UNIVERSITÀ DI PISA

CORSO DI LAUREA IN INFORMATICA

TESI DI LAUREA

SINTESI LOGICA APPROSSIMATA DI FORME SOP

Relatore:
Prof.ssa Anna Bernasconi

Candidato:
Marco Costa

Anno Accademico 2018-2019

Per Aron.

Indice

Introduzione

All'interno delle ricerche nell'ambito della fabbricazione e progettazione di circuiti integrati, svolgono un ruolo centrale i processi di ottimizzazione e minimizzazione di rappresentazioni Booleane. Data una funzione Booleana, sintetizzarne la rappresentazione con minore complessità è indice di numerosi fattori tra cui, ad esempio, la riduzione del costo di produzione o, addirittura, la fattibilità di produzione stessa. Diviene centrale, quindi, studiare e ricercare rappresentazioni equivalenti di funzioni Booleane che permettano di fabbricare circuiti con la minore complessità possibile. In questa tesi, per realizzare una rappresentazione efficiente sfrutteremo la tollerabilità di errore concessa per alcuni ambiti di impiego, come, ad esempio, le applicazioni multimediali, per sintetizzare funzioni approssimate ottenute a partire dalla funzione designata. Sintetizzare un circuito approssimato, garantisce sì l'affidabilità del risultato unicamente all'interno di una determinata soglia di errore, ma permette di sfruttare la soglia designata per ridurre la complessità iniziale del circuito. Tuttavia, come verrà chiarito più avanti, la ricerca di approssimazioni ottime di funzioni Booleane all'interno di una determinata soglia di errore, è un problema che, al caso pessimo, prevede un tempo esponenziale nel numero di input; per questo motivo sono state sviluppate tecniche euristiche che risultano generalmente più *veloci*, senza però garantire l'ottimalità del risultato. L'algoritmo di approssimazione che proponiamo è basato su un'euristica denominata *espansione assistita* [12].

Nel corso dell'elaborato ci concentreremo unicamente su un particolare tipo di forma di rappresentazione di funzioni Booleane: le forme SOP, o *somme di prodotti* (per altre forme sono stati pubblicati algoritmi basati sulla medesima euristica, v. Bernasconi, Ciriani (2014)). Impiegando varie soglie di errore saranno effettuati test su un insieme standard di funzioni; i risultati finali mostreranno che, ad esempio, introducendo una soglia dell'1% si ottengono approssimazioni con un guadagno medio superiore al 13% nel numero di letterali della rappresentazione.

Infine, utilizzando una tecnica denominata *bi-decomposizione*, costruiremo una funzione che, a partire dalla funzione approssimata ottenuta mediante euristica e da un operatore logico, corregga gli errori introdotti con l'approssimazione e fornisca nuovamente la funzione originale, in una rappresentazione possibilmente più compatta. La funzione così costruita può essere vista come un "fattore di correzione" della approssimazione computata.

La tesi è strutturata come segue: nel Capitolo 1 sono descritti i concetti principali sui quali è imperniato l'elaborato; nel secondo Capitolo si propone un algoritmo di sintesi logica approssimata, la sua implementazione ed i risultati ottenuti; infine, nel Capitolo 3 utilizzeremo le tecniche di bi-decomposizione per ricostruire la funzione originale a partire dall'approssimazione computata; l'ultima fase di test prevede di confrontare la complessità della nuova rappresentazione, ottenuta mediante bi-decomposizione, con la rappresentazione originale.

Capitolo 1

Funzioni Booleane e minimizzazione logica

Al fine di comprendere l'elaborato è opportuno definire i concetti di *funzione Booleana* e *minimizzazione logica*.

Una funzione Booleana è una funzione della forma

$$f : \{0,1\}^n \mapsto \{0,1,*\}^m$$

dove n è il numero di input della funzione ed m il numero di output. Se $m = 1$ la funzione è detta *a singolo-output*, altrimenti è denominata multi-output.
Poiché è possibile considerare una funzione multi-output come m funzioni a singolo output ci riferiremo unicamente a queste ultime; per indicare la funzione legata alla k-esima uscita useremo la notazione f_k.
L'elemento del codominio "$*$" è detto *condizione di indifferenza* (o *don't care*), in quanto è legato a valori di input per i quali risulta irrilevante l'output.

Sia $f : \{0,1\}^n \mapsto \{0,1,*\}$ una funzione Booleana in n variabili, valgono le seguenti definizioni:

Definizione 1.1. (Prodotto) Un AND di $1 \leq k \leq n$ variabili distinte è detto *prodotto* (o cubo).

Notazione: indichiamo con xy l'AND tra due variabili x ed y. Si estende naturalmente nel caso di più variabili.

Definizione 1.2. (Mintermine) Un prodotto nel quale ognuna delle n variabili appare esattamente una volta, in forma complementata o meno, è detto mintermine.

Definizione 1.3. (On-set) L'On-set di una funzione f è l'insieme $f^{-1}(1)$. Ci riferiremo ad esso come f^{on}.

Definizione 1.4. (Off-set) L'Off-set di una funzione f è l'insieme $f^{-1}(0)$. Ci riferiremo ad esso come f^{off}.

Definizione 1.5. (DC-set) Il Don't Care set (DC-set) di una funzione f è l'insieme $f^{-1}(*)$. Ci riferiremo ad esso come f^{dc}.

Osservazione. È possibile definire una funzione Booleana utilizzando unicamente due dei tre insiemi soprastanti: dati due insiemi, il terzo è ottenibile mediante differenza insiemistica tra il dominio della funzione e l'unione degli stessi.

Indichiamo l'unione di questi elementi mediante la giustapposizione dei simboli indicanti l'insieme, separati da una virgola: ad esempio l'unione tra l'On-set e il DC-set sarà indicata come $f^{on,dc}$.

Definizione 1.6. (Funzione totale) Una funzione Booleana è detta *completamente specificata* o *totale* se il suo DC-set è vuoto.

Definizione 1.7. (SOP) Una funzione Booleana è una somma-di-prodotti (SOP), o funzione in forma normale disgiuntiva, se è formata da un primo livello di porte AND ed un secondo livello di porte OR. Una funzione in forma SOP è *totale*.

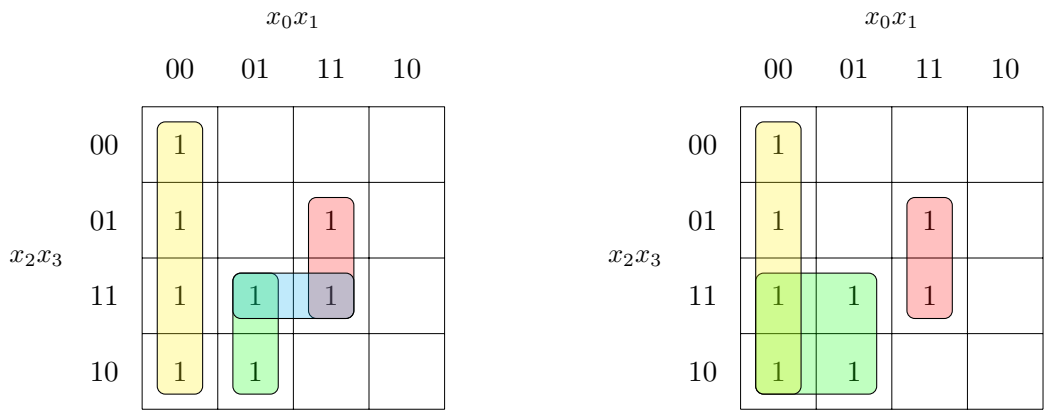
L'insieme di operatori logici {AND, OR, NOT} è *funzionalmente completo*, ovvero può essere usato per rappresentare ogni funzione Booleana. Di conseguenza *ogni* funzione Booleana può essere rappresentata in forma SOP.

Notazione: indichiamo $x + y$ l'OR di due prodotti x ed y. Si estende naturalmente nel caso di più variabili.

Esempio 1. La funzione $f = \overline{x_0}\overline{x_1} + x_0x_1x_3 + \overline{x_0}x_1x_2 + x_1x_2x_3$ è una SOP.

Osservazione. È importante notare come ogni funzione (Booleana e non) possa essere rappresentata in modi diversi e, poiché questo è un aspetto primario nella sintesi logica, distingueremo il concetto di *funzione* e *rappresentazione* (o copertura, in un senso che sarà più chiaro in seguito).

1.1 Mappe di Karnaugh



(a) $f = \overline{x_0}\overline{x_1} + x_0x_1x_3 + \overline{x_0}x_1x_2 + x_1x_2x_3$

(b) $f = \overline{x_0}\overline{x_1} + x_0x_1x_3 + \overline{x_0}x_2$

Figura 1.1: Due mappe di Karnaugh che rappresentano la stessa funzione. La rappresentazione (b) fornisce una riduzione di 4 letterali ed un prodotto rispetto alla rappresentazione (a)

Le mappe di Karnaugh sono un metodo per la rappresentazione di funzioni Booleane. Sono formate da una matrice di 2^n posizioni, dove n è il numero di input della funzione ed ogni posizione della tabella rappresenta un mintermine ed il contenuto il relativo output. Per leggibilità ometteremo l'Off-set della funzione.
Ogni dimensione della matrice rappresenta una coppia di variabili di input; le istanze delle variabili sono ordinate affinché la distanza di Hamming[1] risulti uguale ad 1.
I prodotti sono rappresentati mediante rettangoli contigui con area sempre pari ad una potenza di 2, infatti in una mappa di Karnaugh con n variabili di input, un prodotto di k letterali è formato da un rettangolo di 2^{n-k} caselle.

In Figura 1.1 ne è mostrato un esempio.

[1]La distanza di Hamming tra due stringhe con la stessa lunghezza è il numero di posizioni nelle quali i simboli corrispondenti risultano diversi.

1.2 Minimizzazione logica a due livelli

L'operazione di minimizzazione logica a due livelli consiste nel trovare una SOP g, minima rispetto ad una determinata metrica di costo, che *copra* una determinata funzione Booleana f [7], dove il predicato *copre* è così definito:

Definizione 1.8. (Copertura di una funzione Booleana)
Siano g ed f due funzioni Booleane, diciamo che g *è una copertura per* (o copre) f **sse**

$$f^{on} \subseteq g^{on} \subseteq f^{on,dc}$$

In altre parole, si richiede che l'ON-set di g sia *almeno grande quanto* l'ON-set di f ma *non più grande* dell'unione tra l'ON-set di f ed il suo DC-set.

Quest'ultima affermazione è molto importante ed è un punto chiave nel processo di minimizzazione: il DC-set della funzione originale, a causa della totalità della funzione g, deve subire un assegnamento in $\{0, 1\}$, la scelta di questo assegnamento è determinante nel processo di minimizzazione.

Definizione 1.9. (Implicante ed implicante primo)
Un prodotto p è detto *implicante* di f **sse** $p \implies f$.
Se non esiste altro implicante appartenente ad f che a sua volta implichi p allora p è detto *implicante primo* (PI); ovvero il prodotto non può essere coperto da uno più generico (con meno letterali). Una copertura contenente unicamente PI è detta *prima*.

Dato un insieme X, indicheremo $PI(X)$ come l'insieme di implicanti primi in X.

Definizione 1.10. (EPI) Un PI che sia l'unico a coprire *un qualche* elemento di f^{on} è detto *implicante primo essenziale* (EPI) per f.

Esempio 2. Nella funzione dell'Esempio 1 e Figura 1.1, $x_0x_1\overline{x_2}x_3$ e $x_1x_2x_3$ sono alcuni degli implicanti di f; inoltre il primo non è un PI (essendo implicato da $x_0x_1x_3$) mentre $x_1x_2x_3$ è un implicante primo *non* essenziale. Uno degli EPI della funzione è $\overline{x_0}\overline{x_1}$ poiché è l'unico a coprire il mintermine $x_0x_1\overline{x_2}x_3$.

Definizione 1.11. (Copertura irridondante) Una copertura P di una funzione f è *irridondante* **sse** $\nexists C \subset P \,.\, C$ *copre* f.

Definizione 1.12. (Copertura minima (locale)) Una copertura P di una funzione f è minima (locale) **sse** è *prima* e *irridondante*.

Definizione 1.13. (Copertura minimale) Una copertura minima P di una funzione f è minimale **sse** per ogni copertura minima P' di f, P è la copertura col minor numero di letterali e prodotti.

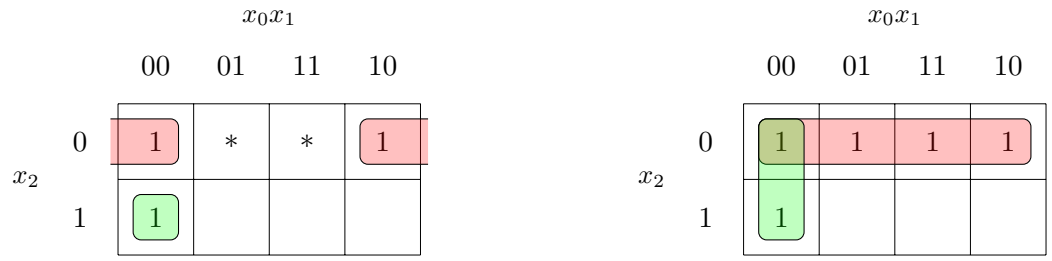
Esempio 3. In Figura 1.1 la copertura (a) è prima ma non irridondante ($P \setminus \{\overline{x_0}x_1x_2\}$ *copre* f). La copertura (b), invece, rappresenta la copertura *minimale* per f.

I minimizzatori logici si dividono in due classi: esatti ed euristici. I minimizzatori esatti ricercano coperture minime (globali), solitamente mediante l'uso dell'algoritmo di Quine-McCluskey [10, 9]; l'algoritmo si divide in due fasi: enumerazione di $PI(f^{on,dc})$ e utilizzo degli implicanti primi per la risoluzione del problema di set covering[2] $\langle f^{on}, PI(f^{1*})\rangle$ associato; poiché il problema di set covering è **NP-completo** lo è anche l'algoritmo di Quine-McCluskey.

Per funzioni con un numero di input elevato risulta necessario affidarsi ad elaborazioni euristiche che restituiscano coperture minime (locali) della funzione desiderata. In molte applicazioni, infatti, una soluzione ben approssimata è sufficiente.

[2]Sia Z un insieme, $X \subseteq Z$ e $Y \subseteq 2^Z$, diciamo che y *copre* x quando $x \in y$. Il problema di set covering $\langle X, Y\rangle$ consiste nel trovare un sottoinsieme S di costo minimo tale che ogni $x \in X$ sia coperto da qualche elemento $y \in S$

1.2.1 Minimizzazione euristica



(a) $f = \overline{x_1}\overline{x_2} + \overline{x_0}\overline{x_1}x_2$; $f^{dc} = \{x_1\overline{x_2}\}$ (b) $f = \overline{x_2} + \overline{x_0}\overline{x_1}$; $f^{dc} = \emptyset$

Figura 1.2: A destra la minimizzazione mediante `Espresso` della funzione (a). Si noti come la scelta dell'assegnamento del DC-set assista il risultato della minimizzazione: se l'assegnamento fosse stato opposto la funzione risultante sarebbe stata $f = \overline{x_1}\overline{x_2} + \overline{x_0}\overline{x_1}$

A differenza della minimizzazione esatta, i minimizzatori euristici effettuano l'iterazione di operazioni basilari sulla funzione. Le operazioni principali sono le seguenti:

- EXPAND sostituisce ogni implicante della funzione con un PI che lo contiene (e che possibilmente contiene altri prodotti o somme di prodotti della funzione).
- IRREDUNDANT rimuove un set massimale di implicanti ridondanti.
- REDUCE sostituisce ogni implicante primo con un implicante con meno letterali che copra tutti i mintermini non contenuti in altri cubi della data rappresentazione.

Il più famoso minimizzatore logico a due livelli è `Espresso` [4]; questo prende come input una funzione Booleana in formato PLA e ne restituisce una minimizzazione totale nel medesimo formato.

A grandi linee l'algoritmo prevede l'applicazione delle operazioni EXPAND e IRREDUNDANT per ottenere una SOP irridondante; da questa vengono estratti gli EPI ed effettuata l'iterazione di REDUCE, EXPAND e IRREDUNDANT finché nessun prodotto possa essere ulteriormente ridotto; infine, vengono applicate differenti euristiche per tentare di ridurre ulteriormente la copertura [7].

Oltre alla minimizzazione euristica, la suite `Espresso` permette di eseguire l'algoritmo di minimizzazione esatta di Quine-McCluskey; inoltre, al suo interno sono contenuti diversi applicativi, tra questi permette di verificare che due distinte PLA rappresentino la stessa funzione Booleana.

Nel seguito, quando ci riferiremo all'operazione di minimizzazione eseguita mediante `Espresso`, ci riferiremo unicamente all'algoritmo euristico.

1.2.2 Il formato PLA

Sia n il numero di ingressi della funzione, m il numero di uscite, un file testuale con estensione ".pla" è valido se rispetta la seguente sintassi (minima) [14]:

1. Le prime due righe del file devono contenere rispettivamente il numero di input e di output della funzione come segue:

```
.i n
.o m
```

2. Una riga per ogni prodotto distinto della funzione; la riga i-esima risulta

$$P_{i,1}P_{i,2}\cdots P_{i,n}\ O_{i,1}O_{i,2}\cdots O_{i,m}$$

e $P_{i,j} \in \{0, 1, -\}$ dove il carattere '1' indica che la j-esima variabile appare non complementata nel prodotto, '0' indica che appare complementata e '-' che non appare; infine $O_{i,j} \in \{0, 1, -\}$ dove il carattere '1' indica che il prodotto i-esimo è parte dell'On-set del j-esimo output, '0' che il prodotto non appare e '-' che è parte del DC-set.

3. *.e* ad indicare la fine del file (opzionale).

Dato che una funzione può essere definita in base a due dei tre insiemi che la compongono, è possibile omettere i prodotti appartenenti all'Off-set della funzione per tutti i suoi output.

Esempio 4. La PLA della funzione dell'esempio 1 risulta:

```
.i 4
.o 1
00-- 1
11-1 1
011- 1
-111 1
```

1.3 BDD

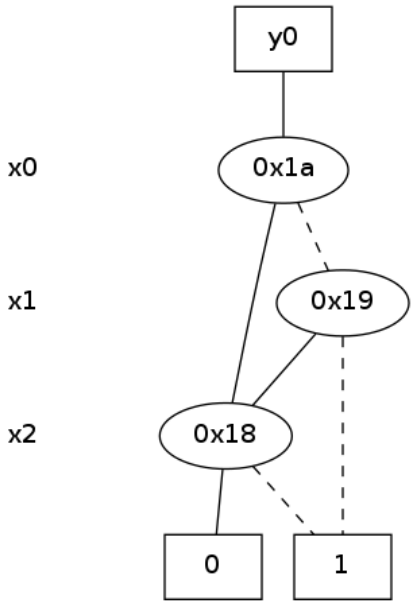


Figura 1.3: BDD per la funzione $f = \overline{x_2} + \overline{x_0}\overline{x_1}$

Un *diagramma di decisione* (DD) su un insieme di variabili Booleane $X = \{x_1, \ldots, x_n\}$ è un grafo diretto ed aciclico, nel quale ogni nodo non terminale è etichettato con una variabile in X e ogni nodo foglia etichettato con un valore in $\{0, 1\}$.
Un Binary Decision Diagram (BDD) è un particolare tipo di DD per la rappresentazione di funzioni Booleane. Il nodo radice rappresenta la funzione f, ogni nodo interno, invece, una delle variabili x_i della funzione; ognuno di essi delinea una decomposizione di Shannon[3]

[3]data una funzione Booleana f con n variabili di input $x_0, x_1, \ldots, x_{n-1}$, vale l'uguaglianza $f(x_0, x_1, \ldots, x_{n-1}) = x_0 \cdot f(1, x_1, \ldots, x_{n-1}) + \overline{x_0} \cdot f(0, x_1, \ldots, x_{n-1})$

sulla propria variabile. Ne segue che ogni nodo interno u detiene due successori: *then(u)* ed $else(u)$, i primi sono rappresentati con archi a linea continua, gli ultimi con archi a linea tratteggiata. I nodi foglia rappresentano le due funzioni costanti 0 ed 1.

Un BDD è *ordinato* se l'ordine dei nodi interni è lo stesso per ogni possibile cammino dal nodo radice ai nodi foglia. Un BDD è *ridotto* se non esiste alcun nodo v tale che $then(v) = else(v)$ e non esistono due nodi distinti che siano radice di sottografi isomorfi. Un BDD contemporaneamente ridotto ed ordinato è detto *ROBDD*.

Lemma. *Per ogni funzione* $f : \{0,1\}^n \mapsto \{0,1\}$ *esiste un unico ROBDD che rappresenta* f.

I ROBDD sono anche detti *BDD in forma canonica* [8]. Nel seguito utilizzeremo il termine "BDD" per indicare un BDD in forma canonica.

Utilizzare le BDD per rappresentare funzioni Booleane permette di implementare in modo efficiente, in relazione alla dimensione del grafo, operazioni logiche come congiunzione, disgiunzione e negazione [3].
La dimensione del grafo è legata sia alla dimensione della funzione rappresentata sia all'ordinamento delle variabili di input: un ordinamento sfavorevole può provocare un incremento esponenziale nella dimensione del grafo rispetto al numero di variabili di input;poiché determinare l'ordinamento ottimo delle variabili è **NP-arduo**, sono state introdotte euristiche efficienti per affrontare il problema [11].

La principale libreria C per la gestione di BDD è CUDD (*Colorado University Decision Diagrams*) [13].

Capitolo 2

Sintesi logica approssimata

Molteplici ambiti di ricerca si focalizzano sulla classificazione post-fabbricazione di circuiti integrati; convenzionalmente, circuiti privi di errori di produzione sono classificati come *perfetti* o, altrimenti, *imperfetti*; storicamente, circuiti classificati come imperfetti venivano giudicati come inadatti e, di conseguenza, scartati. L'idea dietro l'*error tolerance* è che, per molti ambiti di utilizzo come, ad esempio, applicazioni multimediali, circuiti *imperfetti* (all'interno di una determinata soglia di tollerabilità) possono comunque essere utilizzati [5].

A differenza dell'*error tolerance*, l'approccio della sintesi logica approssimata prevede di sfruttare la tollerabilità di errore durante la fase di progettazione e sintesi di un circuito e non successivamente alla fase di produzione. L'intento di questo approccio è produrre approssimazioni "controllate" di circuiti *perfetti* la cui area, intesa come numero di letterali e numero di prodotti, sia inferiore al circuito originale.

Di seguito è proposto un algoritmo euristico per la sintesi logica approssimata di forme SOP basato sull'euristica proposta in [12] e l'algoritmo pubblicato in [1]. L'implementazione è stata realizzata in linguaggio *C*.

2.1 Metriche di errore

Nella letteratura sono state proposte due tipi di metriche per la misurazione dell'errore per la sintesi logica approssimata, *Error magnitude (EM)* ed *Error rate (ER)* [5].

L'*error magnitude*, per un insieme di output, è definito come la quantità massima per la quale il valore numerico alle uscite di un circuito può deviare dal valore esatto. L'*error rate*, invece, è definito come la percentuale di vettori di input per i quali l'output computato dal circuito può variare da quello esatto. Considereremo unicamente la metrica *error rate*.

Per una funzione $f : \{0,1\}^n \mapsto \{0,1,*\}^m$ definiamo la soglia C_t come il *massimo numero di mintermini complementabili*; sia r l'error rate, segue che

$$C_t = r \cdot 2^n$$

2.2 Espansione assistita

Una funzione totale è approssimabile effettuando il complemento dei singoli mintermini dell'ON-set (detti complementi $1 \to 0$) e dell'Off-set (detti complementi $0 \to 1$).

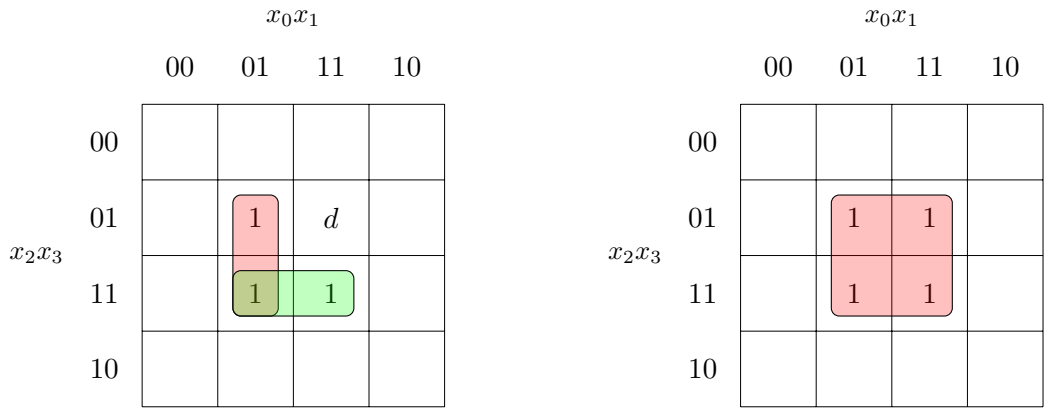
Definizione 2.1. (Guadagno di un'approssimazione) Sia f' la funzione approssimata ed R' la sua rappresentazione, definiamo il *guadagno* ottenuto come la differenza tra il numero di letterali di R' e il numero di letterali della rappresentazione originale[1].

L'analisi esaustiva per la ricerca di complementi ottimali prevede di complementare ogni possibile combinazione di C_t mintermini, sintetizzare il circuito risultante e verificare quale fornisce il guadagno migliore. Questa prevede che, per $C_t = 1$ debbano essere sintetizzati 2^n circuiti, per $C_t = 2$ si ottengano 2^{n^2} nuovi circuiti ed, in generale, una ricerca esaustiva con $C_t = k$ richieda la valutazione di 2^{n^k} distinti circuiti: una complessità inattuabile se non per combinazioni di n e k contenute.

L'algoritmo che proponiamo è basato su un'euristica denominata *espansione assistita*. L'euristica prevede di valutare unicamente i mintermini che impediscono l'espansione di uno o più PI in una determinata direzione; ogni direzione corrisponde ad un'espansione ottenuta eliminando un differente insieme di letterali dall'implicante. Se l'espansione produce un complemento di mintermini *maggiore* di C_t l'espansione viene scartata, altrimenti il prodotto generato viene sottoposto ad una successiva valutazione ed eventualmente aggiunto alla nuova SOP.

Poiché l'euristica prevede unicamente la rimozione di letterali dell'implicante, gli unici mintermini complementabili sono parte dell'Off-set della funzione (complementi $0 \rightarrow 1$).

In Figura 2.1 è mostrato un esempio di espansione assistita.



(a) $f = \overline{x_0}x_1x_3 + x_1x_2x_3$ (b) $f = x_1x_3$

Figura 2.1: Espansione assistita con $C_t = 1$: (a) Il mintermine $x_0x_1\overline{x_2}x_3$ è l'unico ad espandere l'insieme di PI, (b) Espansione effettuata nella direzione x_2 per il PI $x_1x_2x_3$ (o analogamente in direzione x_0 per $\overline{x_0}x_1x_3$).

2.3 Algoritmo

L'algoritmo sfrutta l'euristica dell'espansione assistita mediante rimozione di un singolo letterale dall'implicante primo e ne ottimizza la rappresentazione risultante tramite l'ausilio di tecniche di rimozione di ridondanze e minimizzatori euristici.

La funzione f in ingresso viene minimizzata (ricordiamo che la minimizzazione comporta l'assegnamento di f^{dc}) ottenendo una rappresentazione in forma SOP S di f e, a

[1] Analogamente è possibile utilizzare altre metriche (o combinazioni delle stesse) per valutare il guadagno. Oltre al numero di letterali utilizzeremo anche il numero di prodotti di una data rappresentazione.

partire da questa, si genera un vettore di liste P_S contenente, per ogni output, l'insieme dei cubi della SOP.
Per ogni prodotto $p_i = e_{i,1} \cdot e_{i,2} \cdots e_{i,k_i}$ di ogni output o, $\forall j \in \mathbb{N}.\, 1 \leq j \leq k_i$ vengono generate tutte le espansioni

$$p_i^j = e_{i,1} \cdots e_{i,j-1} \cdot e_{i,j+1} \cdots e_{i,k_i}$$

effettuate mediante rimozione di un singolo letterale; l'intersezione di p_i^j con l'Off-set della funzione fornisce il numero di mintermini $0 \to 1$ complementati, o *costo dell'espansione*, c_i^j. Se questa cardinalità è *minore o uguale* alla soglia C_t il prodotto viene inserito in una coda di priorità massima con peso w_i^j/c_i^j dove w_i^j misura il numero di prodotti della rappresentazione coperti da p_i^j, o *guadagno dell'espansione*; ovvero

$$w_i^j = |\{p_k \in P_S[o] \mid p_k \subset p_i^j\}|\,;$$

altrimenti il prodotto viene scartato. La priorità così definita ci permette di determinare come antecedenti le espansioni il cui rapporto *guadagno/costo* appare maggiore o, in altre parole, *le espansioni che coprono il maggior numero di prodotti complementando il minor numero di mintermini.*

Al termine delle espansioni in ogni possibile direzione dobbiamo selezionare il sottoinsieme di prodotti espansi I che massimizzi il guadagno complessivo all'interno della soglia di mintermini C_t; questo problema, nella sua forma più generica, è noto come *problema dello zaino*[2]. Poiché il problema dello zaino è **NP-arduo** utilizzeremo un metodo di risoluzione euristico mediante selezione greedy.

Definiamo come E l'errore in I, inizialmente uguale a zero; ogni elemento p_i^j viene estratto dalla coda con priorità massima e ne viene calcolato l'*errore effettivo*

$$m_i^j = c_i^j - |f^{dc} \cap p_i^j|$$

definito come il costo c_i^j meno la cardinalità dell'intersezione tra p_i^j e il DC-set originario della funzione[3]: se la somma tra l'errore effettivo m_i^j ed E non supera la soglia C_t il prodotto p_i^j viene aggiunto al sottoinsieme I e l'errore E aggiornato, altrimenti il prodotto viene scartato ed eseguita una nuova iterazione del ciclo.

Ogni volta che un prodotto espanso p_i^j viene aggiunto ad I rimuoviamo dalla coda tutte le altre espansioni provenienti da p_i, infatti, aggiungere ad I un nuovo prodotto proveniente dalla stessa espansione causerebbe l'inserimento di un nuovo termine già parzialmente coperto e un aumento del numero di letterali rispetto al prodotto p_i originale.

Una volta determinato l'insieme I è possibile procedere alla costruzione della rappresentazione SOP finale nel seguente modo:

1. Copia della dell'insieme P_S nel nuovo insieme P'_S.
2. Rimozione dei prodotti in P'_S coperti dai singoli prodotti espansi in I.
3. Aggiunta dei prodotti in I in P'_S.
4. Rimozione dei prodotti coperti dall'unione dei prodotti rimanenti in P'_S[4].

[2]Il problema dello zaino può essere definito come segue: dato un insieme U di n elementi dove ogni elemento $u \in U$ ha un valore intero $v(u) > 0$ e un peso intero $w(u) > 0$ e dato un intero positivo M, trovare un sottoinsieme $U' \subseteq U$ tale che $\sum_{u \in U'} w(u) \leq M$ e la somma dei valori degli elementi in U' sia la massima possibile.

[3]Escludiamo dall'errore computato il complemento di un mintermine appartenente al DC-set di f prima della minimizzazione.

[4]Sia P una rappresentazione, un prodotto $p' \in P$ è coperto dall'unione dei prodotti rimanenti se $p' \subseteq c$ e $c = \sum_{p \in P, p \neq p'} p$

5. Costruzione della SOP S'' prodotta dall'insieme P'_S.

6. La minimizzazione di S'' produce la rappresentazione finale S'[5] e la terminazione dell'algoritmo.

Secondo quanto detto possiamo concludere un'importante proprietà legata alla SOP S' risultante:

Lemma. *Sia R una rappresentazione in forma SOP, $literals(R)$ e $cubes(R)$ rispettivamente il numero di letterali e il numero di prodotti in R, allora vale che*

$$literals(S') \leq literals(S) \quad e \quad cubes(S') \leq cubes(S)$$

Inoltre, se il numero di mintermini complementati è > 0:

$$literals(S') < literals(S)$$

Dimostrazione. Segue dalla rimozione dei prodotti in coda provenienti dalle medesime espansioni, dalla rimozione dei prodotti originali coperti dai prodotti in I e dalla definizione di *espansione assistita*: queste garantiscono che per ogni implicante primo venga aggiunta *al più* una sua unica espansione e che lo stesso implicante venga rimosso poiché coperto dall'espansione stessa.
Se nessun mintermine è complementato allora $literals(S') = literals(S)$. □

Lo pseudocodice dell'algoritmo è mostrato di seguito.

[5] L'utilizzo di alcuni minimizzatori euristici come `Espresso`, potrebbe in alcuni casi peggiorare il numero di letterali e/o prodotti iniziale, per questo motivo l'implementazione prevede di selezionare la migliore tra le due rappresentazioni, prima e dopo la minimizzazione.

Algoritmo 1: Sintesi logica approssimata mediante espansione assistita

Function `ApproximateSyntesis(`f`, `r`):`
input: Funzione f, error rate r.
output: Una rappresentazione SOP della funzione approssimata f' con error rate r.
notazione: n ed m sono il numero di input e output di f.
$P_S[i]$ rappresenta l'insieme di prodotti dell'i-esimo output di una SOP S.
Q è una coda di priorità.

$C_t = r \cdot 2^n$
initPriorityQueue(Q)
$E = 0$ `/* inizializza l'errore */`
$\langle S, f_{min} \rangle$ = SOPminimization(f)
foreach *output* $o \in \{1, \ldots, m\}$ **do**
  foreach *prodotto* $p_i \in P_S[o]$ **do**
    foreach *letterale* $e_{i,j} \in p_i = e_{i,1} \cdot e_{i,2} \cdots e_{i,k_i}$ **do**
      $p_i^j = e_{i,1} \cdots e_{i,j-1} \cdot e_{i,j+1} \cdots e_{i,k_i}$
      $c_i^j = |f_{min}^{off} \cap p_i^j|$
      if $c_i^j \leq C_t$ ***and*** $p_i^j \notin P_S[o]$ **then**
        $w_i^j = 0$
        foreach *prodotto* $p_k \in P_S[o]$ **do**
          if $p_k \subset p_i^j$ **then** w_i^j += 1
        insertInPriorityQueue(Q, $\langle p_i^j, c_i^j, o \rangle$, w_i^j / c_i^j)
$P'_S = P_S$
while $Q \neq \emptyset$ ***and*** $E \neq C_t$ **do**
  $\langle p_i^j, c_i^j, o \rangle$ = pullMaximumFromQueue(Q)
  $m_i^j = c_i^j - |f^{dc} \cap p_i^j|$ `/* mintermini non nel DC-set originario */`
  if $E + m_i^j > C_t$ **then continue**
  E += m_i^j
  foreach $p_i^h \in Q$ **do**
    $Q = Q \setminus p_i^h$ `/* rimuovi da Q tutti i prodotti espansi da` p_i `*/`
  foreach $p_k \in P'_S[o]$ **do**
    `/* rimuovi dalla o-esima SOP i prodotti coperti da` p_i^j `*/`
    if $p_k \subseteq p_i^j$ **then** $P'_S[o] = P'_S[o] \setminus p_k$
  $P'_S[o] = P'_S[o] \cup p_i^j$
foreach *output* $o \in \{1, \ldots, m\}$ **do**
  foreach *prodotto* $p_i \in P'_S[o]$ **do**
    $g_i = constructSum(P'_S[o] \setminus p_i)$
    if $p_i \subseteq g_i$ **then** $P'_S[o] = P'_S[o] \setminus p_i$
$S' = ConstructSOP(P'_S)$
return *minimize(*S'*)*

2.4 Implementazione

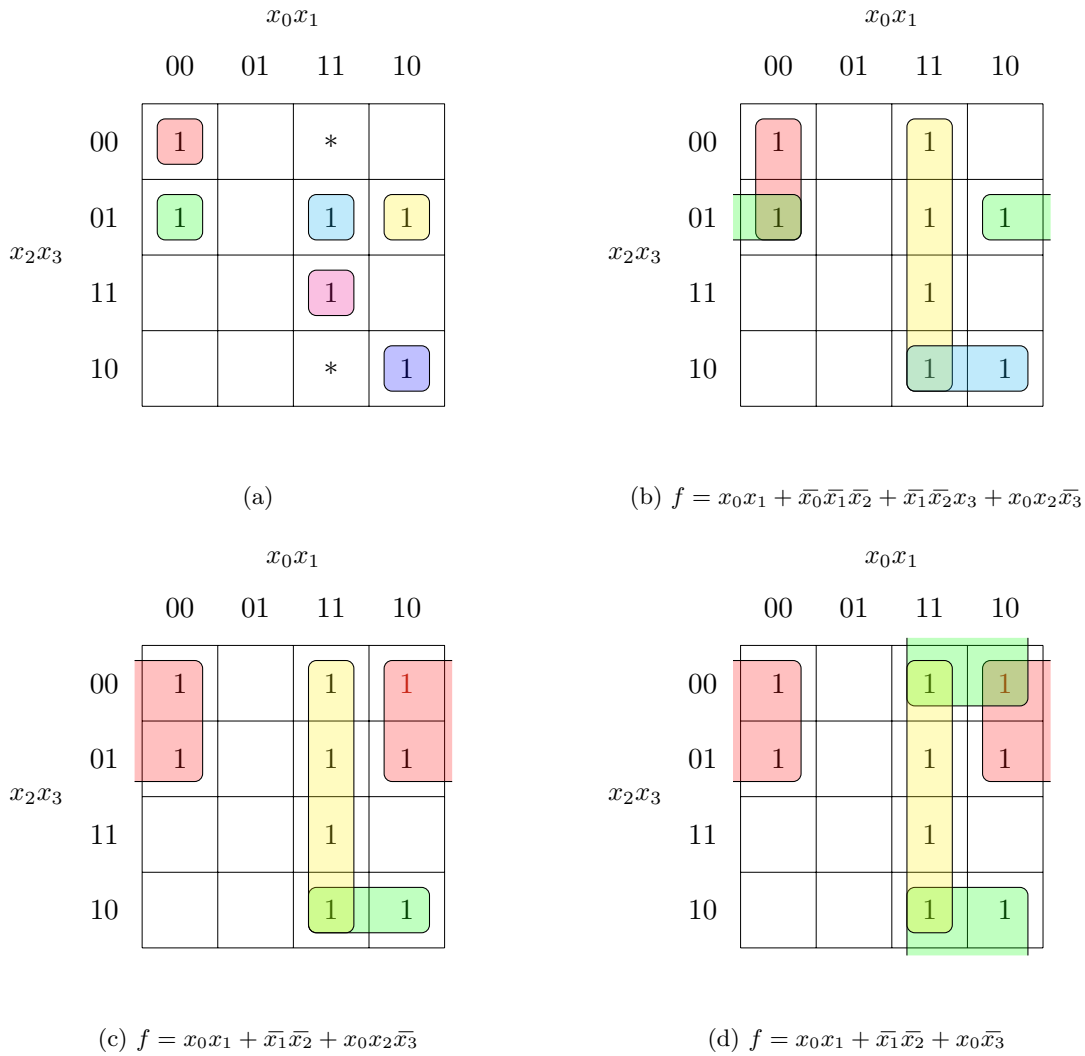


Figura 2.2: Esecuzione dell'algoritmo con $C_t = 1$: (a) Funzione di input f; (b) Minimizzazione di f mediante `Espresso`; (c) Risultato dell'euristica. L'espansione di $\overline{x_1}\,\overline{x_2}x_3$ in direzione x_3 è stata selezionate poiché è l'unica a coprire due prodotti: il mintermine $x_0\overline{x_1}\,\overline{x_2}\,\overline{x_3}$ viene complementato; (d) Esecuzione di `Espresso` sul risultato dell'euristica. Rispetto a (b) vi è un guadagno del 25% nel numero di prodotti e del $45,5\%$ nel numero di letterali.

Come anticipato, l'implementazione è stata realizzata in linguaggio *C*, utilizzando la libreria Cudd per la rappresentazione di funzioni Booleane mediante BDD e il minimizzatore `Espresso`, utilizzato in modalità euristica.

L'applicativo richiede come input il massimo numero di mintermini complementabili C_t (o l'error rate r), una funzione in formato ".pla" e restituisce una funzione approssimata rappresentata in forma SOP nel medesimo formato.

La libreria CUDD è stata utilizzata principalmente per eseguire le operazioni insiemistiche descritte nell'Algoritmo 1, per questo, ciascuno degli operandi coinvolti è rappresentato mediante un BDD in memoria. L'operatore di copertura, invece, è stato implementato mediante confronto tra i singoli letterali dei prodotti, rappresentati mediante vettori di interi.

Di seguito è indicato un sommario delle responsabilità dei singoli file sorgente. L'intero codice è allegato in Appendice.

- **config.h** contiene i parametri di configurazione come, ad esempio, il nome e la destinazione dei file temporanei e di output
- **libpla.c, libpla.h** contengono i metodi e le strutture dati per la gestione e interazione con i file ".pla"
- **main.c** il file principale: contiene l'intera euristica, la gestione di input/output e la verifica di correttezza dell'algoritmo
- **PLAparser.c, PLAparser.h** contengono i metodi per la creazione dei BDD associati ai file ".pla"
- **queue.c, queue.h** contengono la definizione della coda di priorità e l'implementazione dei metodi *push* e *pop* associati
- **utils.h** contiene metodi e strutture dati di utilità generica

2.5 Risultati sperimentali

Di seguito sono mostrati i risultati ottenuti sulle funzioni di benchmark contenute nella suite `Espresso` [14], eseguiti su Intel Core i5-8250U con 8 GB di memoria e sistema Linux.

È importante sottolineare come le proprietà delle funzioni di benchmark indicate (e di conseguenza i valori di guadagno riportati) facciano riferimento alla forma SOP ottenuta dalla *minimizzazione* con `Espresso` stesso e non all'input originale. Lo scopo dell'algoritmo è ottenere un guadagno su rappresentazioni SOP efficienti di funzioni Booleane, un confronto con funzioni non minimizzate produrrebbe risultati fuorvianti e fin troppo favorevoli.

In Tabella 2.1 è mostrato un sottoinsieme dei risultati ottenuti sulle funzioni di benchmark con *error rate* dell'1% comparando numero di letterali e prodotti. Le medie indicate (come tutte le successive) sono relative all'intero insieme di test e non al sottoinsieme mostrato in Tabella. In Figura 2.3 sono riportate le variazioni di guadagno percentuale in funzione dell'*error rate* r.

I risultati medi mostrano un guadagno oltre il 10% nel numero di letterali e prodotti a fronte di un'introduzione di errore dell'1% ed un guadagno superiore al 20% per *error rate* oltre il 10%. Il grafico, inoltre, mostra come la differenza tra le due metriche di guadagno, numero di letterali e numero di prodotti, risulti costante.
Mentre per molti circuiti l'errore introdotto non è sufficiente ad espandere l'insieme di implicanti primi[6] o produce un guadagno infinitesimale, per vari circuiti il guadagno risulta molto più importante: ad esempio, con un *error rate* dell'1% il circuito "rckl.pla" è sottoposto ad una riduzione nel numero di letterali superiore all'80% e superiore al 70% nel numero totale di prodotti.

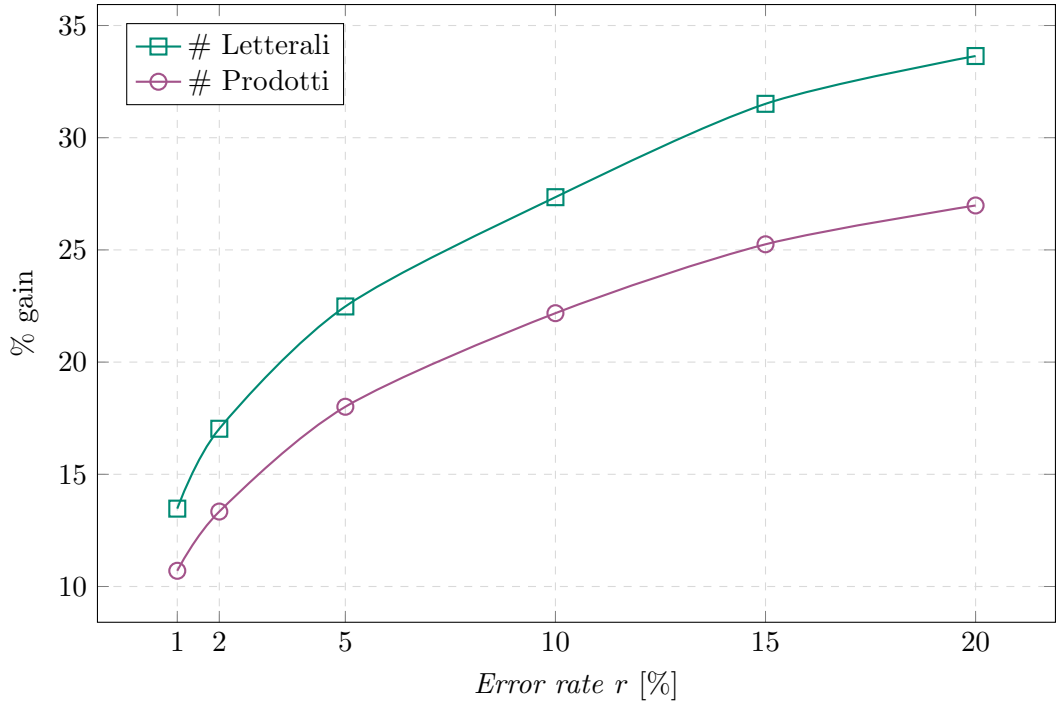
[6] Un guadagno dello 0% indica che nessun mintermine risulta complementato. In altre parole, per ogni PI tutte le possibili direzioni di espansione producono un numero di mintermini complementati $> C_t$.

Tabella 2.1: Risultati dei benchmark con *error rate* $r = 1\%$. Il prefisso "e" indica il risultato ottenuto mediante euristica.

Bench (in/out)	**Lett.**	**Cubi**	**eLett.**	**% gain**	**eCubi**	**% gain**	**Time**
add6 (12/7)	2196	355	1756	20,04	299	15,77	0,06
addm4 (9/8)	1531	235	1368	10,65	215	8,51	0,02
adr4 (8/5)	340	75	312	8,24	70	6,67	0,01
alcom (15/38)	211	49	211	0,00	49	0,00	0,01
amd (14/24)	1521	216	1335	12,23	193	10,65	0,01
apla (10/12)	383	58	340	11,23	55	5,17	0,01
b11 (8/31)	279	59	271	2,87	58	1,69	0,01
b12 (15/9)	198	58	190	4,04	55	5,17	0,01
co14 (14/1)	196	14	157	19,90	13	7,14	0,01
ex5 (8/63)	8406	1459	3374	59,86	743	49,07	0,06
ex7 (16/5)	754	119	638	15,38	107	10,08	0,01
exp (8/18)	1045	153	993	4,98	146	4,58	0,01
gary (15/11)	1901	221	1582	16,78	193	12,67	0,02
ibm (48/17)	882	173	833	5,56	168	2,89	0,01
in4 (32/20)	4277	411	3054	28,59	330	19,71	0,10
in5 (24/14)	1952	208	1342	31,25	155	25,48	0,01
in7 (26/10)	610	90	476	21,97	77	14,44	0,01
jbp (36/57)	1536	220	1442	6,12	217	1,36	0,01
l8err (8/8)	316	64	275	12,97	56	12,50	0,01
mainpla (27/54)	87397	6416	63225	27,66	4938	23,04	6,26
misg (56/23)	180	75	136	24,44	67	10,67	0,01
misj (35/14)	77	48	44	42,86	26	45,83	0,01
newtag (8/1)	18	8	7	61,11	5	37,50	0,01
newtpla1 (10/2)	33	4	29	12,12	4	0,00	0,01
newtpla2 (10/4)	97	15	71	26,80	12	20,00	0,01
newtpla (15/5)	176	23	159	9,66	22	4,35	0,01
newxcpla1 (9/23)	646	132	538	16,72	116	12,12	0,01
opa (17/69)	4039	524	3828	5,22	517	1,34	0,03
pdc (16/40)	3466	520	2240	35,37	360	30,77	0,03
poperom (6/48)	5393	1016	4882	9,48	944	7,09	0,02
prom2 (9/21)	26515	2954	26429	0,32	2945	0,30	0,97
radd (8/5)	340	75	318	6,47	71	5,33	0,01
rckl (32/7)	1896	98	347	81,70	27	72,45	0,01
spla (16/46)	7939	749	5587	29,63	548	26,84	0,10
t1 (21/23)	731	163	708	3,15	161	1,23	0,01
t2 (17/16)	415	74	374	9,88	70	5,41	0,01
ts10 (22/16)	896	128	895	0,11	128	0,00	0,01
vg2 (25/8)	804	110	701	12,81	102	7,27	0,01
vtx1 (27/6)	964	110	817	15,25	100	9,09	0,01
x1dn (27/6)	398	70	267	32,91	54	22,86	0,01
x6dn (39/5)	1388	173	1181	14,91	155	10,40	0,01
Z9sym (9/1)	516	86	438	15,12	76	11,63	0,01
Media				**13,47**		**10,70**	
Max				**81,70**		**72,45**	

Error rate	Media (# Letterali)	Media (# Prodotti)
1%	13.47%	10.70%
2%	17.03%	13.34%
5%	22.48%	18.01%
10%	27.35%	22.18%
15%	31.51%	25.25%
20%	33.64%	26.98%

(a) Guadagni medi percentuali sul numero di letterali e numero di prodotti per vari *error rate*.



(b) Grafico del guadagno medio percentuale in (a)

Figura 2.3: Variazione di guadagno medio sull'intero set di benchmark in funzione dell'*error rate*.

Capitolo 3

Bi-decomposizione di funzioni Booleane

La decomposizione è un paradigma per il partizionamento logico di funzioni mediante componenti più piccole. Una funzione f bi-decomposta può essere riscritta come

$$f = g \ \texttt{op} \ h$$

dove `op` rappresenta un operatore binario e g ed h dipendono dalle stesse variabili di f. In questa tesi è stata analizzata una particolare forma di bi-decomposizione, in cui g è un'approssimazione della funzione f ed h assimilabile ad un "fattore di correzione" per g [2]; l'obbiettivo di questi studi è volto alla costruzione di una *buona* approssimazione g tale per cui la rappresentazione ottenuta mediante bi-decomposizione sia *minore*, in base ad una determinata metrica (ad esempio nel numero di letterali), alla rappresentazione originale [6].

In quest'ultimo capitolo si presentano i risultati ottenuti applicando le tecniche di bi-decomposizione alla approssimazione g costruita tramite l'euristica studiata nel Capitolo 2. Per fare questo è stato scelto come operatore logico l'AND tra due funzioni. In altre parole, data un'approssimazione g di una funzione f ottenuta mediante sintesi logica approssimata, il nostro scopo è costruire una funzione h (o fattore di correzione) tale che $f = g \cdot h$. La rappresentazione ottenuta sarà poi confrontata con la rappresentazione originale.

3.1 Bi-decomposizione mediante AND

L'utilizzo dell'operatore AND per la bi-decomposizione può essere interpretato come una forma di *divisione Booleana*, dove f, g ed h rappresentano rispettivamente il dividendo, divisore e quoziente dell'operatore divisione.

Sia $f : \{0,1\}^n \mapsto \{0,1,*\}$ la funzione Booleana da decomporre e g una funzione totale che sia una approssimazione $0 \to 1$ di f (ovvero tale per cui $f \subseteq g$), per poter rappresentare f come $g \cdot h$, dove h è una funzione non completamente specificata, è necessario che

$$f^{on} \subseteq h^{on} \subseteq f^{on} \cup g^{off}$$

infatti, sia $v \in \{0,1\}^n$ un mintermine in n variabili, possono verificarsi unicamente i seguenti casi:

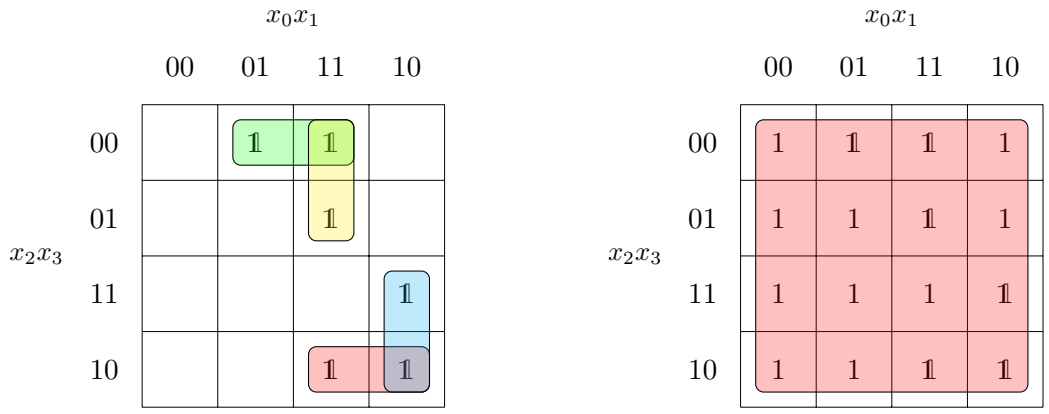
- $v \in f^{on} \Rightarrow (g(v) \cdot h(v) = 1 \Leftrightarrow g(v) = h(v) = 1)$
- $v \in f^{off} \Rightarrow h(v) = \begin{cases} 0 & \text{se } g(v) = 1 \\ * & \text{altrimenti} \end{cases}$

In altre parole, per poter ottenere i mintermini appartenenti all'On-set di f è necessario che entrambe le funzioni g ed h risultino uguali ad 1 per i mintermini appartenenti all'On-set di f, ma ciò è banale per g, essendo un'approssimazione $0 \rightarrow 1$ di f. Per quanto riguarda l'Off-set, invece, h è vincolata a risultare 0 in $g^{on} \setminus f^{on}$, ovvero esattamente nei mintermini *approssimati* da g. Infine, è possibile assegnare qualsiasi valore al rimanente insieme rappresentante il DC-set di h.
Si noti come la dimensione del DC-set di h determini il risultato della minimizzazione della funzione h stessa: maggiore è il DC-set e maggiore è la libertà concessa al minimizzatore nel processo di assegnamento. In Figura 3.1 è mostrata la massima espansione del DC-set di h, ovvero quando la funzione g coincide con f.

Formalizzando, data una approssimazione $0 \rightarrow 1$ di f, la costruzione della funzione h risulta:

- $h^{on} = f^{on}$
- $h^{off} = g^{on} \setminus f^{on}$
- $h^{dc} = g^{off} \cup f^{dc}$

Notazione: per semplificare la visualizzazione dei vari insiemi all'interno delle mappe di Karnaugh, i mintermini appartenenti ad f^{on} saranno indicati con il simbolo '𝟙', l'insieme $g^{on} \setminus f^{on}$ sarà indicato con '1' per la mappa di g e 0 per la mappa di h; inoltre, per quest'ultima, i simboli '1' e il simbolo ' ' indicano l'assegnamento, rispettivamente in 1 e 0, effettuato dal minimizzatore. L'Off-set di g è escluso dalla rappresentazione.



(a) $g = x_0x_2\overline{x_3} + x_1\overline{x_2}\overline{x_3} + x_0\overline{x_1}x_2 + x_0x_1\overline{x_2}$ (b) $h_{min} = 1$

Figura 3.1: Decomposizione banale di $f = x_0x_2\overline{x_3} + x_1\overline{x_2}\overline{x_3} + x_0\overline{x_1}x_2 + x_0x_1\overline{x_2}$ con $f = g$. La funzione h rappresenta la massima espansione del DC-set.

3.2 Algoritmo

L'algoritmo segue da quanto detto nel precedente paragrafo: a partire da una funzione Booleana f ed un error rate r viene ottenuta la funzione g, o divisore della funzione, mediante l'algoritmo di sintesi logica approssimata descritto nel Capitolo 2. Ottenuta la funzione g, la costruzione della funzione h è immediata. Infine, per ottenere una rappresentazione minima $g \cdot h$ è sufficiente minimizzare h.

Lo pseudocodice è mostrato di seguito.

Algoritmo 2: Bi-decomposizione mediante AND e sintesi logica approssimata

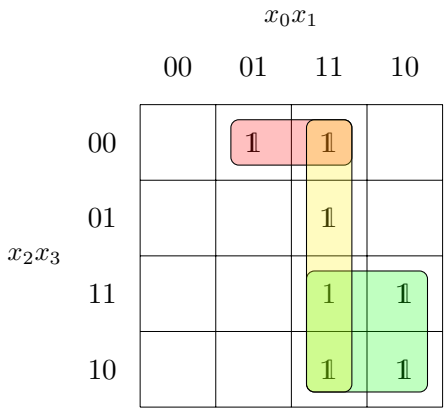
Function `AndDecomposition(`f, r`):`
input: Funzione f, error rate r
output: Minima rappresentazione $g \cdot h$ di f con error rate r

$g = ApproximateSyntesis(f, r)$
$h^{on} = f^{on}$
$h^{dc} = g^{off} \cup f^{dc}$
$h^{off} = g^{on} \setminus f^{on}$
assert$(f = g \cdot h)$

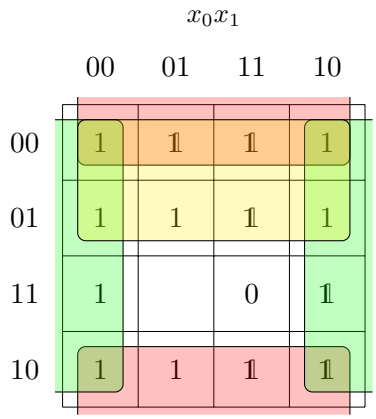
$h_{min} =$ SOPminimization(h)
return $g \cdot h_{min}$

Coerentemente con il precedentemente algoritmo, l'implementazione è stata realizzata in linguaggio *C*, utilizzando la libreria *Cudd* per le operazioni logiche su funzioni Booleane e file di input in formato PLA. Ad ogni esecuzione viene verificata la correttezza della decomposizione $f = g \cdot h$ mediante l'utilizzo del tool di confronto della suite `Espresso`.

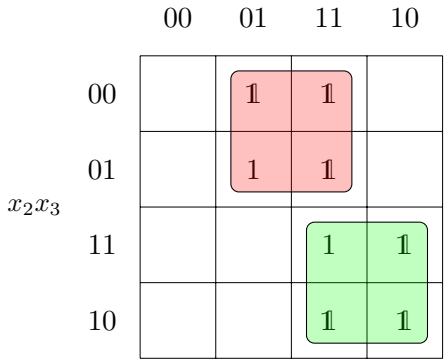
In Figura 3.2 sono mostrate le mappe di Karnaugh per g ed h relative a varie esecuzioni dell'algoritmo con diversi *error rate*; con $r = 13\%$ la decomposizione ottenuta riduce del 33% il numero originale di letterali per quella particolare funzione.
È possibile visionare il codice sorgente nella funzione **andDecomposition()** presente all'interno del file *main.c*.



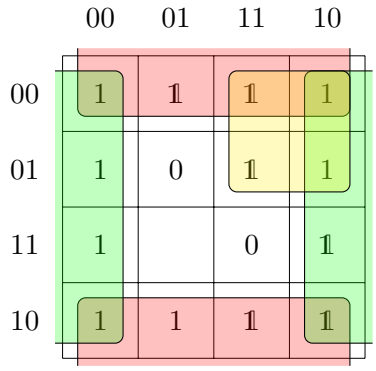
(a) $g = x_1\overline{x_2}\overline{x_3} + x_0x_1 + x_0x_2$



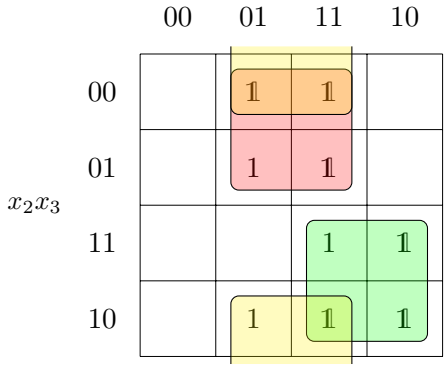
(b) $h_{min} = \overline{x_3} + \overline{x_1} + \overline{x_2}$



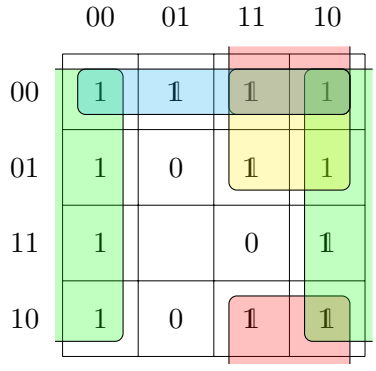
(c) $g = x_1\overline{x_2} + x_0x_2$



(d) $h_{min} = \overline{x_3} + \overline{x_1} + x_0\overline{x_2}$



(e) $g = x_1\overline{x_2} + x_0x_2 + x_1\overline{x_3}$



(f) $h_{min} = x_0\overline{x_3} + \overline{x_1} + x_0\overline{x_2} + \overline{x_2}\overline{x_3}$

Figura 3.2: Esecuzione degli algoritmi di sintesi e approssimazione su $f = x_0x_2\overline{x_3} + x_1\overline{x_2}\overline{x_3} + x_0\overline{x_1}x_2 + x_0x_1\overline{x_2}$ con *error rate* del 7%, 13% e 19%. Gli error rate 7% e 13% mostrano un guadagno sul numero di letterali, in particolare quest'ultimo ($f = (x_1\overline{x_2} + x_0x_2) \cdot (\overline{x_3} + \overline{x_1} + x_0\overline{x_2})$) induce una riduzione del 33% nel numero di letterali rispetto alla rappresentazione f originaria. Con il 19% di errore, invece, la rappresentazione ottenuta è superiore nel numero di letterali rispetto alla rappresentazione originaria.

3.3 Risultati sperimentali

I test sono stati condotti sul medesimo insieme di benchmark e col medesimo hardware utilizzato per l'analisi sperimentale dell'Algoritmo 1; anche in questo caso i valori di benchmark fanno riferimento alla rappresentazione minimizzata. A differenza dei test precedenti utilizzeremo come metrica di confronto unicamente il numero di letterali delle funzioni; questi, infatti, ritraggono una metrica maggiormente realistica nello stimare il costo di mappatura tecnologica di un circuito.

Sono state distinte due categorie di test effettuati sulle PLA di benchmark: con separazione di output o meno. Nel dettaglio, oltre ad eseguire l'algoritmo sulle singole funzioni, le quali sono prevalentemente multi-output, ogni file di benchmark è stato diviso in molteplici file, rappresentanti ognuno un singolo output della funzione. In questo modo, ogni file per funzione ad m output è diviso in m file di funzioni a singolo output e l'algoritmo applicato m volte, una per ogni output distinto.

In Tabella 3.1 sono mostrati i risultati ottenuti sulle funzioni di benchmark multi-output. Le medie ottenute attestano come l'applicazione dell'algoritmo all'insieme originale di funzioni fornisca un guadagno percentuale leggermente positivo, bilanciato da guadagni in media sia favorevoli che sfavorevoli; considerando invece la media unicamente delle funzioni il cui guadagno risulta positivo[1], utilizzando un *error rate* del 10% vi è un guadagno medio del $15,36\%$ nel numero di letterali. In Tabella 3.2, invece, sono indicati i risultati ottenuti sull'insieme di funzioni divise per singolo output. A differenza dei risultati precedenti, la media generica è massima (e maggiore rispetto alla precedente) per $r = 1\%$ e decrescente all'aumentare del valore stesso. Ha una tendenza inversa la media per i guadagni positivi, massima per $r = 20\%$ con un guadagno medio del $17,17\%$. In Figura 3.3 sono indicate le variazioni di guadagno all'aumentare dell'*error rate* r.

Un altro test condotto è stato effettuato sul guadagno medio risultante dalla minima rappresentazione ottenuta per i vari *error rate*. In altre parole, per ogni rappresentazione di f calcolata nei due test precedenti consideriamo la decomposizione equivalente col minor numero di letterali ottenuta, inclusa la decomposizione banale $f' = f \cdot 1$. In Tabella 3.3 sono mostrati i risultati del test, i quali suggeriscono come l'applicazione dell'algoritmo su intere funzioni multi-output, utilizzando *error rate* molteplici, risulti generalmente favorevole. Per le classi di funzioni nelle quali l'algoritmo produce una decomposizione con un minor numero di letterali, invece, l'applicazione dell'algoritmo sui singoli output della funzione fornisce un guadagno medio migliore, precisamente del 19,25%.

Infine, nell'ultimo test si propone la ricostruzione della funzione multi-output a partire dai singoli output col *minor numero di letterali* ottenuti. Nel dettaglio, le decomposizioni col minor numero di letterali ottenute applicando l'algoritmo ai distinti output della funzione sono state unite ottenendo nuovamente una rappresentazione multi-output equivalente al benchmark originario e ne è stato eseguito il confronto nel numero di letterali. In Tabella 3.4 è mostrato il risultato sull'intero insieme di benchmark; al termine della pagina sono indicate le medie ottenute: sulla totalità dei test vi è un guadagno del 13% nel numero di letterali complessivi, invece, escludendo dalla media i 25 su 124 benchmark che non producono guadagno nel numero di letterali vi è una riduzione media del 34,20%. Confrontando con i risultati presenti in Tabella 3.3 quest'ultimo metodo risulta nettamente più efficiente.

[1] Ad esempio è possibile modificare banalmente l'algoritmo di decomposizione nel seguente modo: date f, g, r calcola $f = g \cdot h$, se $literals(g \cdot h) < literals(f)$ restituisci $g \cdot h$, altrimenti restituisci $f' = f \cdot 1$

Tabella 3.1: Sottoinsieme dei risultati dei test sui benchmark multi-output per vari *error rate* r. In fondo sono indicati i risultati medi ottenuti su tutto l'insieme di benchmark.

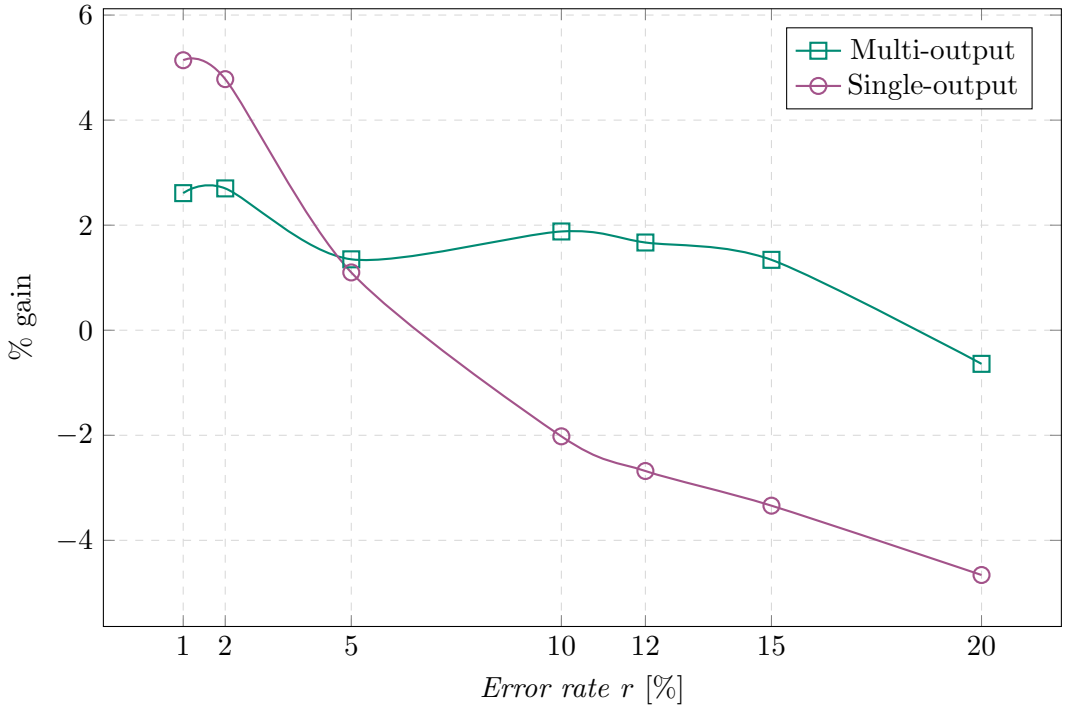
		1%		2%		5%		10%		12%		15%		20%	
Bench (in/out)	Lett.	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain
add6 (12/7)	2196	1790	18,49	1709	22,18	1887	14,07	1676	23,68	1737	20,90	1479	32,65	1525	30,56
addm4 (9/8)	1531	1430	6,60	1361	11,10	1377	10,06	1364	10,91	1472	3,85	1582	-3,33	1462	4,51
adr4 (8/5)	340	320	5,88	291	14,41	272	20,00	302	11,18	312	8,24	336	1,18	326	4,12
alcom (15/38)	211	211	0,00	211	0,00	194	8,06	198	6,16	199	5,69	191	9,48	246	-16,59
alu1 (12/8)	41	41	0,00	41	0,00	41	0,00	44	-7,32	44	-7,32	47	-14,63	50	-21,95
alu3 (10/8)	286	329	-15,03	312	-9,09	324	-13,29	338	-18,18	376	-31,47	371	-29,72	374	-30,77
apla (10/12)	383	386	-0,78	392	-2,35	441	-15,14	425	-10,97	425	-10,97	425	-10,97	425	-10,97
bcc (26/45)	8705	9392	-7,89	9877	-13,46	10219	-17,39	10219	-17,39	10219	-17,39	10219	-17,39	10219	-17,39
bench1 (9/9)	1875	1880	-0,27	1956	-4,32	2006	-6,99	2161	-15,25	2104	-12,21	2118	-12,96	2239	-19,41
br2 (12/8)	402	447	-11,19	466	-15,92	465	-15,67	465	-15,67	465	-15,67	465	-15,67	465	-15,67
co14 (14/1)	196	195	0,51	195	0,51	195	0,51	195	0,51	195	0,51	195	0,51	195	0,51
dc2 (8/7)	279	274	1,79	259	7,17	269	3,58	297	-6,45	301	-7,89	316	-13,26	324	-16,13
dekoder (4/7)	59	59	0,00	59	0,00	59	0,00	59	0,00	59	0,00	55	6,78	55	6,78
dk48 (15/17)	127	189	-48,82	189	-48,82	189	-48,82	190	-49,61	190	-49,61	228	-79,53	228	-79,53
ex1010 (10/10)	5274	5336	-1,18	5382	-2,05	5599	-6,16	5808	-10,13	5900	-11,87	6003	-13,82	6207	-17,69
ex7 (16/5)	754	654	13,26	637	15,52	493	34,62	472	37,40	464	38,46	509	32,49	541	28,25
exam (10/10)	575	635	-10,43	600	-4,35	749	-30,26	748	-30,09	842	-46,43	817	-42,09	831	-44,52
f51m (8/8)	323	313	3,10	309	4,33	297	8,05	295	8,67	310	4,02	331	-2,48	361	-11,76
fout (6/10)	493	492	0,20	485	1,62	506	-2,64	502	-1,83	496	-0,61	515	-4,46	503	-2,03
ibm (48/17)	882	984	-11,56	946	-7,26	892	-1,13	961	-8,96	969	-9,86	1006	-14,06	1088	-23,36
inc (7/9)	283	289	-2,12	281	0,71	262	7,42	253	10,60	265	6,36	274	3,18	284	-0,35
l8err (8/8)	316	286	9,49	293	7,28	294	6,96	324	-2,53	326	-3,16	329	-4,11	366	-15,82
linrom (7/36)	16142	16147	-0,03	16148	-0,04	16148	-0,04	16142	0,00	16139	0,02	9752	39,59	9297	42,40
luc (8/27)	1203	1211	-0,67	1212	-0,75	1195	0,67	1291	-7,32	1228	-2,08	1247	-3,66	1180	1,91
mainpla (27/54)	87397	78586	10,08	80759	7,60	84035	3,85	84352	3,48	85332	2,36	87417	-0,02	88812	-1,62
max1024 (10/6)	2669	2474	7,31	2427	9,07	2236	16,22	2293	14,09	2299	13,86	2370	11,20	2530	5,21
misg (56/23)	180	159	11,67	134	25,56	199	-10,56	180	0,00	180	0,00	180	0,00	180	0,00
newapla1 (12/7)	78	63	19,23	60	23,08	68	12,82	84	-7,69	94	-20,51	104	-33,33	104	-33,33
newapla2 (6/7)	42	42	0,00	45	-7,14	46	-9,52	43	-2,38	42	0,00	42	0,00	42	0,00
newill (8/1)	42	39	7,14	38	9,52	41	2,38	41	2,38	41	2,38	41	2,38	41	2,38
p1 (8/18)	750	733	2,27	745	0,67	806	-7,47	789	-5,20	772	-2,93	772	-2,93	827	-10,27
pdc (16/40)	3466	2387	31,13	2434	29,77	2282	34,16	2223	35,86	2201	36,50	2016	41,83	2110	39,12
poperom (6/48)	5393	4882	9,48	4918	8,81	5222	3,17	5019	6,93	4919	8,79	5247	2,71	5384	0,17
rd73 (7/3)	896	887	1,00	879	1,90	855	4,58	841	6,14	849	5,25	870	2,90	897	-0,11
root (8/5)	464	441	4,96	429	7,54	383	17,46	446	3,88	488	-5,17	487	-4,96	494	-6,47
shift (19/16)	399	399	0,00	399	0,00	398	0,25	455	-14,04	455	-14,04	428	-7,27	429	-7,52
t1 (21/23)	731	774	-5,88	775	-6,02	774	-5,88	769	-5,20	703	3,83	710	2,87	758	-3,69
t2 (17/16)	415	422	-1,69	463	-11,57	462	-11,33	398	4,10	396	4,58	403	2,89	399	3,86
t4 (12/8)	108	108	0,00	113	-4,63	113	-4,63	102	5,56	121	-12,04	126	-16,67	134	-24,07
test4 (8/30)	3874	3839	0,90	3910	-0,93	3930	-1,45	4101	-5,86	4205	-8,54	4210	-8,67	4361	-12,57
tms (8/16)	1804	1249	30,76	1311	27,33	1352	25,06	1476	18,18	1647	8,70	1409	21,90	1465	18,79
vg2 (25/8)	804	886	-10,20	880	-9,45	907	-12,81	847	-5,35	837	-4,10	817	-1,62	816	-1,49
wim (4/7)	55	48	12,73	48	12,73	48	12,73	52	5,45	52	5,45	54	1,82	75	-36,36
x6dn (39/5)	1388	1552	-11,82	1475	-6,27	1561	-12,46	1616	-16,43	1590	-14,55	1562	-12,54	1544	-11,24
xparc (41/73)	49901	49530	0,74	50821	-1,84	52382	-4,97	50273	-0,75	49630	0,54	48630	2,55	46729	6,36
z4 (7/4)	252	251	0,40	247	1,98	239	5,16	221	12,30	210	16,67	232	7,94	232	7,94
Z5xp1 (7/10)	539	447	17,07	412	23,56	390	27,64	415	23,01	395	26,72	378	29,87	367	31,91
Z9sym (9/1)	516	462	10,47	436	15,50	407	21,12	362	29,84	360	30,23	419	18,80	476	7,75
Media			**2,61**		**2,70**		**1,35**		**1,88**		**1,67**		**1,34**		**-0,64**
Media ($gain > 0$)			**9,31**		**10,88**		**14,78**		**15,36**		**14,27**		**14,59**		**15,26**

Tabella 3.2: Sottoinsieme dei risultati dei test sui benchmark single-output per vari *error rate* r. Ogni benchmark è rappresentato come "nome originale_numero output". In fondo sono indicati i risultati medi ottenuti su tutto l'insieme di benchmark.

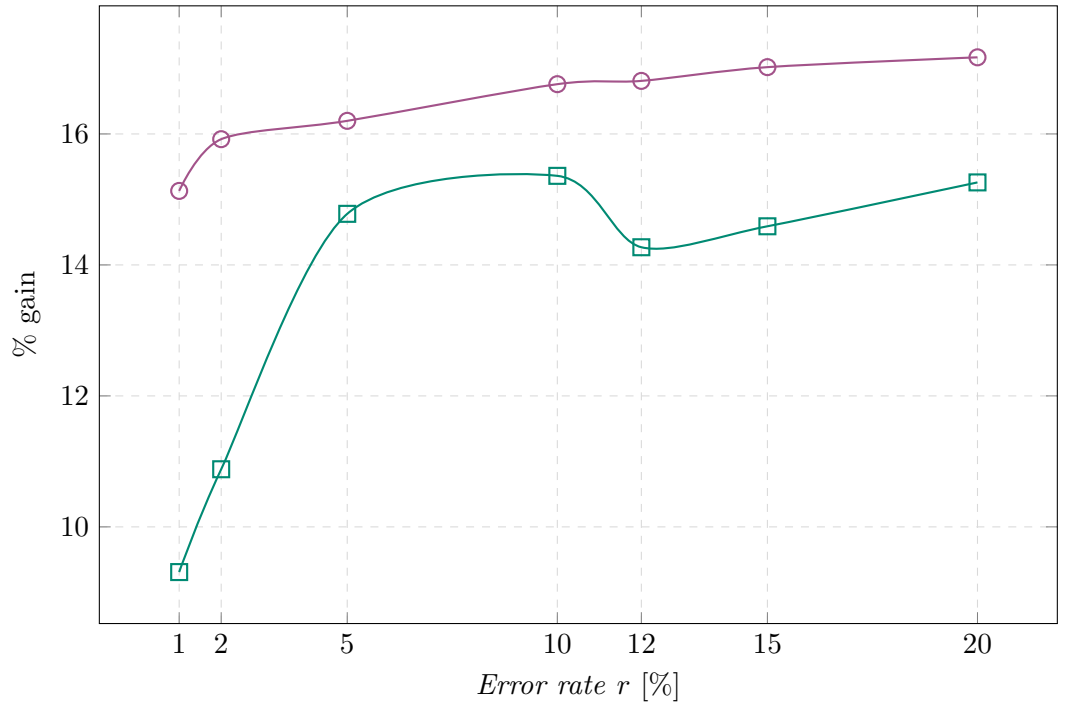
		1%		2%		5%		10%		12%		15%		20%	
Bench (in/out)	Lett.	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain	eLett	% gain
dist_2 (8/1)	108	97	10,19	84	22,22	95	12,04	92	14,81	96	11,11	112	-3,70	103	4,63
dist_3 (8/1)	178	156	12,36	150	15,73	153	14,04	160	10,11	175	1,69	170	4,49	186	-4,49
dist_4 (8/1)	236	230	2,54	219	7,20	201	14,83	207	12,29	217	8,05	224	5,08	221	6,36
dist_5 (8/1)	262	256	2,29	252	3,82	251	4,20	289	-10,31	288	-9,92	314	-19,85	289	-10,31
dk17_10 (10/1)	67	69	-2,99	69	-2,99	69	-2,99	69	-2,99	69	-2,99	69	-2,99	69	-2,99
dk17_11 (10/1)	26	27	-3,85	27	-3,85	27	-3,85	27	-3,85	27	-3,85	27	-3,85	27	-3,85
dk17_4 (10/1)	59	53	10,17	53	10,17	53	10,17	53	10,17	53	10,17	53	10,17	53	10,17
dk17_5 (10/1)	39	40	-2,56	40	-2,56	40	-2,56	40	-2,56	40	-2,56	40	-2,56	40	-2,56
newtpla_3 (15/1)	113	129	-14,16	131	-15,93	132	-16,81	132	-16,81	132	-16,81	132	-16,81	132	-16,81
newtpla_4 (15/1)	14	8	42,86	16	-14,29	16	-14,29	16	-14,29	16	-14,29	16	-14,29	16	-14,29
newxcpla1_11 (9/1)	12	7	41,67	7	41,67	14	-16,67	14	-16,67	14	-16,67	14	-16,67	14	-16,67
newxcpla1_14 (9/1)	9	9	0,00	9	0,00	5	44,44	5	44,44	5	44,44	5	44,44	10	-11,11
newxcpla1_1 (9/1)	11	12	-9,09	12	-9,09	17	-54,55	11	0,00	11	0,00	11	0,00	11	0,00
newxcpla1_22 (9/1)	26	18	30,77	16	38,46	29	-11,54	29	-11,54	29	-11,54	29	-11,54	29	-11,54
newxcpla1_7 (9/1)	22	23	-4,55	26	-18,18	23	-4,55	26	-18,18	26	-18,18	26	-18,18	26	-18,18
newxcpla1_9 (9/1)	7	7	0,00	7	0,00	6	14,29	6	14,29	6	14,29	6	14,29	10	-42,86
p1_10 (8/1)	43	35	18,60	40	6,98	42	2,33	48	-11,63	48	-11,63	48	-11,63	48	-11,63
p1_11 (8/1)	48	49	-2,08	45	6,25	46	4,17	46	4,17	46	4,17	46	4,17	46	4,17
p1_14 (8/1)	18	13	27,78	13	27,78	15	16,67	15	16,67	15	16,67	15	16,67	15	16,67
p1_15 (8/1)	79	71	10,13	67	15,19	68	13,92	65	17,72	65	17,72	65	17,72	65	17,72
p1_17 (8/1)	48	51	-6,25	45	6,25	40	16,67	40	16,67	40	16,67	40	16,67	40	16,67
p1_1 (8/1)	30	32	-6,67	25	16,67	22	26,67	18	40,00	18	40,00	18	40,00	18	40,00
p1_2 (8/1)	83	82	1,20	77	7,23	76	8,43	72	13,25	80	3,61	80	3,61	80	3,61
p1_7 (8/1)	40	37	7,50	37	7,50	41	-2,50	35	12,50	47	-17,50	47	-17,50	47	-17,50
p3_10 (8/1)	43	35	18,60	40	6,98	42	2,33	48	-11,63	48	-11,63	48	-11,63	48	-11,63
p3_11 (8/1)	48	49	-2,08	45	6,25	46	4,17	46	4,17	46	4,17	46	4,17	46	4,17
p3_13 (8/1)	10	10	0,00	6	40,00	12	-20,00	12	-20,00	12	-20,00	12	-20,00	12	-20,00
p3_14 (8/1)	18	13	27,78	13	27,78	15	16,67	15	16,67	15	16,67	15	16,67	15	16,67
p3_1 (8/1)	30	32	-6,67	25	16,67	22	26,67	18	40,00	18	40,00	18	40,00	18	40,00
p3_2 (8/1)	83	82	1,20	77	7,23	76	8,43	72	13,25	80	3,61	80	3,61	80	3,61
pdc_18 (16/1)	158	107	32,28	107	32,28	107	32,28	107	32,28	107	32,28	107	32,28	107	32,28
pdc_19 (16/1)	182	117	35,71	117	35,71	117	35,71	117	35,71	117	35,71	117	35,71	117	35,71
pdc_1 (16/1)	82	47	42,68	47	42,68	47	42,68	54	34,15	54	34,15	54	34,15	54	34,15
pdc_21 (16/1)	78	39	50,00	39	50,00	39	50,00	39	50,00	39	50,00	39	50,00	39	50,00
pdc_23 (16/1)	56	57	-1,79	57	-1,79	57	-1,79	57	-1,79	57	-1,79	57	-1,79	57	-1,79
pdc_25 (16/1)	105	110	-4,76	110	-4,76	110	-4,76	110	-4,76	110	-4,76	110	-4,76	110	-4,76
pdc_27 (16/1)	140	101	27,86	101	27,86	101	27,86	101	27,86	101	27,86	101	27,86	101	27,86
pdc_28 (16/1)	8	8	0,00	8	0,00	8	0,00	8	0,00	8	0,00	8	0,00	8	0,00
pdc_30 (16/1)	164	182	-10,98	219	-33,54	219	-33,54	226	-37,80	226	-37,80	226	-37,80	226	-37,80
pdc_33 (16/1)	21	23	-9,52	26	-23,81	26	-23,81	26	-23,81	26	-23,81	26	-23,81	26	-23,81
pdc_34 (16/1)	107	135	-26,17	135	-26,17	135	-26,17	142	-32,71	142	-32,71	142	-32,71	142	-32,71
pope.rom_2 (6/1)	31	31	0,00	34	-9,68	33	-6,45	41	-32,26	41	-32,26	39	-25,81	38	-22,58
pope.rom_38 (6/1)	24	24	0,00	24	0,00	20	16,67	20	16,67	20	16,67	23	4,17	23	4,17
pope.rom_7 (6/1)	6	6	0,00	6	0,00	6	0,00	6	0,00	6	0,00	6	0,00	6	0,00
pope.rom_9 (6/1)	49	49	0,00	52	-6,12	41	16,33	55	-12,24	55	-12,24	51	-4,08	62	-26,53
prom2_11 (9/1)	430	398	7,44	405	5,81	383	10,93	373	13,26	390	9,30	397	7,67	406	5,58
prom2_8 (9/1)	343	345	-0,58	333	2,92	348	-1,46	364	-6,12	385	-12,24	385	-12,24	394	-14,87
z4_1 (7/1)	56	55	1,79	51	8,93	44	21,43	57	-1,79	41	26,79	53	5,36	46	17,86
z4_2 (7/1)	136	135	0,74	133	2,21	133	2,21	141	-3,68	125	8,09	119	12,50	108	20,59
z4_3 (7/1)	48	48	0,00	48	0,00	47	2,08	39	18,75	39	18,75	40	16,67	48	0,00
z4_4 (7/1)	12	12	0,00	12	0,00	12	0,00	12	0,00	12	0,00	12	0,00	12	0,00
Z5xp1_10 (7/1)	1	1	0,00	1	0,00	1	0,00	1	0,00	1	0,00	1	0,00	1	0,00
Z5xp1_1 (7/1)	11	13	-18,18	13	-18,18	13	-18,18	7	36,36	7	36,36	7	36,36	12	-9,09
Z5xp1_2 (7/1)	27	23	14,81	26	3,70	19	29,63	22	18,52	28	-3,70	28	-3,70	31	-14,81
Z5xp1_3 (7/1)	46	42	8,70	40	13,04	35	23,91	43	6,52	47	-2,17	47	-2,17	53	-15,22
Z5xp1_4 (7/1)	82	78	4,88	76	7,32	73	10,98	75	8,54	85	-3,66	95	-15,85	94	-14,63
Z5xp1_5 (7/1)	60	60	0,00	56	6,67	54	10,00	57	5,00	49	18,33	57	5,00	57	5,00
Z5xp1_6 (7/1)	39	39	0,00	39	0,00	38	2,56	37	5,13	37	5,13	40	-2,56	41	-5,13
Z9sym_1 (9/1)	516	462	10,47	436	15,50	407	21,12	362	29,84	360	30,23	419	18,80	476	7,75
Media			**5,14**		**4,78**		**1,10**		**-2,02**		**-2,68**		**-3,34**		**-4,66**
Media ($gain > 0$)			**15,13**		**15,92**		**16,20**		**16,76**		**16,81**		**17,02**		**17,17**

	Media	Media ($gain > 0$)
Multi-output	12,13	16,26
Single-output	10,54	19,25

Tabella 3.3: Risultati medi ottenuti nei due tipi di test effettuati considerando le migliori decomposizioni risultanti per ogni benchmark.



(a) Media su tutto l'insieme di benchmark



(b) Media sul sottoinsieme di benchmark con guadagno positivo

Figura 3.3: Variazione di guadagno medio sull'intero set di benchmark in funzione dell'*error rate*.

Tabella 3.4: Risultati dei benchmark ottenuti ricostruendo la funzione multi-output utilizzando, per ogni output, la decomposizione col minor numero di letterali ottenuta dall'applicazione dell'algoritmo sul singolo output. In fondo sono indicati i risultati medi ottenuti su tutto l'insieme di benchmark.

Bench	Lett	eLett	% gain	Bench	Lett	eLett	% gain
add6 (12/7)	2196	1087	50,50	max1024 (10/6)	2669	1828	31,51
addm4 (9/8)	1531	1089	28,87	max128 (7/24)	3213	706	78,03
adr4 (8/5)	340	227	33,24	max512 (9/6)	1298	845	34,90
al2 (16/47)	545	306	43,85	misg (56/23)	180	134	25,56
alcom (15/38)	211	176	16,59	mlp4 (8/8)	927	700	24,49
alu2 (10/8)	305	420	-37,70	mp2d (14/14)	245	194	20,82
alu3 (10/8)	286	260	9,09	newapla1 (12/7)	78	57	26,92
amd (14/24)	1521	875	42,47	newapla (12/10)	131	79	39,69
apla (10/12)	383	668	-74,41	newbyte (5/8)	40	40	0,00
b10 (15/11)	1479	1110	24,95	newcond (11/2)	208	177	14,90
b11 (8/31)	279	170	39,07	newcpla1 (9/16)	325	173	46,77
b12 (15/9)	198	139	29,80	newcpla2 (7/10)	193	144	25,39
b2 (16/17)	8749	5152	41,11	newcwp (4/5)	55	37	32,73
b3 (32/20)	4003	2771	30,78	newill (8/1)	42	38	9,52
b4 (33/23)	832	703	15,50	newtag (8/1)	18	14	22,22
b7 (8/31)	279	170	39,07	newtpla2 (10/4)	97	51	47,42
b9 (16/5)	754	430	42,97	newxcpla1 (9/23)	646	204	68,42
bcddiv3 (4/4)	35	41	-17,14	p1 (8/18)	750	703	6,27
bench1 (9/9)	1875	2627	-40,11	p3 (8/14)	535	489	8,60
bench (6/8)	130	207	-59,23	p82 (5/14)	254	167	34,25
br1 (12/8)	516	425	17,64	pdc (16/40)	3466	3606	-4,04
br2 (12/8)	402	314	21,89	poperom (6/48)	5393	885	83,59
clpl (11/5)	55	48	12,73	prom2 (9/21)	26515	6312	76,19
dc1 (4/7)	84	59	29,76	radd (8/5)	340	209	38,53
dc2 (8/7)	279	210	24,73	rckl (32/7)	1896	960	49,37
dist (8/5)	947	728	23,13	rd53 (5/3)	160	122	23,75
dk17 (10/11)	197	382	-93,91	rd73 (7/3)	896	663	26,00
dk27 (9/9)	45	171	-280,00	risc (8/31)	242	175	27,69
dk48 (15/17)	127	534	-320,47	root (8/5)	464	299	35,56
ex1010 (10/10)	5274	8173	-54,97	ryy6 (16/1)	624	275	55,93
ex5 (8/63)	8406	759	90,97	shift (19/16)	399	398	0,25
ex7 (16/5)	754	430	42,97	spla (16/46)	7939	4201	47,08
exep (30/63)	1175	1400	-19,15	sqn (7/3)	224	179	20,09
exp (8/18)	1045	703	32,73	sqr6 (6/12)	295	194	34,24
exps (8/38)	7032	2883	59,00	sym10 (10/1)	1260	656	47,94
f51m (8/8)	323	269	16,72	t1 (21/23)	731	477	34,75
fout (6/10)	493	557	-12,98	t2 (17/16)	415	359	13,49
gary (15/11)	1901	1295	31,88	t3 (12/8)	217	175	19,35
in0 (15/11)	1901	1301	31,56	t4 (12/8)	108	311	-187,96
in1 (16/17)	8749	5152	41,11	test1 (8/10)	1560	1731	-10,96
in3 (35/29)	1815	1091	39,89	test3 (10/35)	19212	22428	-16,74
in4 (32/20)	4277	2919	31,75	test4 (8/30)	3874	6144	-58,60
in5 (24/14)	1952	1336	31,56	tial (14/8)	5218	4502	13,72
inc (7/9)	283	195	31,10	tms (8/16)	1804	436	75,83
jbp (36/57)	1536	1006	34,51	vg2 (25/8)	804	665	17,29
l8err (8/8)	316	439	-38,92	vtx1 (27/6)	964	835	13,38
life (9/1)	672	540	19,64	x1dn (27/6)	398	835	-109,80
linrom (7/36)	16142	1946	87,94	x6dn (39/5)	1388	1066	23,20
log8mod (8/5)	262	201	23,28	x9dn (27/7)	1138	1009	11,34
luc (8/27)	1203	554	53,95	xparc (41/73)	49901	20775	58,37
m181 (15/9)	205	140	31,71	z4 (7/4)	252	200	20,63
m1 (6/12)	619	116	81,26	Z5xp1 (7/10)	539	245	54,55
m2 (8/16)	2215	453	79,55	Z9sym (9/1)	516	360	30,23
m3 (8/16)	2577	605	76,52				
m4 (8/16)	3099	907	70,73	**Media**			**13,00**
mainpla (27/54)	87397	23797	72,77	**Media** ($gain > 0$)			**34,20**

Conclusioni

In questa tesi abbiamo, innanzitutto, descritto ed analizzato un approccio euristico per l'approssimazione di funzioni Booleane al fine di ridurne la complessità. I risultati ottenuti con l'algoritmo di sintesi mostrano, in media, un guadagno del 13,47% nel numero di letterali utilizzando una soglia di errore massima dell'1%, aumentando la soglia al 5% il guadagno ottenuto sale al 22,48%.
Successivamente, utilizzando i risultati ottenuti, abbiamo mostrato come l'approssimazione realizzata mediante espansione assistita produca risultati interessanti se usata come parte del processo di bi-decomposizione; applicando l'algoritmo sia sulla funzione completa che sui singoli output è stato esposto come quest'ultima fornisca, in media, un beneficio maggiore sul numero di letterali per le funzioni con guadagno positivo. Inoltre, è stato proposto un metodo per la decomposizione di funzioni multi-output mediante applicazione dell'algoritmo sui singoli output e, per differenti soglie di errore, ricostruita la funzione iniziale utilizzando le migliori rappresentazioni ottenute, inclusa la decomposizione banale $f' = f \cdot 1$; questo metodo risulta essere, sull'insieme di test, il migliore per guadagno medio nel numero di letterali, precisamente del 34,20% per funzioni con guadagno positivo.

Concludiamo citando alcuni possibili spunti per lavori futuri sia nell'ambito della sintesi logica che nell'ambito della bi-decomposizione; ad esempio, per l'algoritmo di sintesi, è stato scelto di dare precedenza alle espansioni con maggior peso w_i^j/c_i^j, ovvero le espansioni che, complementando il minor numero di mintermini, coprano il maggior numero di prodotti; questa metrica tuttavia fornisce risultati *mediamente* buoni, ma declassa espansioni che, per molte funzioni, risulterebbero, a conti fatti, più efficienti: lo sviluppo di una metrica maggiormente mirata potrebbe risultare in un interessante caso di studio e ad un miglioramento dei risultati euristici. Un'altra possibile modifica all'algoritmo potrebbe riguardare la rimozione di insiemi di letterali con cardinalità maggiore di 1 per l'espansione dell'implicante, e se la variazione di cardinalità rapportato all'incremento del tempo di calcolo produca risultati favorevoli o meno. Per quanto riguarda la bi-decomposizione, in [2] è mostrato come, oltre all'utilizzo dell'operatore AND, sia possibile costruire la funzione h utilizzando l'operatore logico $\not\Rightarrow$ ed una approssimazione $0 \to 1$ di f. Modificando l'Algoritmo 2 sarebbe possibile confrontare se, con l'approssimazione euristica, uno degli operatori risulta mediamente favorevole rispetto all'altro.

Bibliografia

[1] Anna Bernasconi and Valentina Ciriani. 2-spp approximate synthesis for error tolerant applications. In *2014 17th Euromicro Conference on Digital System Design*, pages 411–418. IEEE, 2014.

[2] Anna Bernasconi, Valentina Ciriani, Jordi Cortadella, and Tiziano Villa. Computing the full quotient in bi-decomposition by approximation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020.

[3] Anna Bernasconi, Valentina Ciriani, Fabrizio Luccio, and Linda Pagli. Exploiting regularities for boolean function synthesis. *Theory of Computing Systems*, 39(4):485–501, 2006.

[4] Robert K Brayton, Gary D Hachtel, Curt McMullen, and Alberto Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*, volume 2. Springer Science & Business Media, 1984.

[5] Melvin A Breuer. Intelligible test techniques to support error-tolerance. In *13th Asian test symposium*, pages 386–393. IEEE, 2004.

[6] Jordi Cortadella. Timing-driven logic bi-decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):675–685, 2003.

[7] Olivier Coudert and Tsutomu Sasao. Two-level logic minimization. In *Logic Synthesis and Verification*, pages 1–27. Springer, 2002.

[8] Gorschwin Fey and Rolf Drechsler. Utilizing bdds for disjoint sop minimization. In *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, volume 2, pages II–II. IEEE, 2002.

[9] Edward J McCluskey Jr. Minimization of boolean functions. *Bell system technical Journal*, 35(6):1417–1444, 1956.

[10] Willard V Quine. On cores and prime implicants of truth functions. *The American Mathematical Monthly*, 66(9):755–760, 1959.

[11] Michael Rice and Sanjay Kulhari. A survey of static variable ordering heuristics for efficient bdd/mdd construction. *University of California, Tech. Rep*, 2008.

[12] Doochul Shin and Sandeep K Gupta. Approximate logic synthesis for error tolerant applications. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 957–960. IEEE, 2010.

[13] Fabio Somenzi. Cudd: Cu decision diagram package-release 2.4. 0. *Software available from http://vlsi. colorado. edu*, 2004.

[14] Saeyang Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.

Appendice

Listing 1: main.c

```c
/**
 * @file main.c
 * @author Marco Costa
 * @brief file principale contenente euristica, decomposizione e gestione in/out
 * @date 2019-11-21
 *
 */
#define _POSIX_C_SOURCE 200809L
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/queue.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <limits.h>
#include <time.h>
#include <libgen.h>

#include <cudd.h>

#include "PLAparser.h"

#include "queue.h"
#include "libpla.h"
#include "config.h"
#include "utils.h"

#define print_usage(name)                                           \
    fprintf(stderr, "%s [options] [-m error] input-file.pla\n", name); \
    fprintf(stderr, "%s [options] [-g error] input-file.pla\n", name);

/**
 * @brief definisce il tipo di errore ammesso
 * ct: complementable terms
 */
enum
{
    GLOBAL_OUTPUT_ERROR,
    MULTIPLE_OUTPUT_ERROR
} error_mode = MULTIPLE_OUTPUT_ERROR;
unsigned long long ct = DEFAULT_CT;
double r;
int ct_percent = 0;

enum
{
    VERBOSE_LOG,
    TEST_LOG,
    DECOMPOSITION_LOG
} output_mode = VERBOSE_LOG;

int NUM_OUT;
int NUM_IN;

struct test_stats original_pla_stats;
struct test_stats heuristic_pla_stats;
struct test_stats espresso_pla_stats;

/**
 * @brief Costruisce una PLA data una struttura ParsedPLA
 *
 * @param pla la struttura
 * @param filename il file di out
 */
void mergeToPLA(ParsedPLA *pla, char *filename)
{
```

```
73      int max_matrix_row_size = 0;
74      for (int i = 0; i < NUM_OUT; i++)
75          max_matrix_row_size += N_CUBES[i];
76
77      int **M = safe_malloc(max_matrix_row_size * sizeof(int *));
78      for (int i = 0; i < max_matrix_row_size; i++)
79          M[i] = safe_calloc((NUM_IN + NUM_OUT), sizeof(int));
80
81      struct queue *q = CUBE_LIST;
82      int matrix_row_len = 0;
83      for (int o = 0; o < NUM_OUT; o++)
84      {
85          CubeListEntry *curr;
86          TAILQ_FOREACH(curr, &(q[o]), entries)
87          {
88              int add_tail = 1;
89              if (curr == NULL)
90                  continue;
91              for (int i = 0; i < matrix_row_len; i++)
92              {
93                  if (memcmp(M[i], curr->cube, NUM_IN * sizeof(int)) == 0)
94                  {
95                      M[i][NUM_IN + o] = 1;
96                      add_tail = 0;
97                      break;
98                  }
99              }
100
101             if (add_tail)
102             {
103                 memcpy(M[matrix_row_len], curr->cube, NUM_IN * sizeof(int));
104                 M[matrix_row_len][NUM_IN + o] = 1;
105                 matrix_row_len++;
106             }
107         }
108     }
109
110     printMatrixtoFile(filename, M, matrix_row_len, NUM_IN, NUM_OUT);
111
112     /* PULIZIA */
113     for (int i = 0; i < max_matrix_row_size; i++)
114         free(M[i]);
115     free(M);
116 }
117
118 /**
119  * @brief Costruisce una BDD contenente un singolo prodotto
120  *
121  * @param cube il prodotto come vettore di interi
122  * @param n_var il numero di variabili
123  * @return DdNode* la BDD
124  */
125 DdNode *construct_product(int *cube, int n_var)
126 {
127     DdNode *temp_node;
128     DdNode *new_node = Cudd_ReadOne(manager);
129     Cudd_Ref(new_node);
130
131     for (int i = 0; i < n_var; i++)
132     {
133         DdNode *var = Cudd_bddIthVar(manager, i);
134         if (cube[i] == 0) /* effettua il complemento */
135             temp_node = Cudd_bddAnd(manager, new_node, Cudd_Not(var));
136         else if (cube[i] == 1)
137             temp_node = Cudd_bddAnd(manager, new_node, var);
138         else
139             continue;
140
141         Cudd_Ref(temp_node);
142         Cudd_RecursiveDeref(manager, new_node);
143         new_node = temp_node;
144     }
145
```

```
146     return new_node;
147 }
148 
149 /**
150  * @brief Operatore di copertura mediante confronto tra due vettori di interi
151  *
152  * @param a
153  * @param b
154  * @return int 1 sse a copre b, 2 se a == b, 0 altrimenti
155  */
156 int covers(int *a, int *b, int n_in)
157 {
158     int equals = 2;
159     for (int i = 0; i < n_in; i++)
160     {
161         if (a[i] == 2 && b[i] != 2)
162             equals = 1; /* non sono uguali */
163         else if (a[i] != 2 && a[i] != b[i])
164             return 0;
165     }
166 
167     return equals;
168 }
169 
170 /**
171  * @brief Rimozione dei prodotti ridondanti in coda
172  *
173  * @param prod il prodotto inserito in coda
174  * @param q la coda
175  * @param n_in il numero di var. di input
176  */
177 void invalidateRedundantInQueue(product_t *prod, prior_queue *q, int n_in)
178 {
179     node_t *nodes = q->nodes;
180 
181     for (int i = 1; i <= q->len; i++)
182     {
183         product_t *curr_prod = nodes[i].data;
184 
185         /* c'è  in coda un prodotto proveniente dalla stessa origine o c'è  un prodotto uguale sullo
                stesso output */
186         if ((curr_prod != NULL) && (prod->output_f == curr_prod->output_f) &&
187             ((prod->product_number == curr_prod->product_number) || (memcmp(prod->cube,
                    curr_prod->cube, n_in * sizeof(int)) == 0)))
188             curr_prod->valid = 0;
189     }
190 }
191 
192 /**
193  * @brief Rimozione dei prodotti nella lista coperti da prod.
194  *
195  * @param prod il nuovo prodotto
196  * @param pla la lista di prodotti
197  */
198 void removeCoveredProducts(product_t *prod, ParsedPLA *pla)
199 {
200     CubeListEntry *curr;
201 
202     TAILQ_FOREACH(curr, &(CUBE_LIST[prod->output_f]), entries)
203     {
204         if ((prod->cube != curr->cube) && (covers(prod->cube, curr->cube, NUM_IN)))
205         {
206             TAILQ_REMOVE(&(CUBE_LIST[prod->output_f]), curr, entries);
207             (N_CUBES[prod->output_f])--;
208         }
209     }
210 }
211 
212 /**
213  * @brief Routine di pulizia.
214  *
215  * @param pla la pla principale
216  * @param offset la BDD Off-set
```

```
217  * @param dontPla la BDD DC-set
218  */
219 void cleanRoutine(ParsedPLA *pla, DdNode **offset, ParsedPLA *dontPla)
220 {
221     CubeListEntry *curr;
222 
223     for (int o = 0; o < NUM_OUT; o++)
224     {
225         Cudd_RecursiveDeref(manager, pla->vectorbdd_F[o]);
226         Cudd_RecursiveDeref(manager, dontPla->vectorbdd_F[o]);
227         Cudd_RecursiveDeref(manager, offset[o]);
228 
229         while ((curr = TAILQ_FIRST(&(CUBE_LIST[o]))) != NULL)
230         {
231             TAILQ_REMOVE(&(CUBE_LIST[o]), curr, entries);
232             free(curr->cube);
233             free(curr);
234         }
235     }
236 
237     free(N_CUBES);
238     free(pla->vectorbdd_F);
239     free(dontPla->vectorbdd_F);
240     free(offset);
241 }
242 
243 /**
244  * @brief Costruisce un file PLA a partire dall'On-set e DC-set di una funzione.
245  *
246  * @param filename il file di out
247  * @param on_set BDD rappresentante l'On-set
248  * @param dc_set BDD rappresentante il DC-set
249  */
250 void mergeBDDtoFile(char *filename, DdNode **on_set, DdNode **dc_set)
251 {
252     int max_matrix_row_size = 0;
253 
254     DdGen *gen1, *gen2;
255     int *cube;
256     CUDD_VALUE_TYPE value;
257     for (int i = 0; i < NUM_OUT; i++)
258     {
259         Cudd_ForeachCube(manager, on_set[i], gen1, cube, value)
260         {
261             max_matrix_row_size++;
262         }
263         Cudd_ForeachCube(manager, dc_set[i], gen2, cube, value)
264         {
265             max_matrix_row_size++;
266         }
267     }
268 
269     int **M = safe_malloc(max_matrix_row_size * sizeof(int *));
270     for (int i = 0; i < max_matrix_row_size; i++)
271         M[i] = safe_calloc((NUM_IN + NUM_OUT), sizeof(int));
272 
273     int matrix_row_len = 0;
274     for (int o = 0; o < NUM_OUT; o++)
275     {
276         DdGen *gen1, *gen2;
277         Cudd_ForeachCube(manager, on_set[o], gen1, cube, value)
278         {
279             int add_tail = 1;
280 
281             for (int i = 0; i < matrix_row_len; i++)
282             {
283                 if (memcmp(M[i], cube, NUM_IN * sizeof(int)) == 0)
284                 {
285                     M[i][NUM_IN + o] = 1;
286                     add_tail = 0;
287                     break;
288                 }
289             }
```

```
290
291             if (add_tail)
292             {
293                 memcpy(M[matrix_row_len], cube, NUM_IN * sizeof(int));
294                 M[matrix_row_len][NUM_IN + o] = 1;
295                 matrix_row_len++;
296             }
297         }
298         Cudd_ForeachCube(manager, dc_set[o], gen2, cube, value)
299         {
300             int add_tail = 1;
301
302             for (int i = 0; i < matrix_row_len; i++)
303             {
304                 if (memcmp(M[i], cube, NUM_IN * sizeof(int)) == 0)
305                 {
306                     M[i][NUM_IN + o] = 2;
307                     add_tail = 0;
308                     break;
309                 }
310             }
311
312             if (add_tail)
313             {
314                 memcpy(M[matrix_row_len], cube, NUM_IN * sizeof(int));
315                 M[matrix_row_len][NUM_IN + o] = 2;
316                 matrix_row_len++;
317             }
318         }
319     }
320
321     printMatrixtoFile(filename, M, matrix_row_len, NUM_IN, NUM_OUT);
322
323     for (int i = 0; i < max_matrix_row_size; i++)
324         free(M[i]);
325     free(M);
326 }
327
328 /**
329  * @brief Procedura per la decomposizione euristica di una funzione f data la sua approssimazione g
330  *  mediante operatore logico AND
331  *
332  * @param f_dc il DC-set della funzione f
333  * @param g_file il file PLA della funzione g
334  * @param f_file il file PLA della funzione f
335  */
336 void andDecomposition(ParsedPLA *f_dc, char *g_file, char *f_file)
337 {
338     char *command;
339     asprintf(&command, "espresso -Decho -of %s | sed -e '/\\.[p-type]/d' >" G_FILE, g_file);
340     system(command);
341     free(command);
342     asprintf(&command, "espresso -Decho -of %s | sed -e '/\\.[p-type]/d' >" F_FILE, f_file);
343     system(command);
344     free(command);
345
346     ParsedPLA f_on, g_on;
347     DdNode **g_off = safe_malloc(NUM_OUT * sizeof(DdNode *));
348
349     DdNode **h_off = safe_malloc(NUM_OUT * sizeof(DdNode *));
350     DdNode **h_dc = safe_malloc(NUM_OUT * sizeof(DdNode *));
351
352     parse(F_FILE, 0, &f_on, 0);
353     parse(G_FILE, 0, &g_on, 0);
354
355     /* f_off[i] = !(f_on[i] U f_dc[i]) */
356     for (int i = 0; i < NUM_OUT; i++)
357     {
358         g_off[i] = Cudd_Not(g_on.vectorbdd_F[i]);
359         Cudd_Ref(g_off[i]);
360
361         h_dc[i] = Cudd_bddOr(manager, g_off[i], f_dc->vectorbdd_F[i]);
362         Cudd_Ref(h_dc[i]);
```

```
363     }
364
365     char *curr_onset = safe_calloc((NUM_OUT + 1), sizeof(char));
366     fflush(stdin);
367     mergeBDDtoFile(TEMP_H_DECOMP, f_on.vectorbdd_F, h_dc);
368
369     char *sys_command;
370     asprintf(&sys_command, "espresso " TEMP_H_DECOMP " | sed -e '/\\.[p-type]/d' > " OUT_H_DECOMP);
371     system(sys_command);
372
373     ParsedPLA h_minim;
374     parse(OUT_H_DECOMP, 0, &h_minim, 0);
375
376     DdNode **and_out = safe_malloc(NUM_OUT * sizeof(DdNode *));
377
378     for (int i = 0; i < NUM_OUT; i++)
379         and_out[i] = Cudd_bddAnd(manager, g_on.vectorbdd_F[i], h_minim.vectorbdd_F[i]);
380
381     FILE *eq = fopen(G_TIMES_H_FILE, "w+");
382     fprintf(eq, ".i %d\n.o %d\n", NUM_IN, NUM_OUT);
383     for (int o = 0; o < NUM_OUT; o++)
384     {
385         for (int i = 0; i < NUM_OUT; i++)
386             curr_onset[i] = (i == o) ? '1' : '0';
387
388         DdGen *gen_onset;
389         int *cube;
390         CUDD_VALUE_TYPE value;
391         Cudd_ForeachCube(manager, and_out[o], gen_onset, cube, value)
392         {
393             for (int i = 0; i < NUM_IN; i++)
394                 (cube[i] == 2) ? fprintf(eq, "-") : fprintf(eq, "%d", cube[i]);
395             fprintf(eq, " %s\n", curr_onset);
396         }
397     }
398
399     /* pulizia */
400     fclose(eq);
401     free(curr_onset);
402     free(and_out);
403     free(sys_command);
404
405     for (int i = 0; i < NUM_OUT; i++)
406     {
407         Cudd_RecursiveDeref(manager, g_off[i]);
408         Cudd_RecursiveDeref(manager, h_dc[i]);
409         Cudd_RecursiveDeref(manager, h_off[i]);
410         Cudd_RecursiveDeref(manager, and_out[i]);
411     }
412
413     free(g_off);
414     free(h_dc);
415     free(h_off);
416     free(and_out);
417
418     /* verifica di correttezza */
419     if (output_mode == VERBOSE_LOG)
420     {
421         printf("\n*************************\n");
422         asprintf(&command, "espresso -Dverify %s " G_TIMES_H_FILE, f_file);
423     }
424     else
425         asprintf(&command, "espresso -Dverify %s " G_TIMES_H_FILE " >> /dev/null 2>> /dev/null",
              f_file);
426
427     if (system(command) != 0)
428     {
429         fprintf(stderr, "[!!] decomposition failed\n");
430         exit(EXIT_FAILURE);
431     }
432     free(command);
433 }
434
```

```
435 /**
436  * @brief Euristica di sintesi logica approssimata mediante espansione assistita.
437  *
438  * @param pla la funzione originale
439  * @param offset l'Off-set della funzione
440  * @param dontPla il DC-set della funzione
441  * @param s dati di test
442  * @return double il tempo di calcolo
443  */
444 double heuristic(ParsedPLA *pla, DdNode **offset, ParsedPLA *dontPla, struct test_stats *s)
445 {
446     clock_t beginClock = clock(), endClock;
447 
448     prior_queue *queue = safe_calloc(1, sizeof(prior_queue));
449     int *cube_iterator = safe_malloc(NUM_IN * sizeof(int));
450 
451     for (int o = 0; o < NUM_OUT; o++)
452     {
453         int product_i = 0;
454         CubeListEntry *curr_entry;
455 
456         TAILQ_FOREACH(curr_entry, &(CUBE_LIST[o]), entries)
457         {
458             memcpy(cube_iterator, curr_entry->cube, NUM_IN * sizeof(int));
459 
460             for (int i = 0; i < NUM_IN; i++)
461             {
462                 DdNode *curr_entry_node;
463 
464                 if ((cube_iterator[i] == 1) || (cube_iterator[i] == 0))
465                 {
466                     int dump = cube_iterator[i];
467                     cube_iterator[i] = 2;
468                     curr_entry_node = construct_product(cube_iterator, NUM_IN);
469 
470                     DdNode *intersect = Cudd_bddAnd(manager, curr_entry_node, offset[o]);
471                     Cudd_Ref(intersect);
472                     double complemented_minterms = Cudd_CountMinterm(manager, intersect, NUM_IN);
473 
474                     if ((complemented_minterms <= ct) && (complemented_minterms > 0)) /* può entrare
                          nella coda */
475                     {
476                         int covered_prod = 0;
477                         CubeListEntry *comparison_entry;
478                         TAILQ_FOREACH(comparison_entry, &(CUBE_LIST[o]), entries)
479                         {
480                             int ret = covers(cube_iterator, comparison_entry->cube, NUM_IN);
481 
482                             if (ret == 1) /* a copre b ma a != b */
483                                 covered_prod++;
484                             else if (ret == 2) /* il prodotto è già presente nella PLA */
485                             {
486                                 covered_prod = -1;
487                                 break;
488                             }
489                         }
490 
491                         if (covered_prod >= 0)
492                         {
493                             product_t *cube_queue = safe_malloc(sizeof(product_t));
494                             double priority = (double)covered_prod / complemented_minterms;
495                             cube_queue->output_f = o;
496                             cube_queue->compl_min = complemented_minterms;
497                             cube_queue->covered_prod = covered_prod;
498                             cube_queue->product_number = product_i;
499                             cube_queue->valid = 1;
500                             cube_queue->cube = safe_malloc(NUM_IN * sizeof(int));
501                             cube_queue->offset_inters = intersect;
502                             memcpy(cube_queue->cube, cube_iterator, NUM_IN * sizeof(int));
503 
504                             push(queue, priority, cube_queue);
505                             (N_CUBES[o])++;
506                         }
```

```
507                     else
508                         Cudd_RecursiveDeref(manager, intersect);
509                 }
510
511                 cube_iterator[i] = dump; /* ripristina cubo originale */
512                 Cudd_RecursiveDeref(manager, curr_entry_node);
513             }
514         }
515
516         product_i++;
517     }
518 }
519
520 free(cube_iterator);
521
522 if (output_mode == VERBOSE_LOG)
523 {
524     printf("**********************************\n");
525     if (error_mode == GLOBAL_OUTPUT_ERROR)
526         printf("Errore ammesso: max %lli mintermini sommati su tutti i %d output\n", ct,
              NUM_OUT);
527     else
528         printf("Errore ammesso: max %lli mintermini per ognuno dei %d output\n", ct, NUM_OUT);
529     printf("r = %g, NUM_IN = %d\n", r, NUM_IN);
530
531     printf("*****************************\n");
532     printf("Lunghezza coda prodotti eleggibili: %d\n", queue->len);
533 }
534
535 /* inizio estrazione coda */
536 unsigned long long *current_errors = safe_calloc(NUM_OUT, sizeof(unsigned long long));
537 unsigned long long total_error = 0;
538 int added_product = 0, dcset_error = 0;
539 product_t *curr_prod;
540
541 while (queue->len > 0)
542 {
543     if ((error_mode == GLOBAL_OUTPUT_ERROR) && (total_error >= ct))
544         break;
545
546     curr_prod = pop(queue);
547
548     if ((curr_prod == NULL) || (curr_prod->valid == 0))
549         continue;
550
551     DdNode *dcset_intersect = Cudd_bddAnd(manager, curr_prod->offset_inters,
          dontPla->vectorbdd_F[curr_prod->output_f]);
552     Cudd_Ref(dcset_intersect);
553     double dcset_minterms = Cudd_CountMinterm(manager, dcset_intersect, NUM_IN);
554     double effective_minterms = curr_prod->compl_min - dcset_minterms;
555
556     if (effective_minterms < 0)
557     {
558         fprintf(stderr, "Errore inatteso. Chiusura.\n");
559         printf("COMPL: %g, DCSET: %g\n", curr_prod->compl_min, dcset_minterms);
560         exit(EXIT_FAILURE);
561     }
562
563     Cudd_RecursiveDeref(manager, dcset_intersect);
564
565     if ((error_mode == MULTIPLE_OUTPUT_ERROR && (effective_minterms +
          current_errors[curr_prod->output_f] > ct)) ||
566         (error_mode == GLOBAL_OUTPUT_ERROR && (total_error + effective_minterms > ct)))
567         continue; /* selezione greedy, toglilo dalla coda e continua */
568
569     dcset_error += dcset_minterms;
570     current_errors[curr_prod->output_f] += effective_minterms;
571     total_error += effective_minterms;
572
573     added_product++;
574
575     invalidateRedundantInQueue(curr_prod, queue, NUM_IN);
576     removeCoveredProducts(curr_prod, pla);
```

```
577
578         CubeListEntry *expanded_product = safe_malloc(sizeof(CubeListEntry));
579         expanded_product->cube = curr_prod->cube;
580         TAILQ_INSERT_TAIL(&(CUBE_LIST[curr_prod->output_f]), expanded_product, entries);
581         (N_CUBES[curr_prod->output_f])++;
582
583         free(curr_prod);
584
585 #ifdef DEBUG
586         printf("Scelto prodotto con m_compl = %f, covered = %d *** Ct_%d = %d\n", effective_minterms,
587                curr_prod->covered_prod, curr_prod->output_f, current_errors[curr_prod->output_f]);
588         print_cube(cube->cube, NUM_IN);
589         printf("New queue len: %d\n", queue->len);
590         for (int i = 1; i <= queue->len; i++)
591         {
592             if (queue->nodes[i].data != NULL)
593             {
594                 printf("\t");
595                 print_cube(queue->nodes[i].data->cube, NUM_IN);
596                 printf(" - compl: %g, covered: %d, out: %d, priority: %g, valid: %d\n",
                         queue->nodes[i].data->compl_min,
597                          queue->nodes[i].data->covered_prod, queue->nodes[i].data->output_f,
                                 queue->nodes[i].priority,
598                          queue->nodes[i].data->valid);
599             }
600         }
601 #endif
602     }
603
604     if (output_mode == VERBOSE_LOG)
605     {
606         printf("Prodotti aggiunti: %d\n", added_product);
607         printf("*********************************\n");
608         printf("Errore totale computato: %lli\n", total_error);
609         printf("Errore DC-set: %d\n", dcset_error);
610         printf("Errore per output: ");
611         for (int i = 0; i < (NUM_OUT - 1); i++)
612             printf("%lli, ", current_errors[i]);
613         printf("%lli", current_errors[(NUM_OUT - 1)]);
614         printf("\n");
615
616         struct test_stats temp;
617         printf("\nDopo euristica -> ");
618         mergeToPLA(pla, OUTPUT_PLA);
619         getPLAFileData(OUTPUT_PLA, NUM_OUT, &temp);
620         print_verbose_stats(temp);
621     }
622
623     free(current_errors);
624
625     /**
626      * @brief rimozione dei prodotti coperti dall'OR di tutti i prodotti della funzione
627      *        eccetto lo stesso
628      */
629     for (int o = 0; o < NUM_OUT; o++)
630     {
631         CubeListEntry *outer, *inner;
632         TAILQ_FOREACH(outer, &(CUBE_LIST[o]), entries)
633         {
634             if (outer == NULL)
635                 continue;
636
637             DdNode *single_prod = construct_product(outer->cube, NUM_IN);
638             DdNode *foo_or = Cudd_ReadLogicZero(manager);
639             Cudd_Ref(foo_or);
640
641             TAILQ_FOREACH(inner, &(CUBE_LIST[o]), entries)
642             {
643                 if ((inner != NULL) && (outer != inner))
644                 {
645                     DdNode *curr_node = construct_product(inner->cube, NUM_IN);
646                     DdNode *tmp = Cudd_bddOr(manager, foo_or, curr_node);
647                     Cudd_Ref(tmp);
```

```
                    Cudd_RecursiveDeref(manager, foo_or);
                    Cudd_RecursiveDeref(manager, curr_node);
                    foo_or = tmp;
                }
            }

            /* il prodotto singolo è coperto dall'or, possiamo toglierlo */
            if (Cudd_bddLeq(manager, single_prod, foo_or))
            {
                TAILQ_REMOVE(&(CUBE_LIST[o]), outer, entries);
                (N_CUBES[o])--;
            }

            Cudd_RecursiveDeref(manager, single_prod);
            Cudd_RecursiveDeref(manager, foo_or);
        }
    }

    mergeToPLA(pla, MINIMIZED_OUTPUT_PLA);
    getPLAFileData(MINIMIZED_OUTPUT_PLA, NUM_OUT, s);
    if (output_mode == VERBOSE_LOG)
    {
        printf("Dopo euristica e rimozione ridondanze -> ");
        print_verbose_stats(*s);
    }

    endClock = clock();
    double time_spent = (double)(endClock - beginClock) / CLOCKS_PER_SEC;

    return time_spent;
}

/**
 * @brief funzione main, si veda la funzione "usage" per l'utilizzo da riga di comando
 *
 * @param argc
 * @param argv
 * @return int
 */
int main(int argc, char *argv[])
{
    struct stat st = {0};
    int ret;

    if (stat(TEMP_DIR, &st) == -1)
    {
        ret = mkdir(TEMP_DIR, 0700);
        if (ret == -1)
        {
            fprintf(stderr, "Impossibile creare la cartella " TEMP_DIR " : ");
            perror("");
            exit(EXIT_FAILURE);
        }
    }
    if (stat(OUTPUT_DIR, &st) == -1)
    {
        ret = mkdir(OUTPUT_DIR, 0700);
        if (ret == -1)
        {
            fprintf(stderr, "Impossibile creare la cartella " OUTPUT_DIR " : ");
            perror("");
            exit(EXIT_FAILURE);
        }
    }

    if (argc < 2)
    {
        print_usage(argv[0]);
        exit(EXIT_FAILURE);
    }

    int opt;
    char *endptr;
```

```
721
722     while ((opt = getopt(argc, argv, "dgmt")) != -1)
723     {
724         if ((opt == 'g') || (opt == 'm'))
725         {
726             if (argv[optind][strlen(argv[optind]) - 1] == '%')
727                 ct_percent = 1;
728
729             ct = strtol(argv[optind], &endptr, 10);
730             check_strtol(ct, argv[optind], endptr);
731             if (ct <= 0)
732             {
733                 fprintf(stderr, "[!!] L'errore deve essere >= 0\n");
734                 exit(EXIT_FAILURE);
735             }
736             if ((ct_percent) && (ct > 100))
737             {
738                 fprintf(stderr, "[!!] L'errore percentuale non può superare il 100%%\n");
739                 exit(EXIT_FAILURE);
740             }
741         }
742
743         if (opt == 'g')
744             error_mode = GLOBAL_OUTPUT_ERROR;
745         else if (opt == 'm')
746             error_mode = MULTIPLE_OUTPUT_ERROR;
747         else if (opt == 't')
748             output_mode = TEST_LOG;
749         else if (opt == 'd')
750             output_mode = DECOMPOSITION_LOG;
751         else
752         {
753             print_usage(argv[0]);
754             exit(EXIT_FAILURE);
755         }
756     }
757
758     if (access(argv[argc - 1], F_OK) == -1)
759     {
760         perror("[!!] impossibile accedere al file PLA");
761         print_usage(argv[0]);
762         exit(EXIT_FAILURE);
763     }
764
765     /* minimizzazione della funzione */
766     char *sys_command;
767     asprintf(&sys_command, "espresso -Decho -od %s | sed -e '/\\.[p-type]/d' > " DONT_CARE_PLA,
            argv[argc - 1]);
768     system(sys_command);
769     free(sys_command);
770 #ifndef EXACT_MINIMIZATION
771     asprintf(&sys_command, "espresso %s | sed -e '/\\.[p-type]/d' > " MINIM_PLA, argv[argc - 1]);
772 #endif
773 #ifdef EXACT_MINIMIZATION
774     asprintf(&sys_command, "espresso -Dexact %s | sed -e '/\\.[p-type]/d' > " MINIM_PLA, argv[argc
            - 1]);
775 #endif
776     system(sys_command);
777     free(sys_command);
778
779     /* ********************************************** */
780
781     ParsedPLA dcSetFunc, minimizedFunc;
782     DdNode **offsetBDD;
783     struct test_stats s;
784     double cpu_time;
785
786     /* conversione file e parsing */
787     convertDCSetPLA(DONT_CARE_PLA);
788     parse(DONT_CARE_PLA, 1, &dcSetFunc, 0);
789     parse(MINIM_PLA, 0, &minimizedFunc, 1);
790
791     NUM_IN = minimizedFunc.num_in;
```

```
792     NUM_OUT = minimizedFunc.num_out;
793
794     if ((ct_percent) && (CHAR_BIT * sizeof(ct) < NUM_IN))
795     {
796         fprintf(stderr, "[!!] impossibile utilizzare l'errore percentuale con %d input, "
797                         "specificare il numero di mintermini\n",
798                 NUM_IN);
799         exit(EXIT_FAILURE);
800     }
801
802     offsetBDD = safe_malloc(NUM_OUT * sizeof(DdNode *));
803     for (int i = 0; i < NUM_OUT; i++)
804     {
805         offsetBDD[i] = Cudd_Not(minimizedFunc.vectorbdd_F[i]);
806         Cudd_Ref(offsetBDD[i]);
807     }
808
809     getPLAFileData(MINIM_PLA, NUM_OUT, &original_pla_stats);
810
811     if (output_mode == VERBOSE_LOG)
812     {
813         printf("NUM IN: %d, NUM OUT: %d\n", NUM_IN, NUM_OUT);
814         printf("*********************************\nFunzione originale: ");
815         getPLAFileData(argv[argc - 1], NUM_OUT, &s);
816         print_verbose_stats(s);
817         printf("Funzione minimizzata: ");
818         print_verbose_stats(original_pla_stats);
819     }
820
821     unsigned long long two_pow = (NUM_IN < 31) ? (1L << NUM_IN) : powl(2L, NUM_IN);
822
823     if ((two_pow == 0LL) && (ct_percent))
824     {
825         fprintf(stderr, "[!!] errore inatteso, chiusura.");
826         exit(EXIT_FAILURE);
827     }
828
829     /* la probabilità ct è richiesta in percentuale su 2^NUM_IN */
830     if (ct_percent)
831     {
832         r = (double)ct / 100;
833         ct = floorl(r * two_pow);
834
835         if (ct < 0)
836         {
837             fprintf(stderr, "[!!] errore di range\n");
838             exit(EXIT_FAILURE);
839         }
840     }
841     else
842         r = (two_pow == 0LL) ? 0.0f : (double)ct / two_pow;
843
844     cpu_time = heuristic(&minimizedFunc, offsetBDD, &dcSetFunc, &heuristic_pla_stats);
845
846     system("espresso " MINIMIZED_OUTPUT_PLA " > " ESPRESSO_OUTPUT_PLA);
847     getPLAFileData(ESPRESSO_OUTPUT_PLA, NUM_OUT, &espresso_pla_stats);
848
849     /**
850      * @brief scegliamo tra la PLA dopo euristica e la PLA dopo euristica + espresso
851      *        quale delle due fornisce la riduzione maggiore di letterali e la scegliamo per il
852      *        confronto con la PLA originale minimizzata.
853      * NOTA: la precedenza viene data alla PLA col
854      *         1. minor numero di porte OR
855      *         2. minor numero di letterali AND
856      *         3. PLA euristica + espresso
857      */
858     struct test_stats *chosen_pla;
859
860     if (heuristic_pla_stats.or_port == espresso_pla_stats.or_port)
861         chosen_pla = (espresso_pla_stats.and_lit <= heuristic_pla_stats.and_lit) ?
                 &espresso_pla_stats : &heuristic_pla_stats;
862     else
863         chosen_pla = (espresso_pla_stats.or_port < heuristic_pla_stats.or_port) ?
```

```
            &espresso_pla_stats : &heuristic_pla_stats;
864
865     if (output_mode == VERBOSE_LOG)
866     {
867         printf("*********************************\nConfronto con PLA Espresso:\n");
868         system("espresso -Dverify " MINIMIZED_OUTPUT_PLA " " MINIM_PLA);
869         printf("\n*********************************\nConfronto con PLA euristica con ridondanze:\n");
870         system("espresso -Dverify " MINIMIZED_OUTPUT_PLA " " OUTPUT_PLA);
871         printf("\n*********************************\nEsecuzione di Espresso sulla PLA euristica
                senza ridondanze:\n");
872         print_verbose_stats(espresso_pla_stats);
873     }
874
875     /**
876      * @brief Calcolo del guadagno rispetto alla PLA originale
877      */
878     int or_diff, and_diff;
879     double or_perc, and_perc, tot_perc;
880
881     or_diff = original_pla_stats.or_port - chosen_pla->or_port;
882     or_perc = (original_pla_stats.or_port > 0) ? ((double)or_diff / original_pla_stats.or_port) *
            100 : 0;
883     and_diff = original_pla_stats.and_lit - chosen_pla->and_lit;
884     and_perc = ((double)and_diff / original_pla_stats.and_lit) * 100;
885
886     tot_perc = ((double)or_diff + and_diff) / (original_pla_stats.or_port +
            original_pla_stats.and_lit) * 100;
887
888     if (output_mode == VERBOSE_LOG)
889     {
890         printf("*********************************\nGuadagno: ");
891         printf("OR: %d, AND: %d, TOT: %d\n", or_diff, and_diff, (or_diff + and_diff));
892         printf("Guadagno percentuale: OR: %.2f%%, AND: %.2f%%, TOT: %.2f%%\n", or_perc, and_perc,
                tot_perc);
893         printf("*********************************\nCPU time: %gs\n", cpu_time);
894     }
895     /**
896      * @brief stampa il risultato dei test in formato CSV
897      *
898      * FORMATTAZIONE: nome_file, ct, r, orig_and, orig_or, new_and, new_or, and_%, or_%, CPU_time[s]
899      */
900     else if (output_mode == TEST_LOG)
901     {
902         char *pla_name = basename(argv[argc - 1]);
903         int len = strlen(pla_name);
904         for (int i = 1; i <= 4; i++)
905             pla_name[len - i] = '\0';
906
907         if (cpu_time == 0.00f)
908             cpu_time = 0.01f;
909
910         printf("%s (%d/%d); %lli; %g; %d; %d; %d; %d; %.2f\n",
911                pla_name, NUM_IN, NUM_OUT, ct, (r * (double)100),
912                original_pla_stats.and_lit, original_pla_stats.or_port,
913                chosen_pla->and_lit, chosen_pla->or_port,
914                cpu_time);
915
916         cleanRoutine(&minimizedFunc, offsetBDD, &dcSetFunc);
917         Cudd_Quit(manager);
918         return 0;
919     }
920
921     struct test_stats h_func_stats;
922     andDecomposition(&dcSetFunc, MINIMIZED_OUTPUT_PLA, argv[argc - 1]);
923     getPLAFileData(OUT_H_DECOMP, NUM_OUT, &h_func_stats);
924
925     struct test_stats g_times_h = {.and_lit = chosen_pla->and_lit + h_func_stats.and_lit,
926                                    .or_port = chosen_pla->or_port + h_func_stats.or_port};
927
928     if (output_mode == VERBOSE_LOG)
929     {
930         printf("\n************************\n");
931         printf("OLD SOP LENGTH - AND: %d, OR: %d, TOT: %d\n", original_pla_stats.and_lit,
```

```
932                 original_pla_stats.or_port, (original_pla_stats.and_lit +
                        original_pla_stats.or_port));
933             printf("NEW SOP LENGTH - AND: %d, OR: %d, TOT: %d\n", g_times_h.and_lit, g_times_h.or_port,
934                 (g_times_h.and_lit + g_times_h.or_port));
935         }
936         /**
937          * @brief stampa il risultato della decomposizione in formato CSV
938          *
939          * FORMATTAZIONE: name, ct, r, f_and, g*h_and
940          */
941         else if (output_mode == DECOMPOSITION_LOG)
942         {
943             double and_area_factor, or_area_factor, tot_area_factor;
944             char *pla_name = basename(argv[argc - 1]);
945             int len = strlen(pla_name);
946             for (int i = 1; i <= 4; i++)
947                 pla_name[len - i] = '\0';
948 
949             printf("%s (%d/%d); %d; %d;\n",
950                 pla_name, NUM_IN, NUM_OUT,
951                 original_pla_stats.and_lit, g_times_h.and_lit);
952         }
953 
954         cleanRoutine(&minimizedFunc, offsetBDD, &dcSetFunc);
955 
956         Cudd_Quit(manager);
957 
958         return 0;
959     }
```

Listing 2: config.h

```
/**
 * @file config.h
 * @author Marco Costa
 * @brief Configurazione per le directory temporanee e nomi dei file
 * @date 2019-11-25
 */

#ifndef _CONFIG_H
#define _CONFIG_H

#define TEMP_DIR "/tmp/pla/"
#define OUTPUT_DIR "./out/"

#define DONT_CARE_PLA TEMP_DIR "dontset.pla"
#define MINIM_PLA TEMP_DIR "minimized.pla"
#define OFFSET_PLA TEMP_DIR "offset.pla"

#define MINIMIZED_OUTPUT_PLA OUTPUT_DIR "out_minimized.pla"
#define ESPRESSO_OUTPUT_PLA OUTPUT_DIR "out_espresso.pla"
#define OUTPUT_PLA OUTPUT_DIR "out.pla"

#define DEFAULT_CT 1

/* decomposizione */
#define G_FILE TEMP_DIR "g_file.pla"
#define F_FILE TEMP_DIR "f_file.pla"

#define TEMP_H_DECOMP TEMP_DIR "temp_h_func.pla"
#define G_TIMES_H_FILE TEMP_DIR "decomp_check.pla"
#define OUT_H_DECOMP OUTPUT_DIR "h_func.pla"

#define ORIGINAL_ONSET_PLA TEMP_DIR "original_onset.pla"

#endif
```

Listing 3: libpla.h

```
/**
 * @file libpla.h
 * @author Marco Costa
 * @brief Contiene metodi e strutture per l'interazione tra CUDD e i file PLA
 * @date 2019-11-25
 */

#ifndef _LIBPLA_H
#define _LIBPLA_H

#include <stdio.h>
#include <cudd.h>
#include <sys/queue.h>

DdManager *manager; /**< CUDD manager */

typedef struct CubeListEntry
{
    int *cube;
    TAILQ_ENTRY(CubeListEntry)
    entries;
} CubeListEntry;

int *N_CUBES;
TAILQ_HEAD(queue, CubeListEntry)
*CUBE_LIST;

typedef struct ParsedPLA
{
    int num_in;         /**< number of input variables */
    int num_x;          /**< number of x variables */
    int num_out;        /**< number of output */
    DdNode **vectorbdd_F; /**< BDD array for output */
} ParsedPLA;

/**
 * @brief Struttura per l'inserimento di dati di una PLA
 */
struct test_stats
{
    int prod_in;
    int prod_out;
    int and_lit;
    int or_port;
};

static inline void print_verbose_stats(struct test_stats s)
{
    printf("IN: %d, OUT: %d, TOT: %d, AND LITERALS: %d, OR PORT: %d\n",
           s.prod_in, s.prod_out, (s.prod_in + s.prod_out), s.and_lit, s.or_port);
}

/**
 * @brief Converte un file PLA in un file PLA contenente unicamente il DC-set
 *
 * @param filename il file
 */
void convertDCSetPLA(char *filename);

/**
 * @brief Ottiene il numero di letterali e prodotti da un file PLA
 *
 * @param filename il file
 * @param function_out il numero di uscite della funzione
 * @param s la struttura
 */
void getPLAFileData(char *filename, int function_out, struct test_stats *s);

/**
 * @brief Inizializzazione di una struttura ParsedPLA
 *
 * @param bdd la struttura
```

```
 */
void initParsedPLA(ParsedPLA *bdd);

/**
 * @brief Inizializzazione di una struttura CubeListEntry di dimensione size
 *
 * @param size
 * @return CubeListEntry*
 */
CubeListEntry *alloc_node(int size);

/**
 * @brief Stampa di una matrice come file PLA.
 *
 * @param filename il file di out
 * @param M la matrice
 * @param len numero di righe
 * @param in numero di ingressi
 * @param out numero di uscite
 */
void printMatrixtoFile(char *filename, int **M, int len, int in, int out);

#endif
```

Listing 4: libpla.c

```
/**
 * @file libpla.c
 * @author Marco Costa
 * @brief Implementazione dei metodi contenuti in libpla.h;
 * @date 2019-11-25
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include "libpla.h"
#include "utils.h"

void initParsedPLA(ParsedPLA *bdd)
{
    CUBE_LIST = safe_malloc(bdd->num_out * sizeof(TAILQ_HEAD(queue, CubeListEntry)));
    for (int i = 0; i < bdd->num_out; i++)
        TAILQ_INIT(&(CUBE_LIST[i]));
    N_CUBES = safe_calloc(bdd->num_out, sizeof(int));
}

CubeListEntry *alloc_node(int size)
{
    CubeListEntry *c = safe_malloc(sizeof(CubeListEntry));
    c->cube = safe_malloc(size * sizeof(int));
    return c;
}

void convertDCSetPLA(char *filename)
{
    FILE *ft;
    int ch;
    char skip_buf[1024];

    ft = fopen(filename, "r+");
    if (ft == NULL)
    {
        fprintf(stderr, "cannot open target file %s\n", filename);
        exit(1);
    }

    while ((ch = fgetc(ft)) != EOF)
    {
        if (ch == '.')
            fgets(skip_buf, 1024, ft);
```

```
47          else if (ch == '~')
48          {
49              fseek(ft, -1, SEEK_CUR);
50              fputc('0', ft);
51              fseek(ft, 0, SEEK_CUR);
52          }
53          else if (ch == '2')
54          {
55              fseek(ft, -1, SEEK_CUR);
56              fputc('1', ft);
57              fseek(ft, 0, SEEK_CUR);
58          }
59      }
60      fclose(ft);
61  }
62
63  void printMatrixtoFile(char *filename, int **M, int len, int in, int out)
64  {
65      FILE *f = fopen(filename, "w+");
66      if (f == NULL)
67      {
68          fprintf(stderr, "Impossibile aprire il file %s:", filename);
69          perror(NULL);
70          exit(EXIT_FAILURE);
71      }
72
73      fprintf(f, ".i %d\n.o %d\n", in, out);
74      for (int i = 0; i < len; i++)
75      {
76          for (int j = 0; j < (in + out); j++)
77          {
78              if (j == in)
79              {
80                  // printf(" ");
81                  fprintf(f, " ");
82              }
83              if (M[i][j] == 2)
84              {
85                  // printf("-");
86                  fprintf(f, "-");
87              }
88              else
89              {
90                  // printf("%d", M[i][j]);
91                  fprintf(f, "%d", M[i][j]);
92              }
93          }
94          // printf("\n");
95          fprintf(f, "\n");
96      }
97
98      fclose(f);
99  }
100
101 void getPLAFileData(char *filename, int function_out, struct test_stats *s)
102 {
103     FILE *ft;
104     int ch;
105     char skip_buf[1024];
106     int output = 0, curr_line_literals = 0, curr_function_out = 0;
107     int in_n = 0, out_n = 0, tot_product = 0;
108
109     int *or_literals = safe_calloc(function_out, sizeof(int));
110
111     ft = fopen(filename, "r+");
112     if (ft == NULL)
113     {
114         fprintf(stderr, "cannot open target file %s\n", filename);
115         exit(1);
116     }
117
118     while ((ch = fgetc(ft)) != EOF)
119     {
```

```
120         if (ch == ' ')
121         {
122             output = 1;
123             curr_function_out = 0;
124         }
125         else if (ch == '\n')
126         {
127             curr_line_literals = output = 0;
128         }
129         else if (ch == '.')
130             fgets(skip_buf, 1024, ft);
131         else if (!output && ((ch == '0') || (ch == '1')))
132         {
133             in_n++;
134             curr_line_literals++;
135         }
136         else if ((output) && (ch == '1'))
137         {
138             out_n++;
139             tot_product += curr_line_literals;
140             (or_literals[curr_function_out])++;
141             curr_function_out++;
142         }
143         else if ((output) && (ch == '0'))
144             curr_function_out++;
145     }
146     fclose(ft);
147 
148     int total_or_port = 0;
149     for (int i = 0; i < function_out; i++)
150         total_or_port += or_literals[i];
151 
152     s->and_lit = tot_product;
153     s->or_port = total_or_port;
154     s->prod_in = in_n;
155     s->prod_out = out_n;
156 
157     free(or_literals);
158 }
```

Listing 5: PLAparser.h

```
#ifndef _PLA_PARSER_H
#define _PLA_PARSER_H

/**
 * @file PLAparser.h
 * @author Marco Costa
 * @brief Contiene il prototipo per il metodo di parsing dei file .pla
 * @date 2019-11-19
 */

#include <stdio.h>
#include <stdlib.h>
#include <cudd.h>

#include "libpla.h"

/**
 * @brief legge il file .pla e costruisce la bdd relativa
 * @param inputfile .pla file
 * @param init_manager se 1 il cudd manager deve essere inizializzato
 * @param bdd la bdd risultante
 * @param isMinimized se deve essere costruito il vettore di liste di prodotti
 * @return -1 in caso di errore, 1 altrimenti
 */
int parse(char *inputfile, int init_manager, ParsedPLA *bdd, int isMinimized);

#endif
```

Listing 6: PLAparser.c

```
/**
 * @file PLAparser.c
 * @author Marco Costa
 * @brief Contiene l'implementazione dei metodi per il parsing dei file .pla
 * @date 2019-11-19
 */

#include "PLAparser.h"
#include <string.h>

#include "utils.h"
#include "libpla.h"

#define MAX_LEN 512

int *cube;

/**
 * @brief costruisce un nodo rappresentante un singolo prodotto
 *
 * @param input il vettore di caratteri rappresentante il prodotto
 * @param pla la struttura relativa al file
 * @return DdNode* il nodo rappresentante il prodotto
 */
DdNode *read_product(char *input, ParsedPLA *pla)
{
  DdNode *f;
  DdNode *tmpNode;
  DdNode *var;

  f = Cudd_ReadOne(manager);
  Cudd_Ref(f);

  for (int i = pla->num_in - 1; i >= 0; i--)
  {
    if (input[i] == '-' || input[i] == '4' || input[i] == '~')
    {
      cube[i] = 2;
      continue;
    }
    var = Cudd_bddIthVar(manager, i);
```

```
42      if (input[i] == '0')
43      {
44        tmpNode = Cudd_bddAnd(manager, Cudd_Not(var), f);
45        cube[i] = 0;
46      }
47      else
48      {
49        tmpNode = Cudd_bddAnd(manager, var, f);
50        cube[i] = 1;
51      }
52      Cudd_Ref(tmpNode);
53      Cudd_RecursiveDeref(manager, f);
54      f = tmpNode;
55    }
56
57    return f;
58  }
59
60  /**
61   * @brief inserisce il nodo f all'interno della bdd
62   *
63   * @param f il nodo
64   * @param output l'output legato ad f
65   * @param pla la struttura del file
66   */
67  void build_bdd(DdNode *f, char *output, ParsedPLA *pla, int isMinimized)
68  {
69    DdNode *tmpNode;
70    for (int i = 0; i < pla->num_out; i++)
71    {
72      if (output[i] == '0') // OFF set
73        continue;
74      if (output[i] == '1')
75      { // ON set
76        tmpNode = Cudd_bddOr(manager, f, pla->vectorbdd_F[i]);
77        Cudd_Ref(tmpNode);
78        Cudd_RecursiveDeref(manager, pla->vectorbdd_F[i]);
79        pla->vectorbdd_F[i] = tmpNode;
80
81        if (isMinimized)
82        {
83          CubeListEntry *c = alloc_node(pla->num_in);
84          memcpy(c->cube, cube, pla->num_in * sizeof(int));
85          TAILQ_INSERT_TAIL(&(CUBE_LIST[i]), c, entries);
86          (N_CUBES[i])++;
87        }
88      }
89    }
90
91    Cudd_RecursiveDeref(manager, f);
92  }
93
94  int parse(char *inputfile, int init_manager, ParsedPLA *pla, int isMinimized)
95  {
96    char tmp[MAX_LEN];
97    int done = 0;
98    DdNode *f;
99    FILE *PLAFile;
100   PLAFile = fopen(inputfile, "r");
101   if (PLAFile == NULL)
102   {
103     fprintf(stderr, "Error opening file %s\n ", inputfile);
104     return -1;
105   }
106   while (!done && fscanf(PLAFile, "%s\n", tmp) != EOF)
107   {
108     if (tmp[0] == '.')
109     {
110       switch (tmp[1])
111       {
112       case 'i':
113       {
114         fscanf(PLAFile, "%d\n", &(pla->num_in));
```

```
115         if (init_manager)
116           manager = Cudd_Init(pla->num_in, 0, CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS, 0);
117       }
118       break;
119       case 'o':
120       {
121         int i;
122
123         fscanf(PLAFile, "%d\n", &(pla->num_out));
124         pla->vectorbdd_F = (DdNode **)calloc(pla->num_out, sizeof(DdNode *));
125         if (pla->vectorbdd_F == NULL)
126         {
127           fprintf(stderr, "INPUT vectorbdd_F: Error in calloc\n");
128           Cudd_Quit(manager);
129           fclose(PLAFile);
130           return -1;
131         }
132         for (i = 0; i < pla->num_out; i++)
133         {
134           pla->vectorbdd_F[i] = Cudd_ReadLogicZero(manager);
135           Cudd_Ref(pla->vectorbdd_F[i]);
136         }
137         done = 1;
138       }
139       break;
140       }
141     }
142   }
143   if (pla->num_in <= 0 || pla->num_out <= 0)
144   {
145     fclose(PLAFile);
146     return -1;
147   }
148
149   int readInput = 1;
150   int readOutput = 0;
151   int inputreaded = 0;
152   int outputreaded = 0;
153   char delimit[] = "|";
154   char *input = (char *)calloc(pla->num_in + 1, sizeof(char));
155   char *output = (char *)calloc(pla->num_out + 1, sizeof(char));
156
157   cube = safe_malloc(pla->num_in * sizeof(int));
158   if (isMinimized)
159     initParsedPLA(pla);
160
161   while (fscanf(PLAFile, "%s", tmp) > 0)
162   {
163
164     if (strlen(tmp) > pla->num_in + pla->num_out)
165     {
166       char *tmpstr;
167       char *p1 = strtok_r(tmp, delimit, &tmpstr);
168       strcpy(input, p1);
169       char *p2 = strtok_r(NULL, "\n", &tmpstr);
170       strcpy(output, p2);
171       f = read_product(input, pla);
172       Cudd_Ref(f);
173       build_bdd(f, output, pla, isMinimized);
174       readInput = 0;
175       readOutput = 0;
176     }
177     else if (readInput)
178     {
179       inputreaded += strlen(tmp);
180       if (inputreaded < pla->num_in)
181       { // input on multiple lines
182         input = strcat(input, tmp);
183       }
184       else
185       {
186         input = strcat(input, tmp);
187         readInput = 0;
```

```
188         readOutput = 1;
189         inputreaded = 0;
190         f = read_product(input, pla); // reads the product in PLA
191         Cudd_Ref(f);
192         input[0] = '\0';
193       }
194     }
195     else if (readOutput)
196     {
197       outputreaded += strlen(tmp);
198       if (outputreaded < pla->num_out)
199       { // output on multiple lines
200         output = strcat(output, tmp);
201       }
202       else
203       {
204         output = strcat(output, tmp);
205         readInput = 1;
206         readOutput = 0;
207         outputreaded = 0;
208         build_bdd(f, output, pla, isMinimized); // inserts the product in BDD
209         output[0] = '\0';
210       }
211     }
212   }
213
214   free(input);
215   free(output);
216   free(cube);
217   fclose(PLAFile);
218
219   return 1;
220 }
```

Listing 7: queue.h

```
#ifndef _QUEUE_H
#define _QUEUE_H

/**
 * @file queue.h
 * @author Marco Costa
 * @brief Include i prototipi e le strutture per la gestione della coda di
 *  priorità
 * @version 0.1
 */

#include <stdio.h>
#include <stdlib.h>

#include <cudd.h>

#define INIT_SIZE 20

/**
 * @brief Struttura rappresentante un singolo prodotto espanso
 */
typedef struct
{
    int *cube;        /* prodotto rappresentato */
    int output_f;     /* funzione di output */
    int covered_prod; /* numero di prodotti coperti */
    double compl_min; /* numero di mintermini complementati */
    int product_number; /* espansione di provenienza */
    int valid;        /* validità prodotto in coda */
    DdNode *offset_inters;
} product_t;

typedef struct
{
    double priority;
    product_t *data;
} node_t;

typedef struct
{
    node_t *nodes;
    int len;
    int size;
} prior_queue;

/**
 * @brief Inserimento di un elemento in coda con priorità "priority"
 *
 * @param h la coda di priorità
 * @param priority la priorità
 * @param data l'elemento da inserire in coda
 */
void push(prior_queue *h, double priority, product_t *data);

/**
 * @brief Estrazione del prodotto con priorità massima dalla coda jh
 *
 * @param h la coda
 * @return product_t* il prodotto con priorità massima
 */
product_t *pop(prior_queue *h);

#endif
```

Listing 8: queue.c

```
/**
 * @file queue.c
 * @author Marco Costa
 * @brief Implementazione delle funzioni per l'utilizzo della coda di priorità
 * @version 0.1
```

```
 */

#include <stdio.h>
#include <stdlib.h>

#include "queue.h"

void push(prior_queue *h, double priority, product_t *data)
{
    /* reallocazione del vettore in mancanza di memoria */
    if (h->len + 1 >= h->size)
    {
        h->size = h->size ? h->size * 2 : INIT_SIZE;
        h->nodes = (node_t *)realloc(h->nodes, h->size * sizeof(node_t));
    }

    int i = h->len + 1;
    int j = i / 2;

    while (i > 1 && h->nodes[j].priority < priority)
    {
        h->nodes[i] = h->nodes[j];
        i = j;
        j = j / 2;
    }
    h->nodes[i].priority = priority;
    h->nodes[i].data = data;
    h->len++;
}

product_t *pop(prior_queue *h)
{
    int i, j, k;

    product_t *data = h->nodes[1].data;

    h->nodes[1] = h->nodes[h->len];

    h->len--;

    i = 1;
    while (i != h->len + 1)
    {
        k = h->len + 1;
        j = 2 * i;
        if (j <= h->len && h->nodes[j].priority > h->nodes[k].priority)
            k = j;

        if (j + 1 <= h->len && h->nodes[j + 1].priority > h->nodes[k].priority)
            k = j + 1;

        h->nodes[i] = h->nodes[k];
        i = k;
    }
    return data;
}
```

Listing 9: utils.h

```c
#ifndef _UTILS_H
#define _UTILS_H

/**
 * @file utils.h
 * @author Marco Costa
 * @brief Funzioni di utilità generica per gestione della memoria
 *  e controlli di errore
 */

#include <stdio.h>
#include <stdlib.h>

#define print_cube(cube, n)  \
    for (int k = 0; k < n; k++) \
        printf("%d", cube[k]); \
    printf("\n");

#define min(a, b) \
    ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a < _b ? _a : _b; })

#define check_strtol(res, str, endptr)                    \
    if (endptr == str)                                    \
    {                                                     \
        perror("[!!] impossibile parsare il valore");     \
        exit(EXIT_FAILURE);                               \
    }                                                     \
    if ((res == LONG_MAX || res == LONG_MIN) && errno == ERANGE) \
    {                                                     \
        perror("[!!] il valore è out of range");          \
        exit(EXIT_FAILURE);                               \
    }

static inline void *safe_malloc(size_t n)
{
    void *p = malloc(n);
    if (!p && n > 0)
    {
        fprintf(stderr, "Impossibile allocare la memoria\n");
        exit(EXIT_FAILURE);
    }
    return p;
}

static inline void *safe_calloc(size_t nmemb, size_t nsize)
{
    void *p = calloc(nmemb, nsize);
    if (!p && nmemb > 0 && nsize > 0)
    {
        fprintf(stderr, "Impossibile allocare la memoria\n");
        exit(EXIT_FAILURE);
    }
    return p;
}

#endif
```