettoriale di $(\{0,1\}^n, \oplus)$. Dato un punto $\alpha \in \{0,1\}^n$, l'insieme $A = \alpha \oplus V = \{\alpha \oplus v \mid v \in V\}$ è uno *spazio affine* su V con punto di traslazione α.

Esempio 7. Per il sottospazio vettoriale $S = \{000, 010, 011, 001\}$ e il vettore $\alpha = 110$, l'insieme $A = \alpha \oplus S = \{110, 100, 101, 111\}$ è uno spazio affine su S.

Una proprietà interessante degli spazi affini è che $(\alpha \oplus V \equiv V) \iff \alpha \in V$, ovvero, se si sceglie come α un punto di V allora lo spazio affine A è lo spazio vettoriale V stesso. Se A è uno spazio affine, esiste un unico spazio vettoriale V tale che per ogni $\alpha \in A$, $A = \alpha \oplus V$. Questo spazio può essere calcolato come $V = \alpha \oplus A$, dove α è un qualunque punto di A. Inoltre, se A e A' sono spazi affini sullo stesso spazio vettoriale V, allora A e A' o coincidono oppure sono disgiunti.

Esempio 8. Si consideri lo spazio affine $A = \{0010, 0011, 0100, 0101\}$. Il nostro scopo è trovare l'unico spazio vettoriale V tale che $A = \alpha \oplus V$. Scegliendo $\alpha = 0010 \in A$, abbiamo: $V = \alpha \oplus A = 0010 \oplus A = \{0000, 0001, 0110, 0111\}$ che è uno spazio vettoriale. È facile verificare che scegliendo un vettore diverso in A come punto di traslazione si ottiene lo stesso risultato, ovvero $V = 0010 \oplus A = 0011 \oplus A = 0100 \oplus A = 0101 \oplus A$.

4.2 Rappresentazione di spazi affini

Poiché il punto di traslazione α può essere scelto come un qualsiasi vettore di A e, lo spazio vettoriale V può essere rappresentato da una qualsiasi delle sue basi, dobbiamo definire una rappresentazione unica di A. A tal fine introduciamo un po' di notazioni.

Definizione 20. Sia A uno spazio affine su uno spazio vettoriale V. Il punto di *traslazione canonica* α_A è il punto minimo di A in ordine binario.

Definizione 21. Sia V uno spazio vettoriale la cui matrice è ordinata in ordine binario, con le righe indicizzate da 0 a $2^k - 1$. E sia $A = \alpha \oplus V$ uno spazio affine su V. L'insieme di punti di V con indici $2^0, 2^1, ..., 2^k - 1$ sarà chiamato *base canonica* B_A di V (o, equivalentemente, di A).

Definizione 22. La *rappresentazione canonica* (α_A, B_A) di uno spazio affine A è data dal suo punto di traslazione canonica insieme alla sua base canonica.

La funzione caratteristica di uno spazio affine può essere rappresentata da una semplice espressione, chiamata *pseudoprodotto*, che consiste in un AND di EXOR di letterali [7]. Ad esempio, $x_1x_2(x_3 \oplus x_4)(x_3 \oplus x_6 \oplus x_7)$ è uno pseudoprodotto che rappresenta lo spazio affine $A = (\alpha_A, B_A)$ = (1000001, {0000011, 0000100, 0011001}). La funzione caratteristica di uno spazio affine può essere espressa in vari modi come uno pseudoprodotto. Inoltre, la rappresentazione di uno spazio affine non è univoca e pseudoprodotti diversi possono essere funzioni caratteristiche dello stesso spazio affine. Tra queste forme, un'espressione canonica (CEX) data in [8] è di particolare rilevanza poiché fornisce una rappresentazione univoca per gli spazi affini.

4.3 Algoritmo per il calcolo della copertura lineare

L'eliminazione di Gauss-Jordan, dato un insieme booleano S, ci permette di calcolare il più piccolo spazio affine che contiene S, a partire dalla rappresentazione SOP[1] dell'insieme stesso. Tale procedura, consente di ottenere una matrice in forma a scalini ridotta per righe [9]. Sia ora $m = |S|$. Se eseguiamo l'eliminazione di Gauss-Jordan sulla matrice $m \times n$ le cui righe sono mintermini dell'insieme S o prodotti che rappresentano mintermini dell'insieme S, otteniamo una matrice equivalente in forma a scalini ridotta per righe, dalla quale possiamo facilmente estrarre una base, per il più piccolo spazio vettoriale contenente l'insieme S. Ci interessa ottenere lo spazio affine più piccolo contenente l'insieme S, poiché, la sua dimensione può essere inferiore. A tal fine, notiamo che se il vettore nullo (il vettore con tutte le componenti a zero) appartiene all'insieme S, allora A è uno spazio vettoriale, infatti, ogni volta che uno spazio affine contiene il vettore nullo, allora questo è effettivamente uno spazio vettoriale. Altrimenti, S può essere contenuto in uno spazio affine che non è uno spazio vettoriale.

Esempio 9. Si consideri l'insieme S dei vettori binari di $\{0,1\}^3$ con esattamente una componente uguale a 1. Si ha che $S = \{001, 010, 100\}$. Il più piccolo sottospazio vettoriale che contiene S è tutto lo spazio vettoriale $\{0,1\}^3$ che, ha dimensione 3. Se invece si considera il più piccolo spazio affine che contiene S, questo è $A = (\alpha_A, B_A) = (001, \{011, 101\}) = \{001, 010, 100, 111\}$ che, ha dimensione 2. Di conseguenza lo spazio affine che contiene S è più piccolo dello spazio vettoriale che contiene S.

A questo punto è possibile costruire A tramite l'esecuzione dell'Algoritmo 3. Per prima cosa si sceglie un qualsiasi mintermine di S, chiamiamolo v e, si calcola l'insieme $v \oplus S$. Se il vettore nullo appartiene a S, scegliamo $v = \mathbf{0}$. Calcoliamo quindi il più piccolo spazio vettoriale V_A contenente $v \oplus S$ mediante eliminazione di Gauss-Jordan. Infine deriviamo $A \supset S$ da V_A come $A = v \oplus V_A$, dove v è lo stesso vettore scelto nel primo passo. E' interessante notare che, ogni volta che S non contiene il vettore nullo, possiamo scegliere qualsiasi mintermine $v \in S$ nel primo passo senza cambiare il risultato finale, cioè lo spazio affine A [10].

Si noti che, l'Algoritmo 3, genera lo spazio in questione a partire dalla rappresentazione SOP di un'insieme, senza dover generare tutti i suoi mintermini. Si descrive l'intuizione di questo approccio con un esempio pratico.

Esempio 10. Si consideri l'insieme S = {000000, 001000, 010001, 010011, 011001, 011011} che è rappresentato dalla seguente SOP: $\bar{x}_1\bar{x}_2\bar{x}_4\bar{x}_5\bar{x}_6 + \bar{x}_1x_2\bar{x}_4x_6$. Il prodotto $\bar{x}_1x_2\bar{x}_4x_6$ è rappresentato nella forma PLA (per maggiori dettagli sul formato PLA si consultino i capitoli successivi) dalla riga 01-0-1. Per ogni *don't care* nel prodotto, possiamo generare un vettore composto da tutti 0, tranne un 1 nella posizione corrispondente al *don't care*. Nel nostro caso generiamo i vettori 001000 e 000010. Questi vettori sarebbero sicuramente generati durante la fase di eliminazione di Gauss-Jordan. Infatti abbiamo: 001000 = 010001⊕011001 e 000010 = 010001⊕010011. La matrice che deve essere elaborata dall'algoritmo di eliminazione di Gauss-Jordan conterrà quindi: il vettore originale con 0 al posto del don't care (010001) e, i nuovi vettori generati (001000 e 000010). Quindi il primo prodotto può essere rappresentato da 000000 e 001000 e il secondo da 010001, 001000 e 000010. Pertanto, l'input per la fase di eliminazione di Gauss-Jordan è dato dall'insieme di vettori: $\{000000, 001000, 010001, 00010\}$.

La complessità dell'Algoritmo 3 è $\mathcal{O}(nP^2)$ dove P è il numero di prodotti nella SOP data per l'insieme S [10].

4.3.1 Costruire un BDD dall'espressione CEX

L'Algoritmo 3, come descritto precedentemente, restituisce l'espressione CEX per la copertura lineare di S. Ai fini di questo studio sarà necessario manipolare e eseguire algoritmi sullo spazio restituito

[1]Sum Of Product. E' una disgiunzione di prodotti. Un prodotto è una congiunzione di letterali

Algoritmo 3: Build_A

Input: Rappresentazione SOP dell'insieme S

Output: L'espressione CEX per lo spazio affine A

```
1  // Quando uno degli operandi è "*", l'operatore ⊕ restituisce "*";
2  // Il vettore e_i è l'i-esimo vettore della base canonica di {0,1}^n;
3  // 0 è il vettore nullo;
4  if (0 ∈ S) then
5  |  v = 0;
6  else
7  |  v = qualunque mintermine di S;
8  g = v ⊕ S;
9  GE = ∅;
10 for each p ∈ g do
11 |  for i = 1, ..., n do
12 |  |  if (p[i] == " * ") then
13 |  |  |  GE = {e_i} ∪ GE;
14 |  |  |  p[i] = "0";
15 |  |  GE = {p} ∪ GE;
16 // Si noti che GE contiene punti senza don't care;
17 V_A = GaussJordanElimination(GE);
18 if (dim(V_A) == n) then
19 |  return ∅;
20 else
21 |  A = v ⊕ V_A;
22 |  return A;
```

dall'Algoritmo 3. Viste tutte le considerazioni fatte nel Capitolo 2, ancora una volta, ricorreremo all'utilizzo dei BDD per la rappresentazione del suddetto spazio. Si procede quindi descrivendo l'idea intuitiva che ci permette di passare dall'espressione CEX di un insieme di vettori alla sua rappresentazione tramite i BDD.

Ricordiamo che un'espressione CEX consiste in prodotto di fattori EXOR. Pertanto, per costruire un BDD che rappresenta lo stesso spazio di una espressione CEX si deve:

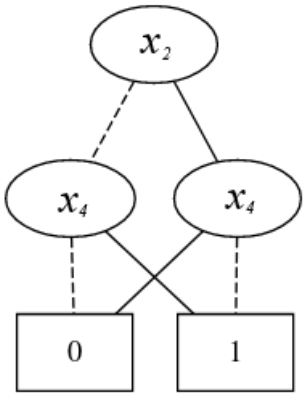
1. per ogni fattore realizzare un BDD che fa lo EXOR di tutte le variabile nel fattore considerato;
2. costruire il BDD finale facendo l'AND di tutti i BDD costruiti precedentemente.

Si mostra il processo descritto intuitivamente con un esempio pratico.

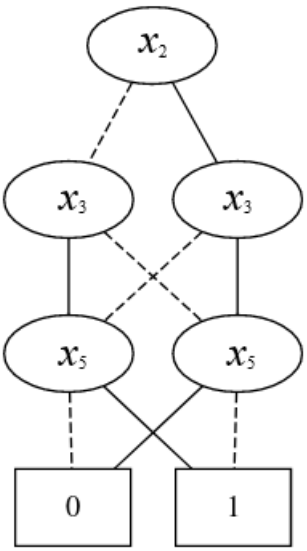
Esempio 11. Sia A lo spazio affine definito come $A = (\alpha_A, B_A) = (000100, \{\ 001010, 010110, 100000\})$. Tale spazio affine è rappresentato come $\text{CEX}(A) = (x_2 \oplus x_4)(x_2 \oplus x_3 \oplus \bar{x}_5)\bar{x}_6$. Per ottenere il BDD di A a partire dalla CEX si deve:

1. costruire il BDD per $x_2 \oplus x_4$;
2. costruire il BDD per $x_2 \oplus x_3 \oplus \bar{x}_5$;
3. costruire il BDD per $\bar{x}_6$;

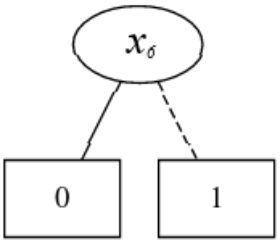
4. costruire il BDD per l'AND dei BDD ottenuti ai passi precedenti.



(a) BDD per il punto 1.



(b) BDD per il punto 2.



(c) BDD per il punto 3.

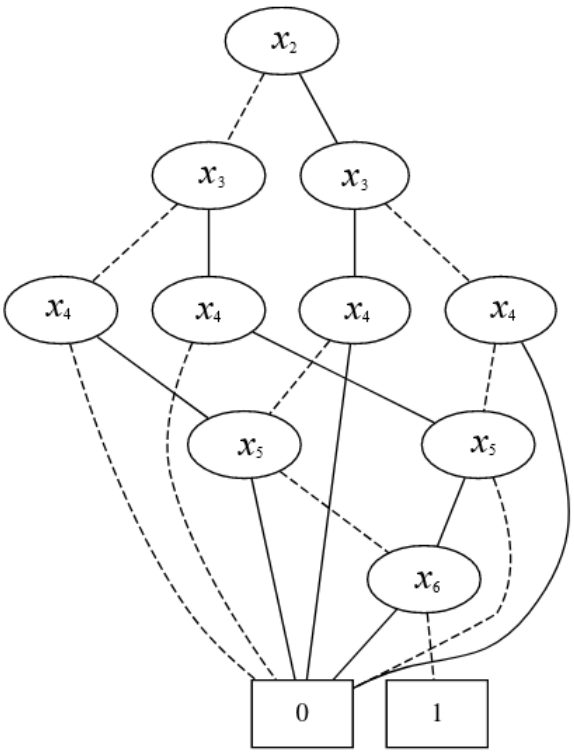


Figura 4.2 – BDD per il punto 4.

Capitolo 5

Costruzione dello spazio vettoriale L_S

Nel Capitolo 3 è stata introdotta la nozione di autosimmetria per funzioni non completamente specificate. In particolare è stato mostrato che, per ottenere il grado di autosimmetria di una funzione booleana non completamente specificata, occorre calcolare il più grande spazio vettoriale contenuto in S. Nel capitolo precedente si è accennato al fatto che il calcolo di tale spazio sembri essere particolarmente complesso dal punto di vista computazionale. In questo capitolo si discuterà in modo più approfondito la complessità di tale problema e, dopo di ciò, saranno presentate alcune euristiche. La prima euristica che verrà presentata è ripresa da un precedente lavoro su questo argomento. La seconda euristica che è proposta è un miglioramento della prima. La terza euristica, invece, creando lo spazio vettoriale generato da un sottoinsieme dei vettori appartenenti a una base della copertura lineare di S e decrementando di volta in volta il numero dei vettori considerati, si ferma, quando lo spazio generato è contenuto in S.

5.1 Algoritmo enumerativo esatto

Un algoritmo enumerativo che ci permetta di calcolare effettivamente lo spazio vettoriale L_S, potrebbe consistere in una ricerca esaustiva, di un sottospazio vettoriale contenuto in S, in ordine di dimensione decrescente a partire da $d = \lfloor \log_2 |S| \rfloor$. Tale algoritmo ha una complessità computazionale che cresce in modo esponenziale, in quanto richiede di enumerare tutti i possibili sottoinsiemi di i vettori linearmente indipendenti di S, il cui numero è limitato superiormente dal coefficiente binomiale gaussiano $\binom{r}{i}_2 = \mathcal{O}(2^{i(r-i)})$, dove r è il numero massimo di vettori linearmente indipendenti di S, e $1 \leq i \leq d$. Si consultino [14] e [15] per maggiori dettagli su tale stima.

Esempio 12. Si consideri l'insieme $S = \{0000, 0011, 1000, 1001, 1010, 1011\}$. Dato che $\lfloor \log_2 |S| \rfloor = \lfloor \log_2 6 \rfloor = 2$, l'algoritmo inizierà considerando i sottospazi vettoriali generati da un insieme di 2 vettori linearmente indipendenti di S. Tale algoritmo termina la sua esecuzione con il sottoinsieme $B = \{0011, 1001\}$ che genera il sottospazio vettoriale 2-dimensionale $L_S = \{0000, 0011, 1001, 1010\}$, il quale è contenuto in S.

L'Algoritmo 4, facendo uso dell'Algoritmo 3 presentato nel Capitolo precedente, calcola effettivamente lo spazio vettoriale L_S.

Algoritmo 4: Build_L_S

Input: BDD per l'insieme S
Output: BDD per lo spazio vettoriale L_S

```
1 d = ⌊log2 |S|⌋;
2 while d ≥ 1 do
3     Sia ps un insieme di tutte le possibili combinazioni di d vettori linearmente indipendenti
        di S;
4     for each p ∈ ps do
5         A = Build_A(p);
6         if A ⊆ S then
7             return A;
8     d = d − 1;
```

5.2 Euristica di espansione di L_0

L'euristica che si propone per il calcolo dello spazio L_S, prenderà in input il BDD dell'insieme S e quello della funzione completamente specificata f_0, definita nel seguente modo:

$$f_0^{\text{on}} = f^{\text{on}} \text{ e } f_0^{\text{off}} = f^{\text{dc}} \cup f^{\text{off}}.$$

f_0 è quindi la funzione f in cui si assegnano tutti i *don't care* a zero. Inoltre, l'euristica considerata, utilizza tre nuovi operatori, ovvero la differenza tra insiemi, la traslazione di un insieme rispetto ad un punto e l'estrazione di un mintermine. I primi due, in termini di operazioni fra BDD, possono essere implementati come segue:

- **Differenza tra insiemi**: siano A e B i BDD delle funzioni caratteristiche degli stessi insiemi:

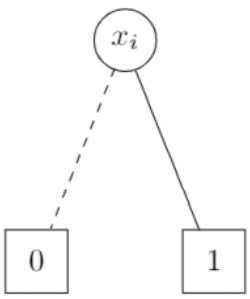
  $$A \backslash B = A \wedge \neg B$$

  In accordo alla Tabella 2.1, la differenza $A \backslash B$ può essere calcolata in tempo $\mathcal{O}(G_A \cdot G_B)$ e produce un BDD dello stesso ordine di grandezza.

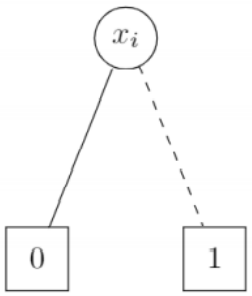
- **Traslazione di un insieme rispetto ad un punto**: sia $\alpha \in \{0,1\}^n$ un vettore booleano, sia A il BDD della funzione caratteristica dello stesso insieme nelle variabili $x_1, x_2, ..., x_n$:

  $$\alpha \oplus A = A_{|x_i = x_i \oplus \alpha_i} \text{ con } i = 1, 2, ..., n$$

Possiamo notare che l'operazione $x_i \oplus \alpha_i$ richiede tempo costante, in quanto se $\alpha_i = 0$ allora il nodo di x_i non deve essere modificato. Mentre se $\alpha_i = 1$, basterà scambiare gli archi uscenti di x_i, come mostrato in Figura 5.1. Dunque la traslazione di un insieme rispetto ad un punto può essere calcolata in tempo $\mathcal{O}(G_A)$. Inoltre la dimensione del BDD risultante sarà la stessa di quella iniziale



(a) BDD per $x_i \oplus 0$.



(b) BDD per $x_i \oplus 1$.

Figura 5.1

Per quanto riguarda l'estrazione di un mintermine, in [6] viene mostrato che può essere implementata in tempo $\mathcal{O}(n)$, dove n è il numero di variabili del BDD preso in considerazione.

L'idea intuitiva di questa euristica è la seguente: partendo dallo spazio vettoriale L_0, si decide di volta in volta se espanderlo con lo spazio affine $A = \alpha \oplus L_S$, dove α è un mintermine scelto dallo spazio $S \setminus L_S$.

A questo punto siamo pronti per presentare l'Algoritmo 5.

Teorema 4. *Per ogni funzione booleana non completamente specificata f, sia $f' = f^{on} \cup f^{dc}$, sia S l'insieme calcolato tramite l'Algoritmo 2 e sia f_0 la funzione f dove si assegnano tutte le condizioni di indifferenza a zero. La chiamata di procedura Build_L_S_espansione(f_0, S), calcola uno spazio vettoriale L_S contenuto in S, tale per cui $|L_S| \geq |L_f|$, in tempo al più esponenziale in $|f'|$.*

Algoritmo 5: Build_L_S_espansione

Input: BDD per la funzione f_0 e per l'insieme S

Output: BDD per lo spazio vettoriale L_S calcolato

```
1  L_0 = VectorSpace(f_0);
2  L_S = L_0;
3  S' = S;
4  while |S'| ≥ |L_S| do
5      α = mintermine scelto dal BDD per S' \ L_s;
6      A = α ⊕ L_S;
7      if A ⊆ S then
8          L_S = L_S ∪ A;
9      else
10         S' = S' \ A;
11 return L_S;
```

Dimostrazione. La procedura Build_L_S_espansione(f_0, S) calcola uno spazio vettoriale estendendo quello ottenuto tramite la procedura VectorSpace applicata alla funzione f_0 (si veda l'Algoritmo 1). Ad ogni iterazione, L_S può essere aggiornato facendo l'unione con lo spazio affine $(\alpha \oplus L_S)$, dove α è un punto scelto da $S' \setminus L_S$. Quindi l'insieme restituito al termine delle iterazioni mantiene la struttura di spazio vettoriale.

Si osservi inoltre che, ogni volta che viene fatta l'unione con lo spazio affine A, si aumenta la dimensione dello spazio vettoriale risultante.

Per quanto riguarda la sua dimensione, in qualunque modo venga calcolato L_f segue dalla sua definizione che, ogni vettore in L_f appartiene anche ad L_S, dunque la sua dimensione sarà minore o uguale a quella di L_S.

Infine le iterazioni eseguite sono $\mathcal{O}(|S'|-|L_S|)$. Il caso pessimo si può verificare quando VectorSpace(f_0) restituisce uno spazio vettoriale di dimensione zero. In tal caso, per come è definito l'insieme S', il numero delle iterazioni è proporzionale a $|f'|$. Notiamo infine che ad ogni iterazione il BDD A ha dimensione pari a G_{L_S}, l'aggiornamento di S' coinvolge A: dunque la sua dimensione viene moltiplicata per un fattore $\mathcal{O}(G_{L_S})$ ad ogni iterazione. Di conseguenza il tempo di esecuzione dell'Algoritmo 5 comprenderà un termine che può risultare esponenziale in $|f'|$. □

L'algoritmo appena presentato, trattandosi di un'euristica, non dà alcuna garanzia sull'ottimalità della soluzione calcolata. Possiamo notare questo comportamento supponendo di avere una funzione, tale per cui il relativo insieme S contiene i seguenti vettori: $S = \{00000, 00001, 00100, 01000, 01100, 10000, 10100, 11000, 11100\}$. Supponiamo inoltre che il suo spazio vettoriale, calcolato nella prima istruzione dell'Algoritmo 5, sia composto dal solo vettore nullo. Se come primo mintermine viene selezionato il vettore 00001, si ottiene lo spazio $L_f = \{00000, 00001\}$ di dimensione 1, che non può essere ulteriormente esteso. Invece il più grande spazio vettoriale contenuto in S è composto dai seguenti 8 vettori: $S = \{00000, 00100, 01000, 01100, 10000, 10100, 11000, 11100\}$.

5.3 Euristica di espansione di L_0 e L_1

Si ricorda che f_0 e f_1 sono le funzioni completamente specificate tali che: $f_0^{on} = f^{on}$ e $f_1^{on} = f^{on} \cup f^{dc}$, i cui spazi vettoriali associati sono rispettivamente L_0 e L_1.

Come dimostrato tramite il Teorema 2, L_0 e L_1 sono entrambi contenuti in S. Questo ci porta ad una nuova euristica che estende il comportamento dell'euristica presentata precedentemente. Difatti, tale euristica, espande lo spazio vettoriale L_0 e L_1 per poi restituire come risultato lo spazio espanso che ha dimensione maggiore.

Tale approccio euristico è implementato dall'Algoritmo 6.

Algoritmo 6: Build_L_S_espansione_L_0_L_1

Input: BDD per la funzione f_0, f_1 e per l'insieme S
Output: BDD per lo spazio vettoriale L_S calcolato

```
1 L_{S_0} = Build_L_S_espansione(f_0, S);
2 L_{S_1} = Build_L_S_espansione(f_1, S);
3 if |L_{S_0}| ≥ |L_{S_1}| then
4 |   return L_{S_0};
5 else
6 |   return L_{S_1};
```

La correttezza dell'Algoritmo 6 è garantita dal Teorema 2 e dalla correttezza stessa dell'Algoritmo 5. Per quanto riguarda la complessità, poiché l'Algoritmo 5 potrebbe risultare esponenziale nel numero di punti della funzione completamente specificata f', anche questo algoritmo potrebbe risultare esponenziale.

5.4 Euristica di contrazione della copertura lineare

L'idea dell'euristica che si propone ora, è la seguente: una volta ottenuto l'insieme S, si calcolano i vettori linearmente indipendenti di S. Questi sono una base, che chiamiamo B, per la copertura lineare di S. A questo punto si procede considerando tutti i possibili sottoinsiemi di vettori appartenenti alla base B, in ordine di dimensione decrescente a partire da $d = \log_2 |S|$. A questo punto, se lo spazio vettoriale generato dai vettori considerati è contenuto in S, abbiamo finito.

Tale approccio euristico è implementato dall'Algoritmo 7.

Algoritmo 7: Build_L_S_contrazione

Input: BDD per l'insieme S

Output: BDD per lo spazio vettoriale L_S

```
1 d = ⌊log2 |S|⌋;
2 B = base per la copertura lineare di S;
3 while d ≥ 1 do
4     Sia ps un insieme di tutte le possibili combinazioni di d vettori appartenenti a B;
5     for each p ∈ ps do
6         A = Build_A(p);
7         if A ⊆ S then
8             return A;
9     d = d − 1;
```

Per quanto concerne la complessità dell'Algoritmo 7, si ha che, a ogni iterazione, si devono considerare tutte le possibili combinazioni di r vettori appartenenti alla base B, con $r = 1, ..., \lfloor \log_2 |S| \rfloor$. Le iterazioni del ciclo *while* più esterno sono al più $d = \lfloor \log_2 |S| \rfloor$. Il ciclo *for* più interno è ripetuto al più $\sum_{r=1}^{d} \binom{m}{r}$ volte, dove m è la dimensione della copertura lineare di S ($d < m$). Il costo del corpo di questo ciclo è invece polinomiale nella dimensione dell'input. Il costo dell'Algoritmo 7 è dunque dominato dal numero di iterazioni, che cresce come $\mathcal{O}(d \cdot 2^m)$.

Capitolo 6

Implementazione e risultati sperimentali

In questo capitolo si esporranno le principali scelte implementative fatte per la realizzazione degli algoritmi presentati nel corso dell'elaborato. Dopo di ciò, saranno raccolti e messi a confronto i risultati sperimentali ottenuti eseguendo i test di autosimmetria con 6 algoritmi differenti. Tali algoritmi, sono stati eseguiti su un sottoinsieme della ben nota suit di benchmark *Espresso* [16], contenente solo le funzioni aventi dei *don't care* nell'output. In totale sono state esaminate 30 funzioni e, un numero complessivo di output pari a 482.

Le caratteristiche della macchina utilizzata per eseguire la sperimentazione sono:

- sistema operativo: Ubuntu (64-bit) (macchina virtuale);
- processore: Intel Core i5-7200U;
- CPU: 2.50 GHz $\times$ 2;
- memoria RAM: 4.0 GB

6.1 Formato PLA

Le funzioni utilizzate per valutare i metodi proposti sono state fornite secondo lo standard *Espresso* [16], che permette di dare una specifica a due livelli di una funzione booleana. Ciascuna funzione viene rappresentata da un file con estensione .pla, contenente un insieme di opzioni seguite da una matrice di caratteri. Ai fini dell'implementazione degli algoritmi discussi, le opzioni utilizzate sono le seguenti:

- .i $<n>$ il numero di input;
- .o $<n>$ il numero di output.

La specifica della funzione inizia dopo le prime righe contenenti le opzioni. Supponiamo che n sia il numero di input ed m il numero di output della funzione, per ciascuna riga i primi n caratteri specificano un prodotto. Ogni posizione all'interno di questi n caratteri indica una variabile di input. Ciascun carattere può essere 0, se la corrispondente variabile appare negata nel prodotto; 1 se compare affermata e - se non deve essere presente. Segue uno spazio ed altri m caratteri che possono assumere valore 1, se il corrispondente prodotto compare nell'espressione dell'uscita considerata, 0 se non compare nell'espressione e * per indicare una condizione di indifferenza. Un possibile esempio di funzione specificata tramite lo standard *Espresso* è riportato di seguito.

.i	2
.o	1
00	1
01	*
10	*
11	0

Tabella 6.1 – Specifica della funzione f in Tabella 3.2 tramite il formato PLA.

6.2 CUDD: Colorado University Decision Diagram

Tutte le operazioni tra BDD sono state implementate con l'ausilio della libreria *Colorado University Decision Diagram* [17]. CUDD permette di creare e manipolare diversi tipi di Decision Diagram, tra cui i BDD introdotti nel Capitolo 2. Ciascun BDD è implementato componendo delle strutture dette *DdNode* o semplicemente nodi che, oltre a diverse informazioni di gestione, contengono: un campo *Index*, che identifica in maniera univoca la variabile assegnata a quel nodo, un campo *Value* che rappresenta un valore di verità e infine due puntatori che rappresentano i due archi uscenti di ciascun nodo.

Inoltre, utilizza un bit per indicare il complemento di un arco. Così quando un BDD viene valutato, l'attraversamento di un arco complementato fa' si che si consideri il complemento del valore ottenuto al termine della valutazione. Questo permette di ottenere delle rappresentazioni ancora più efficienti in termini di memoria utilizzata. CUDD prevede l'utilizzo di una particolare struttura dati, detta *DdManager*, che deve essere inizializzata prima dell'utilizzo della libreria e, deve essere passata come parametro a quasi ogni funzione. Lo scopo principale di questa struttura è quello di mantenere ciascun BDD utilizzato nel corso dell'applicazione in forma canonica.

Come discusso nel Capitolo 2, l'ordinamento delle variabili impatta notevolmente sulla dimensione di un BDD. A tal proposito CUDD fornisce un insieme di algoritmi per il riordinamento dinamico delle variabili. L'ordinamento può essere effettuato esplicitamente tramite una chiamata di funzione, anche se viene comunque effettuato quando il numero di nodi di un BDD raggiunge una certa soglia.

Infine, dato che manipolare diversi BDD contemporaneamente può richiedere una grande quantità di memoria, CUDD mette a disposizione dell'utente un sistema di Garbage Collection basato su reference counting. Ciascun nodo possiede un contatore che indica da quanti altri nodi è riferito. I contatori sono incrementati e decrementati dall'utente tramite chiamate esplicite di funzione. Quando il Garbage Collector va in esecuzione recupera solo la memoria dei nodi con contatore uguale a zero.

6.3 Struttura del codice

Tutti gli algoritmi presentati nell'elaborato sono stati implementati in *C*. Di seguito si da una breve panoramica dei moduli sviluppati.

- **autosymmetry.c**: contiene l'implementazione degli Algoritmi 1, 2, 4, 5, 6 e 7 che, sono stati presentati nel corso dell'elaborato. Ciascuno di essi manipola i BDD delle funzioni ricevute in input tramite le funzionalità messe a disposizione dalla libreria CUDD.
- **parser.c**: effettua il parsing dei file pla, descritti precedentemente. Fornendo per ciascun output della funzione descritta nel file, un BDD separato. In oltre contiene funzioni di utilità, ad esempio per scrivere un BDD in formato pla.
- **binmat.c**: implementa le matrici binarie e fornisce funzione per manipolarle. In particolare offre la funzione che esegue l'eliminazione di Gauss-Jordan.
- **logic.c**: contiene l'implementazione dell'Algoritmo 3 e alcune funzioni di supporto a tale algoritmo.
- **all_zeros_test.c** e **all_ones_test.c**: implementano il test di autosimmetria assegnando tutti i don't care rispettivamente a zero ed uno.
- **generalized_test_Uno.c**: Implementa il test di autosimmetria descritto dall'*euristica uno* (Algoritmo 5).
- **generalized_test_Uno_Migliorata.c**: Implementa il test di autosimmetria descritto dall'*euristica uno migliorata* (Algoritmo 6).
- **generalized_test_Due.c**: Implementa il test di autosimmetria descritto dall'*euristica due* (Algoritmo 7).
- **generalized_test_Ottimo.c**: Implementa il test di autosimmetria descritto dall'*algoritmo enumerativo esatto* (Algoritmo 4).

6.4 Risultati sperimentali

Come già accennato nella parte introduttiva del capitolo, ai fini di fornire una prova empirica di quanto discusso nel corso dell'elaborato, sono stati sviluppati diversi test. Il primo è il test di autosimmetria standard, introdotto in [5], eseguito sul solo *on-set* della funzione che, di fatto, assegna tutte le condizioni di indifferenza a zero: i risultati di tale test sono riportati nella seconda colonna delle Tabelle 6.3 e 6.4. Il secondo invece, effettua il test di autosimmetria sulla funzione ottenuta assegnando tutte le condizioni di indifferenza ad uno, secondo la definizione data in [1]: i risultati di tale test sono riportati nella terza colonna delle Tabelle 6.3 e 6.4. Il terzo test implementa l'*euristica di espansione di* L_0 presentata nella Sezione 5.2: i risultati di tale test sono riportati nella quarta colonna della Tabella 6.4 e nella seconda colonna della Tabella 6.2. Il quarto test implementa l'*euristica di contrazione della copertura lineare* presentata nella Sezione 5.4: i risultati di tale test sono riportati nella quinta colonna della Tabella 6.4 e nella terza colonna della Tabella 6.2. Il quinto test implementa l'*euristica di espansione di* L_0 *e* L_1 presentata nella Sezione 5.3: i risultati di tale test sono riportati nella sesta colonna della Tabella 6.4 e nella quarta colonna della Tabella 6.3. Infine, il sesto e ultimo test, implementa l'*algoritmo enumerativo esatto* presentato nella Sezione 5.1: i risultati di tale test sono riportati nella settima colonna della Tabella 6.4 e nella quarta colonna della Tabella 6.2.

Lo scopo delle Tabelle 6.2 e 6.3 è quello di raccogliere e confrontare, per ogni funzione, i gradi di autosimmetria medi e massimi ottenuti tramite l'esecuzione dei test che sono stati presentati all'inizio di questa sezione. Inoltre in tali tabelle sono anche riportati, per ogni funzione, i tempi di esecuzione dei test. Nella Tabella 6.4, a differenza delle Tabelle 6.2 e 6.3, è stato considerato ciascun output di una funzione come se fosse una funzione distinta, pertanto per ogni output di una funzione è riportato il grado di autosimmetria. In tutte le tabelle, sono riportati solo i risultati ottenuti dalle funzioni che sono risultate essere autosimmetriche e per le quali si hanno i valori di autosimmetria ottimi. Nelle Tabelle A.1, A.2 e A.3, contenute nell'Appendice A, sono riportati i risultati sperimentali completi ottenuti sull'insieme di tutte le funzioni benchmark che presentano *don't care* nell'output.

Di seguito si fornisce una breve descrizione dei dati presentati:

- *Funzione*: il nome della funzione presa in considerazione.
- n, m: rispettivamente il numero di input e output di una funzione.
- k: grado di autosimmetria medio degli output di una funzione.
- k_0, k_1: grado di autosimmetria rilevato per un singolo output di una funzione, assegnando rispettivamente tutti i *don't care* a 0 e a 1.
- k_{L_0}: grado di autosimmetria rilevato per un singolo output di una funzione tramite l'*euristica di espansione di* L_0 introdotta nella sezione 5.2.
- k_{cont}: grado di autosimmetria rilevato per un singolo output di una funzione tramite l'*euristica di contrazione della copertura lineare* introdotta nella Sezione 5.4.
- $k_{L_0L_1}$: grado di autosimmetria rilevato per un singolo output di una funzione tramite l'*euristica di espansione di* L_0 *e* L_1 introdotta nella Sezione 5.3.
- k_{ottimo}: grado di autosimmetria rilevato per un singolo output di una funzione tramite l'algoritmo enumerativo esatto, presentato nella Sezione 5.1, quando disponibile.
- Max: massimo grado di autosimmetria rilevato per un output di una funzione.
- $Time$: tempi di esecuzione espressi in millisecondi.

- −: l'esecuzione del test, dopo una soglia massima di 30 minuti, è stata abortita. Pertanto i risultati per quel test non sono disponibili.

Con un confronto preliminare tra la Tabella 6.4 e la Tabella A.1, si può notare che per il 36.1% delle funzioni testate non è disponibile il grado di autosimmetria ottimo, dato l'elevato costo computazionale dell'*algoritmo enumerativo esatto*. Inoltre, sempre confrontando le due tabelle in questione, si osserva che, le funzioni che sono risultate essere autosimmetriche sono il 71.5% dell'insieme complessivo dei benchmark esaminati.

Come si può osservare dalla Tabella 6.4, considerando ogni output come una funzione a se stante, l'*euristica di contrazione della copertura lineare* rileva un grado di autosimmetria minore rispetto a quello rilevato tramite l'*euristica di espansione di* L_0. Difatti, il grado di autosimmetria rilevato con l'*euristica di contrazione della copertura lineare*, diminuisce del 2.38% rispetto a quello rilevato tramite l'*euristica di espansione di* L_0. Continuando a osservare la Tabella 6.4 si evince che, l'*euristica di espansione di* L_0 *e* L_1 presenta un aumento del grado di autosimmetria rilevato di circa lo 0.23% rispetto all'*euristica di espansione di* L_0 al costo però di un maggiore tempo di esecuzione. Il grado di autosimmetria rilevato con l'*algoritmo enumerativo esatto* aumenta rispetto a quello rilevato tramite l'*euristica di espansione di* L_0 *e* L_1 dell'1.18%. Infine, è interessante notare che, l'*algoritmo enumerativo esatto*, ha rilevato un grado di autosimmetria maggiore rispetto a quello rilevato da tutte le altre euristiche solo per le funzioni *bench(4), p1(2), p1(16), p3(2)* e *test4(18)*, a fronte delle 308 funzioni per cui è disponibile il grado di autosimmetria ottimo.

Si prendono ora in considerazione le Tabelle 6.2 e 6.3, dove si considerano gli output aggregati delle funzioni. Si osservi che l'*euristica di contrazione della copertura lineare* ha rilevato un grado di autosimmetria medio inferiore del 3.33% rispetto a quello rilevato dall'*euristica di espansione di* L_0, mentre il grado massimo di autosimmetria rilevato risulta essere invariato. Confrontando invece l'*euristica di espansione di* L_0 *e* L_1 con l'*euristica di espansione di* L_0 si osserva che la prima euristica rileva un grado di autosimmetria medio maggiore dello 0.91% rispetto alla seconda, rilevando anche un grado di autosimmetria massimo maggiore di un unità.

Confrontando invece i tempi di esecuzione si evince che, l'*euristica di contrazione della copertura lineare* è la più efficiente in quanto impiega il 14.6% di tempo in meno dell'*euristica di espansione di* L_0 e il 54.1% di tempo in meno dell'*euristica di espansione di* L_0 *e* L_1.

Avendo concluso le osservazioni sulla sperimentazione emerge quanto segue: tra tutte le euristiche presentate, l'*euristica di espansione di* L_0 *e* L_1 è quella che ha rilevato l'autosimmetria maggiore mentre l'*euristica di contrazione della copertura lineare* è quella più efficiente e con il miglior rapporto qualità del risultato su tempo di esecuzione. Come già discusso, in alcuni casi nessuna delle euristiche presentate ha rilevato il grado di autosimmetria ottimo.

	Euristica di esp. di L_0			Euristica di contrazione			Algo. enumerativo esatto		
Funzione, n, m	k	Max	$Time$	k	Max	$Time$	k	Max	$Time$
alu2, 10, 8	3.375	7	4.177	3.375	7	4.909	3.375	7	5.027
alu3, 10, 8	2.750	6	3.288	2.750	6	3.451	2.750	6	3.396
apla, 10, 12	4.417	6	41.733	3.250	6	53.136	-	-	-
b10, 15, 11	2.636	15	10.006	2.636	15	11.351	2.636	15	54.840
bcc, 26, 45	10.755	14	59.558	10.755	14	53.864	10.755	14	60.919
bcd.div3, 4, 4	1.500	2	0.203	1.500	2	0.792	1.500	2	0.903
bench, 6, 8	1.875	3	1.937	1.750	3	3.759	2.000	3	210.973
dekoder, 4, 7	1.285	2	0.403	1.000	1	1.523	1.285	2	1.489
dk17, 10, 11	5.727	7	21.873	4.909	7	37.886	-	-	-
dk27, 9, 9	6.444	8	5.436	6.222	8	12.677	-	-	-
dk48, 15, 17	11.588	12	131.919	11.941	13	10212.934	-	-	-
exam, 10, 10	4.100	10	28.322	4.000	10	47.231	-	-	-
exp, 8, 18	1.444	8	2.101	1.444	8	6.809	1.444	8	7.699
exps, 8, 38	0.395	4	2.490	0.263	3	13.520	-	-	-
fout, 6, 10	0.100	1	0.260	0.100	1	3.208	0.100	1	2.999
inc, 7, 9	1.888	4	0.778	1.888	4	2.415	1.888	4	2.179
l8err, 8, 8	4.000	8	0.432	4.000	8	2.006	4.000	8	1.826
p1, 8, 18	3.222	8	15.285	2.883	8	18.098	3.500	8	90.018
p3, 8, 14	3.786	8	11.309	3.375	8	13.451	4.071	8	87.273
pdc, 16, 40	9.875	14	757.339	9.600	14	5976.430	-	-	-
spla, 16, 46	3.978	13	27899.712	3.695	13	7807.671	-	-	-
t2, 17, 16	9.500	13	22.802	9.062	12	1035.178	-	-	-
t4, 12, 8	6.875	8	19.487	7.000	10	35.667	-	-	-
test4, 8, 30	0.633	1	19.609	0.666	2	28.654	0.700	2	34.740
x1dn, 27, 6	12.500	16	846.334	13.000	16	136.116	-	-	-
wim, 4, 7	1.285	2	0.361	1.000	1	1.350	1.285	2	2.591
Totale	115.933	200	29907.154	112.064	200	25524.086	-	-	-

Tabella 6.2 – Risultati sperimentali ottenuti tramite l'esecuzione dei tre algoritmi per le funzioni per cui si è rilevato un grado di autosimmetria maggiore di 0.

	All zeros test			All ones test			Euristica di esp. di L_0 e L_1		
Funzione, n, m	k	Max	$Time$	k	Max	$Time$	k	Max	$Time$
alu2, 10, 8	2.750	6	1.805	1.125	6	2.513	3.375	7	13.702
alu3, 10, 8	2.750	6	1.095	0.625	2	2.836	2.750	6	9.439
apla, 10, 12	0.166	1	1.439	0.083	1	2.713	4.500	6	77.206
b10, 15, 11	2.545	15	5.534	1.181	3	6.710	2.636	15	25.076
bcc, 26, 45	10.755	14	36.243	10.755	14	39.360	10.755	14	143.765
bcd.div3, 4, 4	0.500	2	0.130	0.250	1	0.130	1.500	2	0.432
bench, 6, 8	0.000	0	0.686	0.000	0	0.875	1.875	3	3.501
dekoder, 4, 7	0.000	0	0.252	0.285	1	0.133	1.285	2	0.756
dk17, 10, 11	0.090	1	0.877	0.090	1	1.029	5.727	7	41.845
dk27, 9, 9	0.333	1	0.468	0.111	1	0.503	6.555	8	10.377
dk48, 15, 17	0.176	2	1.620	0.058	1	1.805	11.647	12	237.318
exam, 10, 10	1.000	10	3.342	1.100	10	4.545	4.200	10	56.728
exp, 8, 18	1.111	8	1.605	0.000	0	3.232	1.444	8	12.900
exps, 8, 38	0.184	1	5.946	0.131	1	7.379	0.395	4	28.519
fout, 6, 10	0.000	0	1.135	0.000	0	0.923	0.100	1	4.366
inc, 7, 9	0.777	3	0.496	0.888	3	0.524	1.888	4	1.983
l8err, 8, 8	3.000	8	0.884	4.000	8	0.501	4.000	8	2.720
p1, 8, 18	1.111	8	1.624	0.000	0	3.452	3.277	8	28.023
p3, 8, 14	1.428	8	1.153	0.000	0	1.625	3.857	8	20.813
pdc, 16, 40	3.275	14	10.387	0.000	0	22.038	10.000	14	1724.054
spla, 16, 46	2.521	13	9.754	2.521	13	8.678	3.978	13	51957.550
t2, 17, 16	8.625	12	1.213	8.187	11	1.248	9.625	14	41.546
t4, 12, 8	2.250	3	0.810	2.250	3	0.900	6.875	8	27.282
test4, 8, 30	0.000	0	19.946	0.000	0	8.826	0.633	1	78.493
x1dn, 27, 6	9.166	13	2.894	3.000	3	5.923	12.833	16	1091.619
wim, 4, 7	0.142	1	0.140	0.285	1	0.125	1.285	2	0.789
Totale	54.655	150	111.478	36.925	84	128.526	116.995	201	55640.802

Tabella 6.3 – Risultati sperimentali ottenuti tramite l'esecuzione dei tre algoritmi per le funzioni per cui si è rilevato un grado di autosimmetria maggiore di 0.

Tabella 6.4 – Sono riportati i valori di autosimmetria rilevati degli algoritmi considerati su ciascun output delle funzioni per il sottoinsieme di output per cui si ha a disposizione il grado ottimo.

Funzione (Output)	k_0	k_1	k_{L_0}	k_{cont}	$k_{L_0L_1}$	k_{ottimo}
alu2(0)	6	0	7	7	7	7
alu2(1)	4	0	5	5	5	5
alu2(2)	2	0	3	3	3	3
alu2(3)	0	0	1	1	1	1
alu2(4)	1	1	1	1	1	1
alu2(5)	0	0	1	1	1	1
alu2(6)	3	2	3	3	3	3
alu2(7)	6	6	6	6	6	6
alu3(0)	6	0	6	6	6	6
alu3(1)	4	0	4	4	4	4
alu3(2)	2	0	2	2	2	2
alu3(4)	1	1	1	1	1	1
alu3(6)	3	2	3	3	3	3
alu3(7)	6	2	6	6	6	6

b10(0)	15	1	15	15	15	15
b10(1)	1	1	1	1	1	1
b10(2)	3	3	3	3	3	3
b10(3)	4	3	4	4	4	4
b10(4)	1	1	2	2	2	2
b10(5)	2	2	2	2	2	2
b10(6)	1	1	1	1	1	1
b10(10)	1	1	1	1	1	1
bcc(0)	11	11	11	11	11	11
bcc(1)	11	11	11	11	11	11
bcc(2)	11	11	11	11	11	11
bcc(3)	10	10	10	10	10	10
bcc(4)	10	10	10	10	10	10
bcc(5)	10	10	10	10	10	10
bcc(6)	10	10	10	10	10	10
bcc(7)	10	10	10	10	10	10
bcc(8)	10	10	10	10	10	10
bcc(9)	10	10	10	10	10	10
bcc(10)	10	10	10	10	10	10
bcc(11)	10	10	10	10	10	10
bcc(12)	10	10	10	10	10	10
bcc(13)	10	10	10	10	10	10
bcc(14)	11	11	11	11	11	11
bcc(15)	11	11	11	11	11	11
bcc(16)	10	10	10	10	10	10
bcc(17)	11	11	11	11	11	11
bcc(18)	10	10	10	10	10	10
bcc(19)	11	11	11	11	11	11
bcc(20)	11	11	11	11	11	11
bcc(21)	11	11	11	11	11	11
bcc(22)	11	11	11	11	11	11
bcc(23)	10	10	10	10	10	10
bcc(24)	11	11	11	11	11	11
bcc(25)	11	11	11	11	11	11
bcc(26)	11	11	11	11	11	11
bcc(27)	11	11	11	11	11	11
bcc(28)	11	11	11	11	11	11
bcc(29)	11	11	11	11	11	11
bcc(30)	11	11	11	11	11	11
bcc(31)	11	11	11	11	11	11
bcc(32)	11	11	11	11	11	11
bcc(33)	11	11	11	11	11	11
bcc(34)	11	11	11	11	11	11
bcc(35)	11	11	11	11	11	11

bcc(36)	11	11	11	11	11	11
bcc(37)	11	11	11	11	11	11
bcc(38)	11	11	11	11	11	11
bcc(39)	11	11	11	11	11	11
bcc(40)	11	11	11	11	11	11
bcc(41)	14	14	14	14	14	14
bcc(42)	11	11	11	11	11	11
bcc(43)	11	11	11	11	11	11
bcc(44)	11	11	11	11	11	11
bcd.div3(0)	2	1	2	2	2	2
bcd.div3(1)	0	0	1	1	1	1
bcd.div3(2)	0	0	2	2	2	2
bcd.div3(3)	0	0	1	1	1	1
bench(0)	0	0	3	3	3	3
bench(1)	0	0	3	3	3	3
bench(2)	0	0	1	1	1	1
bench(3)	0	0	2	1	2	2
bench(4)	0	0	1	1	1	2
bench(5)	0	0	3	3	3	3
bench(6)	0	0	1	1	1	1
bench(7)	0	0	1	1	1	1
dekoder(0)	0	1	2	1	2	2
dekoder(1)	0	1	2	1	2	2
dekoder(2)	0	0	1	1	1	1
dekoder(3)	0	0	1	1	1	1
dekoder(4)	0	0	1	1	1	1
dekoder(5)	0	0	1	1	1	1
dekoder(6)	0	0	1	1	1	1
exp(0)	0	0	3	3	3	3
exp(3)	8	0	8	8	8	8
exp(4)	8	0	8	8	8	8
exp(6)	1	0	1	1	1	1
exp(10)	0	0	1	1	1	1
exp(11)	0	0	1	1	1	1
exp(12)	2	0	2	2	2	2
exp(13)	1	0	2	2	2	2
exps(5)	1	1	1	1	1	1
exps(6)	1	1	1	1	1	1
exps(16)	1	1	1	1	1	1
exps(18)	1	1	1	1	1	1
exps(19)	1	1	1	1	1	1
exps(31)	1	0	3	1	3	3
exps(32)	1	0	4	3	4	4
fout(4)	0	0	1	1	1	1

inc(0)	1	1	1	1	1	1
inc(1)	1	1	1	1	1	1
inc(4)	0	1	4	4	4	4
inc(5)	1	1	3	3	3	3
inc(6)	0	1	4	4	4	4
inc(7)	1	0	1	1	1	1
inc(8)	3	3	3	3	3	3
l8err(0)	8	8	8	8	8	8
l8err(1)	8	8	8	8	8	8
l8err(2)	8	8	8	8	8	8
l8err(7)	0	8	8	8	8	8
p1(0)	0	0	6	7	6	7
p1(1)	0	0	2	1	2	2
p1(2)	0	0	2	1	2	3
p1(3)	8	0	8	8	8	8
p1(4)	8	0	8	8	8	8
p1(5)	0	0	2	1	2	2
p1(6)	1	0	2	1	2	2
p1(7)	0	0	2	1	3	3
p1(8)	0	0	1	1	1	1
p1(9)	0	0	3	2	3	3
p1(10)	0	0	3	2	3	3
p1(11)	0	0	3	2	3	3
p1(12)	2	0	5	6	5	6
p1(13)	1	0	6	6	6	6
p1(14)	0	0	1	1	1	1
p1(15)	0	0	1	1	1	1
p1(16)	0	0	2	1	2	3
p1(17)	0	0	1	1	1	1
p3(0)	0	0	6	7	6	7
p3(1)	0	0	2	1	2	2
p3(2)	0	0	2	1	2	3
p3(3)	8	0	8	8	8	8
p3(4)	8	0	8	8	8	8
p3(5)	0	0	2	1	2	2
p3(6)	1	0	2	1	2	2
p3(7)	0	0	2	1	3	3
p3(8)	0	0	1	1	1	1
p3(9)	0	0	3	2	3	3
p3(10)	0	0	3	2	3	3
p3(11)	0	0	3	2	3	3
p3(12)	2	0	5	6	5	6
p3(13)	1	0	6	6	6	6
test4(2)	0	0	1	1	1	1

test4(4)	0	0	1	1	1	1
test4(5)	0	0	1	1	1	1
test4(6)	0	0	1	1	1	1
test4(7)	0	0	1	1	1	1
test4(8)	0	0	1	1	1	1
test4(9)	0	0	1	1	1	1
test4(11)	0	0	1	1	1	1
test4(12)	0	0	1	1	1	1
test4(13)	0	0	1	1	1	1
test4(14)	0	0	1	2	1	2
test4(17)	0	0	1	1	1	1
test4(18)	0	0	1	1	1	2
test4(20)	0	0	1	1	1	1
test4(21)	0	0	1	1	1	1
test4(22)	0	0	1	1	1	1
test4(23)	0	0	1	1	1	1
test4(25)	0	0	1	1	1	1
test4(26)	0	0	1	1	1	1
x1dn(0)	11	3	11	11	11	11
x1dn(1)	9	3	9	9	9	9
wim(0)	0	0	1	1	1	1
wim(1)	0	0	1	1	1	1
wim(2)	0	1	2	1	2	2
wim(3)	0	0	1	1	1	1
wim(4)	1	1	2	1	2	2
wim(5)	0	0	1	1	1	1
wim(6)	0	0	1	1	1	1
Totale	677	567	839	819	841	851

Capitolo 7

Conclusioni

Si conclude riassumendo i risultati principali di questo elaborato. E' stato studiato dettagliatamente il grado di autosimmetria delle funzioni booleane non completamente specificate, proponendo una definizione più generale che sfrutta a pieno il grado di libertà presente nelle funzioni booleane non completamente specificate. Tuttavia rilevare il grado di autosimmetria di una funzione non completamente specificata tramite l'approccio presentato, richiede di calcolare il più grande spazio vettoriale contenuto in un dato insieme. Tale problema sembra essere particolarmente difficile da un punto di vista computazionale. Sono stati quindi sviluppati e testati separatamente diversi procedimenti euristici che consentono di realizzare il test di autosimmetria di tali funzioni, oltre ad aver anche presentato un algoritmo enumerativo esatto per tale problema. Sono stati infine riportati e confrontati i risultati sperimentali ottenuti, discutendo quali siano le euristiche da preferire rispetto a certe metriche.

Rimane un problema teorico molto interessante quello di definire, in modo efficiente, lo spazio vettoriale più grande contenuto in un dato insieme di vettori booleani. Un'altra possibile direzione futura è lo studio delle proprietà dei BDD che rappresentano spazi affini, al fine di migliorare ulteriormente l'efficienza dei metodi proposti, ad esempio mediante la definizione di un ordinamento ad hoc delle variabili.

Appendice A

Risultati sperimentali completi

Si presentano infine i risultati sperimentali completi ottenuti sull'insieme di tutte le funzioni benchmark che presentano *don't care* nell'output. Si indica con "$-$", la non terminazione del test dopo una soglia massima di 30 minuti. Pertanto, i risultati per tale test non saranno disponibili.

Tabella A.1 – Sono riportati i valori di autosimmetria rilevati dagli algoritmi considerati su ciascun output delle funzioni.

Funzione (Output)	k_0	k_1	k_{L_0}	k_{cont}	$k_{L_0L_1}$	k_{ottimo}
alu2(0)	6	0	7	7	7	7
alu2(1)	4	0	5	5	5	5
alu2(2)	2	0	3	3	3	3
alu2(3)	0	0	1	1	1	1
alu2(4)	1	1	1	1	1	1
alu2(5)	0	0	1	1	1	1
alu2(6)	3	2	3	3	3	3
alu2(7)	6	6	6	6	6	6
alu3(0)	6	0	6	6	6	6
alu3(1)	4	0	4	4	4	4
alu3(2)	2	0	2	2	2	2
alu3(3)	0	0	0	0	0	0
alu3(4)	1	1	1	1	1	1
alu3(5)	0	0	0	0	0	0
alu3(6)	3	2	3	3	3	3
alu3(7)	6	2	6	6	6	6
apla(0)	0	0	4	3	4	-
apla(1)	0	0	4	3	4	-
apla(2)	0	0	4	3	4	-
apla(3)	0	0	4	3	4	-
apla(4)	0	0	4	2	4	-
apla(5)	1	0	6	5	6	-
apla(6)	0	0	5	6	5	-
apla(7)	0	0	4	2	4	-
apla(8)	0	0	5	3	5	-

apla(9)	0	0	4	2	4	-
apla(10)	0	0	4	3	5	-
apla(11)	1	1	5	4	5	-
b10(0)	15	1	15	15	15	15
b10(1)	1	1	1	1	1	1
b10(2)	3	3	3	3	3	3
b10(3)	4	3	4	4	4	4
b10(4)	1	1	2	2	2	2
b10(5)	2	2	2	2	2	2
b10(6)	1	1	1	1	1	1
b10(7)	0	0	0	0	0	0
b10(8)	0	0	0	0	0	0
b10(9)	0	0	0	0	0	0
b10(10)	1	1	1	1	1	1
bcc(0)	11	11	11	11	11	11
bcc(1)	11	11	11	11	11	11
bcc(2)	11	11	11	11	11	11
bcc(3)	10	10	10	10	10	10
bcc(4)	10	10	10	10	10	10
bcc(5)	10	10	10	10	10	10
bcc(6)	10	10	10	10	10	10
bcc(7)	10	10	10	10	10	10
bcc(8)	10	10	10	10	10	10
bcc(9)	10	10	10	10	10	10
bcc(10)	10	10	10	10	10	10
bcc(11)	10	10	10	10	10	10
bcc(12)	10	10	10	10	10	10
bcc(13)	10	10	10	10	10	10
bcc(14)	11	11	11	11	11	11
bcc(15)	11	11	11	11	11	11
bcc(16)	10	10	10	10	10	10
bcc(17)	11	11	11	11	11	11
bcc(18)	10	10	10	10	10	10
bcc(19)	11	11	11	11	11	11
bcc(20)	11	11	11	11	11	11
bcc(21)	11	11	11	11	11	11
bcc(22)	11	11	11	11	11	11
bcc(23)	10	10	10	10	10	10
bcc(24)	11	11	11	11	11	11
bcc(25)	11	11	11	11	11	11
bcc(26)	11	11	11	11	11	11
bcc(27)	11	11	11	11	11	11
bcc(28)	11	11	11	11	11	11
bcc(29)	11	11	11	11	11	11

bcc(30)	11	11	11	11	11	11
bcc(31)	11	11	11	11	11	11
bcc(32)	11	11	11	11	11	11
bcc(33)	11	11	11	11	11	11
bcc(34)	11	11	11	11	11	11
bcc(35)	11	11	11	11	11	11
bcc(36)	11	11	11	11	11	11
bcc(37)	11	11	11	11	11	11
bcc(38)	11	11	11	11	11	11
bcc(39)	11	11	11	11	11	11
bcc(40)	11	11	11	11	11	11
bcc(41)	14	14	14	14	14	14
bcc(42)	11	11	11	11	11	11
bcc(43)	11	11	11	11	11	11
bcc(44)	11	11	11	11	11	11
bcd.div3(0)	2	1	2	2	2	2
bcd.div3(1)	0	0	1	1	1	1
bcd.div3(2)	0	0	2	2	2	2
bcd.div3(3)	0	0	1	1	1	1
bench1(0)	0	0	0	0	0	0
bench1(1)	0	0	0	0	0	0
bench1(2)	0	0	0	0	0	0
bench1(3)	0	0	0	0	0	0
bench1(4)	0	0	0	0	0	0
bench1(5)	0	0	0	0	0	0
bench1(6)	0	0	0	0	0	0
bench1(7)	0	0	0	0	0	0
bench1(8)	0	0	0	0	0	0
bench(0)	0	0	3	3	3	3
bench(1)	0	0	3	3	3	3
bench(2)	0	0	1	1	1	1
bench(3)	0	0	2	1	2	2
bench(4)	0	0	1	1	1	2
bench(5)	0	0	3	3	3	3
bench(6)	0	0	1	1	1	1
bench(7)	0	0	1	1	1	1
dekoder(0)	0	1	2	1	2	2
dekoder(1)	0	1	2	1	2	2
dekoder(2)	0	0	1	1	1	1
dekoder(3)	0	0	1	1	1	1
dekoder(4)	0	0	1	1	1	1
dekoder(5)	0	0	1	1	1	1
dekoder(6)	0	0	1	1	1	1
dk17(0)	0	0	5	5	5	-

dk17(1)	0	0	6	5	6	-
dk17(2)	0	0	5	4	5	-
dk17(3)	0	0	5	3	5	-
dk17(4)	0	0	6	4	6	-
dk17(5)	0	0	7	7	7	-
dk17(6)	0	0	7	7	7	-
dk17(7)	0	0	6	7	6	-
dk17(8)	0	0	5	2	5	-
dk17(9)	0	0	5	4	5	-
dk17(10)	1	1	6	6	6	-
dk27(0)	1	0	7	6	7	-
dk27(1)	1	0	7	6	7	-
dk27(2)	0	0	5	5	6	-
dk27(3)	0	0	6	7	6	-
dk27(4)	0	0	7	5	7	-
dk27(5)	0	0	6	7	6	-
dk27(6)	0	0	6	7	6	-
dk27(7)	0	0	6	5	6	-
dk27(8)	1	1	8	8	8	-
dk48(0)	0	0	12	13	12	-
dk48(1)	0	0	12	13	12	-
dk48(2)	0	0	12	9	12	-
dk48(3)	0	0	12	11	12	-
dk48(4)	0	0	11	9	11	-
dk48(5)	0	0	12	13	12	-
dk48(6)	0	0	11	11	11	-
dk48(7)	0	0	12	13	12	-
dk48(8)	0	0	11	13	11	-
dk48(9)	1	0	12	12	12	-
dk48(10)	0	0	12	13	12	-
dk48(11)	0	0	12	13	12	-
dk48(12)	0	0	12	13	12	-
dk48(13)	0	0	12	13	12	-
dk48(14)	0	0	10	11	11	-
dk48(15)	2	1	12	13	12	-
dk48(16)	0	0	10	10	10	-
ex1010(0)	0	0	0	0	0	0
ex1010(1)	0	0	0	0	0	0
ex1010(2)	0	0	0	0	0	0
ex1010(3)	0	0	0	0	0	0
ex1010(4)	0	0	0	0	0	0
ex1010(5)	0	0	0	0	0	0
ex1010(6)	0	0	0	0	0	0
ex1010(7)	0	0	0	0	0	0

ex1010(8)	0	0	0	0	0	0
ex1010(9)	0	0	0	0	0	0
exam(0)	0	1	4	3	4	-
exam(1)	0	10	10	10	10	-
exam(2)	0	0	2	3	2	-
exam(3)	0	0	2	2	2	-
exam(4)	0	0	3	3	3	-
exam(5)	0	0	3	2	3	-
exam(6)	0	0	3	3	3	-
exam(7)	0	0	2	2	3	-
exam(8)	10	0	10	10	10	-
exam(9)	0	0	2	2	2	-
exp(0)	0	0	3	3	3	3
exp(1)	0	0	0	0	0	0
exp(2)	0	0	0	0	0	0
exp(3)	8	0	8	8	8	8
exp(4)	8	0	8	8	8	8
exp(5)	0	0	0	0	0	0
exp(6)	1	0	1	1	1	1
exp(7)	0	0	0	0	0	0
exp(8)	0	0	0	0	0	0
exp(9)	0	0	0	0	0	0
exp(10)	0	0	1	1	1	1
exp(11)	0	0	1	1	1	1
exp(12)	2	0	2	2	2	2
exp(13)	1	0	2	2	2	2
exp(14)	0	0	0	0	0	0
exp(15)	0	0	0	0	0	0
exp(16)	0	0	0	0	0	0
exp(17)	0	0	0	0	0	0
exps(0)	0	0	0	0	0	0
exps(1)	0	0	0	0	0	0
exps(2)	0	0	0	0	0	0
exps(3)	0	0	0	0	0	0
exps(4)	0	0	0	0	0	0
exps(5)	1	1	1	1	1	1
exps(6)	1	1	1	1	1	1
exps(7)	0	0	0	0	0	0
exps(8)	0	0	0	0	0	0
exps(9)	0	0	0	0	0	0
exps(10)	0	0	0	0	0	0
exps(11)	0	0	0	0	0	0
exps(12)	0	0	0	0	0	0
exps(13)	0	0	0	0	0	0

exps(14)	0	0	0	0	0	0
exps(15)	0	0	0	0	0	0
exps(16)	1	1	1	1	1	1
exps(17)	0	0	0	0	0	0
exps(18)	1	1	1	1	1	1
exps(19)	1	1	1	1	1	1
exps(20)	0	0	0	0	0	0
exps(21)	0	0	0	0	0	0
exps(22)	0	0	0	0	0	0
exps(23)	0	0	0	0	0	0
exps(24)	0	0	0	0	0	0
exps(25)	0	0	0	0	0	0
exps(26)	0	0	0	0	0	0
exps(27)	0	0	0	0	0	0
exps(28)	0	0	0	0	0	0
exps(29)	0	0	0	0	0	0
exps(30)	0	0	3	1	3	-
exps(31)	1	0	3	1	3	3
exps(32)	1	0	4	3	4	4
exps(33)	0	0	0	0	0	0
exps(34)	0	0	0	0	0	0
exps(35)	0	0	0	0	0	0
exps(36)	0	0	0	0	0	0
exps(37)	0	0	0	0	0	0
fout(0)	0	0	0	0	0	0
fout(1)	0	0	0	0	0	0
fout(2)	0	0	0	0	0	0
fout(3)	0	0	0	0	0	0
fout(4)	0	0	1	1	1	1
fout(5)	0	0	0	0	0	0
fout(6)	0	0	0	0	0	0
fout(7)	0	0	0	0	0	0
fout(8)	0	0	0	0	0	0
fout(9)	0	0	0	0	0	0
inc(0)	1	1	1	1	1	1
inc(1)	1	1	1	1	1	1
inc(2)	0	0	0	0	0	0
inc(3)	0	0	0	0	0	0
inc(4)	0	1	4	4	4	4
inc(5)	1	1	3	3	3	3
inc(6)	0	1	4	4	4	4
inc(7)	1	0	1	1	1	1
inc(8)	3	3	3	3	3	3
l8err(0)	8	8	8	8	8	8

l8err(1)	8	8	8	8	8	8
l8err(2)	8	8	8	8	8	8
l8err(3)	0	0	0	0	0	0
l8err(4)	0	0	0	0	0	0
l8err(5)	0	0	0	0	0	0
l8err(6)	0	0	0	0	0	0
l8err(7)	0	8	8	8	8	8
p1(0)	0	0	6	7	6	7
p1(1)	0	0	2	1	2	2
p1(2)	0	0	2	1	2	3
p1(3)	8	0	8	8	8	8
p1(4)	8	0	8	8	8	8
p1(5)	0	0	2	1	2	2
p1(6)	1	0	2	1	2	2
p1(7)	0	0	2	1	3	3
p1(8)	0	0	1	1	1	1
p1(9)	0	0	3	2	3	3
p1(10)	0	0	3	2	3	3
p1(11)	0	0	3	2	3	3
p1(12)	2	0	5	6	5	6
p1(13)	1	0	6	6	6	6
p1(14)	0	0	1	1	1	1
p1(15)	0	0	1	1	1	1
p1(16)	0	0	2	1	2	3
p1(17)	0	0	1	1	1	1
p3(0)	0	0	6	7	6	7
p3(1)	0	0	2	1	2	2
p3(2)	0	0	2	1	2	3
p3(3)	8	0	8	8	8	8
p3(4)	8	0	8	8	8	8
p3(5)	0	0	2	1	2	2
p3(6)	1	0	2	1	2	2
p3(7)	0	0	2	1	3	3
p3(8)	0	0	1	1	1	1
p3(9)	0	0	3	2	3	3
p3(10)	0	0	3	2	3	3
p3(11)	0	0	3	2	3	3
p3(12)	2	0	5	6	5	6
p3(13)	1	0	6	6	6	6
pdc(0)	1	0	12	12	12	-
pdc(1)	1	0	12	12	12	-
pdc(2)	0	0	11	11	11	-
pdc(3)	0	0	4	4	4	-
pdc(4)	0	0	5	5	5	-

pdc(5)	0	0	7	5	7	-
pdc(6)	13	0	14	14	14	-
pdc(7)	0	0	11	11	11	-
pdc(8)	0	0	5	5	5	-
pdc(9)	0	0	6	6	6	-
pdc(10)	3	0	12	10	12	-
pdc(11)	4	0	11	11	12	-
pdc(12)	8	0	11	11	11	-
pdc(13)	8	0	12	11	12	-
pdc(14)	14	0	14	14	14	-
pdc(15)	13	0	13	13	13	-
pdc(16)	0	0	13	13	13	-
pdc(17)	0	0	13	13	13	-
pdc(18)	0	0	13	13	13	-
pdc(19)	0	0	13	13	13	-
pdc(20)	4	0	9	11	9	-
pdc(21)	7	0	12	12	12	-
pdc(22)	4	0	12	10	12	-
pdc(23)	0	0	7	7	7	-
pdc(24)	0	0	7	7	8	-
pdc(25)	9	0	12	11	12	-
pdc(26)	2	0	12	11	12	-
pdc(27)	8	0	12	12	12	-
pdc(28)	0	0	6	7	7	-
pdc(29)	0	0	7	7	7	-
pdc(30)	1	0	10	9	10	-
pdc(31)	5	0	8	6	9	-
pdc(32)	8	0	10	10	10	-
pdc(33)	0	0	6	7	6	-
pdc(34)	0	0	7	6	7	-
pdc(35)	0	0	7	6	7	-
pdc(36)	0	0	6	6	6	-
pdc(37)	0	0	9	10	10	-
pdc(38)	9	0	12	11	12	-
pdc(39)	9	0	12	11	12	-
spla(0)	0	0	0	0	0	-
spla(1)	0	0	0	0	0	-
spla(2)	1	1	2	1	2	-
spla(3)	1	1	2	2	2	-
spla(4)	1	1	2	2	2	-
spla(5)	0	0	1	1	1	-
spla(6)	0	0	1	1	1	-
spla(7)	0	0	2	2	2	-
spla(8)	13	13	13	13	13	-

spla(9)	0	0	2	2	2	-
spla(10)	0	0	1	1	1	-
spla(11)	0	0	1	1	1	-
spla(12)	0	0	2	2	2	-
spla(13)	7	7	9	8	9	-
spla(14)	8	8	10	10	10	-
spla(15)	5	5	7	6	7	-
spla(16)	4	4	6	6	6	-
spla(17)	8	8	10	10	10	-
spla(18)	8	8	10	10	10	-
spla(19)	0	0	1	1	1	-
spla(20)	0	0	1	1	1	-
spla(21)	0	0	2	1	2	-
spla(22)	0	0	2	1	2	-
spla(23)	1	1	2	1	2	-
spla(24)	9	9	11	11	11	-
spla(25)	9	9	11	11	11	-
spla(26)	0	0	1	1	1	-
spla(27)	0	0	1	1	1	-
spla(28)	0	0	1	1	1	-
spla(29)	0	0	2	1	2	-
spla(30)	0	0	1	1	1	-
spla(31)	8	8	9	9	9	-
spla(32)	0	0	1	1	1	-
spla(33)	0	0	1	1	1	-
spla(34)	9	9	11	11	11	-
spla(35)	0	0	2	2	2	-
spla(36)	0	0	2	2	2	-
spla(37)	0	0	2	2	2	-
spla(38)	0	0	2	2	2	-
spla(39)	3	3	5	4	5	-
spla(40)	0	0	2	1	2	-
spla(41)	4	4	5	4	5	-
spla(42)	8	8	10	8	10	-
spla(43)	5	5	7	6	7	-
spla(44)	4	4	6	6	6	-
spla(45)	0	0	1	1	1	-
t2(0)	5	5	5	5	5	-
t2(1)	5	5	5	5	5	-
t2(2)	6	6	6	6	6	-
t2(3)	8	8	8	8	8	-
t2(4)	10	10	10	10	10	-
t2(5)	6	6	6	6	6	-
t2(6)	4	4	7	6	7	-

t2(7)	7	7	9	7	9	-
t2(8)	11	10	12	12	14	-
t2(9)	10	10	11	10	11	-
t2(10)	11	11	12	12	12	-
t2(11)	11	11	12	12	12	-
t2(12)	12	10	13	12	13	-
t2(13)	11	9	12	11	12	-
t2(14)	11	9	12	11	12	-
t2(15)	10	10	12	12	12	-
t4(0)	3	3	6	6	6	-
t4(1)	1	1	6	5	6	-
t4(2)	3	3	8	7	8	-
t4(3)	1	1	6	6	6	-
t4(4)	1	1	5	6	5	-
t4(5)	3	3	8	8	8	-
t4(6)	3	3	8	8	8	-
t4(7)	3	3	8	10	8	-
test1(0)	0	0	0	0	0	0
test1(1)	0	0	0	0	0	0
test1(2)	0	0	0	0	0	0
test1(3)	0	0	0	0	0	0
test1(4)	0	0	0	0	0	0
test1(5)	0	0	0	0	0	0
test1(6)	0	0	0	0	0	0
test1(7)	0	0	0	0	0	0
test1(8)	0	0	0	0	0	0
test1(9)	0	0	0	0	0	0
test3(0)	0	0	0	0	0	0
test3(1)	0	0	0	0	0	0
test3(2)	0	0	0	0	0	0
test3(3)	0	0	0	0	0	0
test3(4)	0	0	0	0	0	0
test3(5)	0	0	0	0	0	0
test3(6)	0	0	0	0	0	0
test3(7)	0	0	0	0	0	0
test3(8)	0	0	0	0	0	0
test3(9)	0	0	0	0	0	0
test3(10)	0	0	0	0	0	0
test3(11)	0	0	0	0	0	0
test3(12)	0	0	0	0	0	0
test3(13)	0	0	0	0	0	0
test3(14)	0	0	0	0	0	0
test3(15)	0	0	0	0	0	0
test3(16)	0	0	0	0	0	0

test3(17)	0	0	0	0	0	0
test3(18)	0	0	0	0	0	0
test3(19)	0	0	0	0	0	0
test3(20)	0	0	0	0	0	0
test3(21)	0	0	0	0	0	0
test3(22)	0	0	0	0	0	0
test3(23)	0	0	0	0	0	0
test3(24)	0	0	0	0	0	0
test3(25)	0	0	0	0	0	0
test3(26)	0	0	0	0	0	0
test3(27)	0	0	0	0	0	0
test3(28)	0	0	0	0	0	0
test3(29)	0	0	0	0	0	0
test3(30)	0	0	0	0	0	0
test3(31)	0	0	0	0	0	0
test3(32)	0	0	0	0	0	0
test3(33)	0	0	0	0	0	0
test3(34)	0	0	0	0	0	0
test4(0)	0	0	0	0	0	0
test4(1)	0	0	0	0	0	0
test4(2)	0	0	1	1	1	1
test4(3)	0	0	0	0	0	0
test4(4)	0	0	1	1	1	1
test4(5)	0	0	1	1	1	1
test4(6)	0	0	1	1	1	1
test4(7)	0	0	1	1	1	1
test4(8)	0	0	1	1	1	1
test4(9)	0	0	1	1	1	1
test4(10)	0	0	0	0	0	0
test4(11)	0	0	1	1	1	1
test4(12)	0	0	1	1	1	1
test4(13)	0	0	1	1	1	1
test4(14)	0	0	1	2	1	2
test4(15)	0	0	0	0	0	0
test4(16)	0	0	0	0	0	0
test4(17)	0	0	1	1	1	1
test4(18)	0	0	1	1	1	2
test4(19)	0	0	0	0	0	0
test4(20)	0	0	1	1	1	1
test4(21)	0	0	1	1	1	1
test4(22)	0	0	1	1	1	1
test4(23)	0	0	1	1	1	1
test4(24)	0	0	0	0	0	0
test4(25)	0	0	1	1	1	1

test4(26)	0	0	1	1	1	1
test4(27)	0	0	0	0	0	0
test4(28)	0	0	0	0	0	0
test4(29)	0	0	0	0	0	0
x1dn(0)	11	3	11	11	11	11
x1dn(1)	9	3	9	9	9	9
x1dn(2)	9	3	13	13	13	-
x1dn(3)	13	3	16	16	16	-
x1dn(4)	13	3	16	16	16	-
x1dn(5)	0	3	10	13	12	-
wim(0)	0	0	1	1	1	1
wim(1)	0	0	1	1	1	1
wim(2)	0	1	2	1	2	2
wim(3)	0	0	1	1	1	1
wim(4)	1	1	2	1	2	2
wim(5)	0	0	1	1	1	1
wim(6)	0	0	1	1	1	1

	Euristica di esp. di L_0			Euristica di contrazione			Algo. enumerativo esatto		
Funzione, n, m	k	Max	$Time$	k	Max	$Time$	k	Max	$Time$
alu2, 10, 8	3.375	7	4.177	3.375	7	4.909	3.375	7	5.027
alu3, 10, 8	2.75	6	3.288	2.75	6	3.451	2.75	6	3.396
apla, 10, 12	4.417	6	41.733	3.25	6	53.136	-	-	-
b10, 15, 11	2.636	15	10.006	2.636	15	11.351	2.636	15	54.84
bcc, 26, 45	10.755	14	59.558	10.755	14	53.864	10.755	14	60.919
bcd.div3, 4, 4	1.5	2	0.203	1.5	2	0.792	1.5	2	0.903
bench1, 9, 9	0	0	24.614	0	0	21.115	0	0	23.001
bench, 6, 8	1.875	3	1.937	1.75	3	3.759	2	3	210.973
dekoder, 4, 7	1.285	2	0.403	1	1	1.523	1.285	2	1.489
dk17, 10, 11	5.727	7	21.873	4.909	7	37.886	-	-	-
dk27, 9, 9	6.444	8	5.436	6.222	8	12.677	-	-	-
dk48, 15, 17	11.588	12	131.919	11.941	13	10212.934	-	-	-
ex1010, 10, 10	0	0	105.573	0	0	63.166	0	0	73.663
exam, 10, 10	4.1	10	28.322	4	10	47.231	-	-	-
exp, 8, 18	1.444	8	2.101	1.444	8	6.809	1.444	8	7.699
exps, 8, 38	0.395	4	2.49	0.263	3	13.52	-	-	-
fout, 6, 10	0.1	1	0.26	0.1	1	3.208	0.1	1	2.999
inc, 7, 9	1.888	4	0.778	1.888	4	2.415	1.888	4	2.179
l8err, 8, 8	4	8	0.432	4	8	2.006	4	8	1.826
p1, 8, 18	3.222	8	15.285	2.883	8	18.098	3.5	8	90018
p3, 8, 14	3.786	8	11.309	3.375	8	13.451	4.071	8	87273
pdc, 16, 40	9.875	14	757.339	9.6	14	5976.43	-	-	-
spla, 16, 46	3.978	13	27899.712	3.695	13	7807.671	-	-	-
t2, 17, 16	9.5	13	22.802	9.062	12	1035.178	-	-	-
t4, 12, 8	6.875	8	19.487	7	10	35.667	-	-	-
test1, 8, 10	0	0	13.05	0	0	11.126	0	0	9.521
test3, 10, 35	0	0	343.207	0	0	241.442	0	0	222.927
test4, 8, 30	0.633	1	19.609	0.666	2	28.654	0.7	2	34.74
x1dn, 27, 6	12.5	16	846.334	13	16	136.116	-	-	-
wim, 4, 7	1.285	2	0.361	1	1	1.35	1.285	2	2.591

Tabella A.2 – Risultati sperimentali ottenuti tramite l'esecuzione di tre algoritmi.

	All zeros test			All ones test			Euristica di esp. di L_0 e L_1		
Funzione, n, m	k	Max	$Time$	k	Max	$Time$	k	Max	$Time$
alu2, 10, 8	2.750	6	1.805	1.125	6	2.513	3.375	7	13.702
alu3, 10, 8	2.75	6	1.095	0.625	2	2.836	2.750	6	9.439
apla, 10, 12	0.166	1	1.439	0.083	1	2.713	4.500	6	77.206
b10, 15, 11	2.545	15	5.534	1.181	3	6.710	2.636	15	25.076
bcc, 26, 45	10.755	14	36.243	10.755	14	39.360	10.755	14	143.765
bcd.div3, 4, 4	0.500	2	0.130	0.250	1	0.130	1.500	2	0.432
bench1, 9, 9	0	0	8.855	0	0	19.351	0	0	67.384
bench, 6, 8	0	0	0.686	0	0	0.875	1.875	3	3.501
dekoder, 4, 7	0	0	0.252	0.285	1	0.133	1.285	2	0.756
dk17, 10, 11	0.090	1	0.877	0.090	1	1.029	5.727	7	41.845
dk27, 9, 9	0.333	1	0.468	0.111	1	0.503	6.555	8	10.377
dk48, 15, 17	0.176	2	1.620	0.058	1	1.805	11.647	12	237.318
ex1010, 10, 10	0	0	44.253	0	0	44.957	0	0	265.113
exam, 10, 10	1	10	3.342	1.100	10	4.545	4.200	10	56.728
exp, 8, 18	1.111	8	1.605	0	0	3.232	1.444	8	12.900
exps, 8, 38	0.184	1	5.946	0.131	1	7.379	0.395	4	28.519
fout, 6, 10	0	0	1.135	0	0	0.923	0.100	1	4.366
inc, 7, 9	0.777	3	0.496	0.888	3	0.524	1.888	4	1.983
l8err, 8, 8	3	8	0.884	4	8	0.501	4.000	8	2.720
p1, 8, 18	1.111	8	1.624	0	0	3.452	3.277	8	28.023
p3, 8, 14	1.428	8	1.153	0	0	1.625	3.857	8	20.813
pdc, 16, 40	3.275	14	10.387	0	0	22.038	10.000	14	1724.054
spla, 16, 46	2.521	13	9.754	2.521	13	8.678	3.978	13	51957.550
t2, 17, 16	8.625	12	1.213	8.187	11	1.248	9.625	14	41.546
t4, 12, 8	2.25	3	0.810	2.250	3	0.900	6.875	8	27.282
test1, 8, 10	0	0	4.346	0	0	9.305	0	0	32.098
test3, 10, 35	0	0	104.855	0	0	227.414	0	0	894.880
test4, 8, 30	0	0	19.946	0	0	8.826	0.633	1	78.493
x1dn, 27, 6	9.166	13	2.894	3	3	5.923	12.833	16	1091.619
wim, 4, 7	0.142	1	0.14	0.285	1	0.125	1.285	2	0.789

Tabella A.3 – Risultati sperimentali ottenuti tramite l'esecuzione di tre algoritmi.

Bibliografia

[1] F. Luccio, A. Bernasconi, V. Ciriani e L. Pagli. “Three-Level Logic Minimization Based on Function Regularities”. In: *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 22, no. 8, agosto 2003, pag. 1005-1016

[2] S. A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *ACM Symposium on Theory of Computing* 1971, pp. 151–158.

[3] H. R. Andersen. "An Introduction to Binary Decision Diagrams". The IT University of Copenhagen, 1999.

[4] B. Bollig, M. Sauerhoff, D. Sieling e I. Wegener. "Binary Decision Diagrams". FB Informatik, LS2, Univ. Dortmund.

[5] F. Luccio, A. Bernasconi, V. Ciriani e L. Pagli. “Exploiting Regularities for Boolean Function Synthesis”. Theory of Computing Systems 39, 485–501 (2006).

[6] R. E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: IEEE Transactions on Computers 35 (1986), pp. 667–691.

[7] V. Ciriani. “Synthesis of SPP Three-Level Logic Networks using Affine Spaces”. In: IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 22, no. 10, pp. 1310–1323, 2003.

[8] F. Luccio, L. Pagli. “On a New Boolean Function with Applications”. In: IEEE Transactions on Computers, vol. 48, no. 3, pp. 296–310, 1999.

[9] Liebler R. 2003. "Basic Matrix Algebra with Algorithms and Applications". Chapman & Hall/CRC.

[10] A. Bernasconi, V. Ciriani. "Dimension-reducible Boolean functions based on affine spaces". ACM Trans. Des. Autom. Electron. Syst. 16, 2, Articolo 13 (Marzo 2011).

[11] A. Bernasconi, V. Ciriani, F. Luccio e L. Pagli. “Fast Three-Level Logic Minimization Based on Autosymmetry”. In: ACM/IEEE 39th Design Automation Conference (DAC), 2002, pp. 425–430.

[12] V. Kravets e K. Sakallah. “Generalized Symmetries of Boolean Functions”. In: International Conference on Computer-Aided Design (ICCAD), 2000, pp. 526–532.

[13] V. N. Kravets e K. A. Sakallah. “Constructive library-aware synthesis using symmetries”. In: 2000 Design, Automation and Test in Europe (DATE). IEEE Computer Society / ACM, 2000, pp. 208–213.

[14] J. Goldman, G. Rota. "The Number of Subspaces of a Vector Space". Defense Technical Information Center, 1969.

[15] D. E. Knuth. “Subspaces, subsets, and partitions”. Journal of Combinatorial Theory, Series A, vol. 10, no. 2, pp. 178 – 180, 1971.

[16] S. Yang. "Logic Synthesis and Optimization Benchmarks User Guide". Microelectonic Center of North Carolina. 1991.

[17] F. Somenzi. "CUDD: CU Decision Diagram Package Release 2.7.0". Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder. 2018.