UNIVERSITÀ DEGLI STUDI DI NAPOLI
"FEDERICO II"

DIPARTIMENTO DI INGEGNERIA ELETTRICA E

TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea in Informatica
Tesi Sperimentale di Laurea Magistrale

ALGORITMI SIMBOLICI PER LA GENERAZIONE E RISOLUZIONE DI GIOCHI DI PARITÀ

Relatori
Prof. Marco Faella
Prof. Massimo Benerecetti
Prof. Fabio Mogavero
Correlatore
Prof. Piero Andrea Bonatti

Candidato
Orlando Aprea
Matr.: N97/226

Anno Accademico 2017 – 2018

Indice

1 Introduzione **1**

1.1 Cosa sono i Parity Game? 3

1.2 Perchè risolvere i Parity Game è utile 4

1.3 Perché utilizzare una codifica simbolica 6

2 Parity Games **8**

2.1 Definizione formale 8

2.2 Algoritmi di risoluzione 11

2.2.1 Nozioni fondamentali 11

2.2.2 Zielonka 13

2.2.3 Priority Promotion 15

2.3 Codifica Simbolica 18

3 Il tool LTSmin **21**

3.1 Struttura per Parity Games simbolici 21

3.1.1 I vertici in LTSmin 22

3.1.2 La relazione di transizione in LTSmin 23

3.1.3 Criteri implementativi 25

3.1.4 Le informazioni di supporto 26

3.2 Backend per la codifica simbolica 28

3.2.1 Gestione dei wrappers 28

3.2.2 I tipi di Backend 32

3.3 Da LPS a Parity Game 34

4 La libreria CuDD **39**

4.1 Caratteristiche 39

4.1.1 La struttura base 39

4.1.2 Tecniche per il miglioramento della performance 40

4.2 Integrazione di CuDD in LTSmin 42

4.2.1 Configurazione e installazione 42

4.2.2 Implementazione degli elementi di base 43

4.2.2.1 vector_set: 44

4.2.2.2 vector_relation: 47

4.2.2.3 vector_domain: 49

4.2.3 Implementazione dei Wrapper 50

5 Implementazione dei solver **58**

5.1 Algoritmo di Zielonka 58

5.1.1 Ottimizzazioni 61

5.2 Priority Promotion 65

5.2.1 Ottimizzazioni 69

5.3 L'Attrattore 71

6 Generatori Simbolici di Parity Games **73**

6.1 Convertitore di giochi espliciti in simbolici 74

6.1.1 Il tool PGSolver 74

6.2 Generatore di giochi simbolici random 78

6.2.1 Codifica di clausole 79

6.2.2 Generazione e suddivisione di insiemi di stati 80

6.2.3 Generazione e suddivisione delle transizioni: 84

6.2.4 Modalità alternativa di generazione 88

7 Sperimentazioni **91**

7.1 Generazione di test e raccolta dati 91

7.2 Analisi delle prestazioni 95

7.2.1 Le prestazioni di Zielonka 95

7.2.2 Le prestazioni di Priority Promotion 102
7.2.3 Prestazioni a confronto 105
7.2.4 Esempi di prestazioni su modelli reali 108

A Implementazioni **111**

B Manuale Utente **115**
B.1 Installazione 115
B.2 Utilizzo 117
B.2.1 I Risolutori 117
B.2.2 I Generatori 118

Elenco delle figure **120**

Bibliografia **125**

Capitolo 1

Introduzione

Fin da quando l'informatica teorica e la complessità computazionale si sono affermate come importanti aree di studio, è stato posto in atto dalla comunità scientifica uno sforzo sempre maggiore, nel tenativo di definire e circoscrivere nel miglior modo possibile, le classi di complessità in cui al giorno d'oggi dividiamo i problemi informatici. Conseguentemente, hanno guadagnato molta popolarità quei problemi di cui, da una parte, si sospetta l'appartenenza a una classe di complessità inferiore a quella che gli è attualmente attribuita come minima; dall'altra, presentano numerose applicazioni pratiche, o quanto meno ne è dimostrata la riducibilità in tempo polinomiale rispetto a problemi interessanti nella pratica. In quest'ottica la risoluzione di Parity Games rappresenta, come vedremo, un esempio emblematico di tali problemi. Proprio per questo motivo, codificare al meglio i Parity Games e progettare metodi di risoluzione sempre più efficienti e scalabili rispetto alla taglia del problema, risultano compiti particolarmente rilevanti. Scopo del seguente elaborato è quello di ristrutturare alcuni tra i principali algoritmi di solving finora proposti in letteratura, affinchè possano agire su PG codificati in una formulazione simbolica efficace. Successivamente, esplorare le performance di ciascuno e verificare quali siano i miglioramenti o peggioramenti, effettuando un confronto delle prestazioni tra algoritmi simbolici. Infine verificare se e come ciascun algoritmo tragga particolare beneficio, da una rappresentazione simbolica rispetto ad una esplicita.
Il seguente testo è cosi strutturato:

Di seguito verrà illustrato informalmente cosa sono i Parity Game, qual è la loro utilità, il significato di rappresentazione simbolica e perché con essa si aspira ad una performance migliore dei solver di PG.

Nel *Capitolo 2* verrà data una definizione più formale di cosa effettivamente sono i PG, verranno illustrati in maniera astratta gli algoritmi di risoluzione presi in esame e come è stata effettivamente ideata la codifica simbolica.

Nel *Capitolo 3* verrà invece presentato LTSmin, tool di model checking, già usato per la risoluzione di Parity Game simbolici e di cui il lavoro effettuato ha portato ad una integrazione.

Nel *Capitolo 4* sarà preso in oggetto CuDD la principale libreria per la gestione di Binary Decision Diagrams, una delle più usate strutture dati, per effettuare codifiche simboliche.

Nel *Capitolo 5* verrà posta l'attenzione sull'implementazione dei solver e tutte le migliorie e le ottimizzazioni pratiche ed algoritmiche inserite in essi.

Nel *Capitolo 6* ci si concentrerà sui sistemi ideati per generare, PG simbolici randomici, con tutte le strategie utilizzate per ottenere casi di studio paragonabili a PG generati a partire da situazioni reali; così da poter effettuare verifiche delle prestazioni il più possibile veritiere.

Infine nel *Capitolo 7* verranno mostrati i risultati sperimentali ottenuti, partendo dai dati raccolti dai solver sui diversi giochi generati.

1.1 Cosa sono i Parity Game?

Nei successivi capitoli verrà fornita una definizione formale di cosa sono i PG e quali sono le loro caratteristiche salienti. Tuttavia, di seguito è illustrata una prima idea informale di come sono strutturati e come vengono "giocati".
Possiamo definire i PG, come giochi infiniti a due giocatori svolti su grafi diretti di dimensione finita. I giocatori effettuano una mossa all'interno del gioco, muovendo un "token" tra i vertici del grafo, scegliendo uno dei percorsi che offrono gli archi che collegano i vertici. Ciascuno dei nodi è idealmente posseduto da uno dei due giocatori, cosicchè quando il token si trova su uno specifico vertice, il giocatore ad effettuare la mossa è colui che "possiede" il vertice. Inoltre a ciascuno dei nodi del grafo è assegnato un ranking (o priorità); tale valore è utilizzato ai fini di determinare chi sia il vincitore all'interno del gioco. Condizione di vittoria per un giocatore di un PG, è che il valore di ranking migliore incontrato infinitamente spesso durante il gioco, sia della "parità" del dato giocatore. Per meglio definire che cosa s'intende "vincere un Parity Game", di seguito introduciamo un primo accenno a tre importanti caratteristiche di tali giochi:

- Una volta creato il gioco e stabilito un vertice di partenza è possibile determinare chi sia il giocatore sempre vincente, generando per esso una strategia, che se seguita, lo porti alla vittoria indipendentemente dalle mosse dell'avversario;

- Come lasciato intendere dal punto precedente, chi sia il giocatore vincente di un PG è determinato oltre che dalla struttura del gioco, anche dal vertice di partenza; più in generale vedremo che per ciascun vertice esiste una strategia vincente, ma non può accadere che esista una strategia vincente per entrambi i giocatori;

- Infine una caratteristica molto comoda dei PG è il loro essere "senza memoria", e cioè, per determinare la "prossima mossa ottimale" il giocatore vincente non ha necessità di ricordare nessuna delle mosse precedentemente effettuate[Grädel et al., 2002].

Queste tre caratteristiche evidenziano che il parlare di vittoria di un PG risulta in qualche modo improprio: in base al vertice di partenza infatti, il gioco può essere vinto da un giocatore piuttosto che da un altro, determinando così delle zone di vertici vincenti (dette winning regions). Possiamo quindi meglio affermare che un vertice si considera vinto da uno dei due giocatori, se a partire da esso, qualsiasi siano le successive mosse dell'avversario, esiste una "strategia" vincente per quel giocatore.

1.2 Perchè risolvere i Parity Game è utile

I PG presentano caratteristiche particolamente utili e interessanti dal punto di vista della complessità computazionale e del model checking, in quanto:

- tale problema appartiene alla classe $UPTime \cap CoUPTime$, e non è stata ancora dimostrata la sua non risolubilità in tempo polinomiale [Jurdziński, 1998],
- il problema della risoluzione di un PG può trovare numerose applicazioni, è dimostrata infatti, l'esistenza di una riduzione in tempo lineare da esso, ai problemi di model checking e soddisfacibilità di formule in una logica molto espessiva come la modal μ-calculus e viceversa [Emerson et al., 2001][Grädel et al., 2002] [Emerson and Jutla, 1991] [Kupferman and Vardi, 1998][Obdržálek, 2006].

Particolarmente evidente nel secondo punto, è l'importanza che i PG possono rivestire. Dato, infatti, un sistema a transizioni di stati e una sua proprietà espressa con una formula in logica μ-calculus, è possibile, partendo da essi, codificare un PG, la cui risoluzione è legata alla soddisfacibilità della formula. Data, infatti, una formula α, un transition system Y, uno stato del sistema y_0, un PG codificato in base ad Y e α e detta strategia S_x una strategia vincente per il player x nel gioco a partire dalla configurazione iniziale (y_0, α) possiamo dire che: Y a partire dallo stato y_0 soddisfa la formula α se e solo se esiste una strategia vincente S_x all'interno del PG, a partire dal vertice codificato con (y_0, α).
Vediamo un semplice esempio pratico: Consideriamo il gioco da tavolo Forza Quattro; riassumendo rapidamente, il gioco ha le seguenti caratteristiche:

- si gioca su una griglia 6×7 in cui è possibile inserire tasselli del proprio colore,
- ogni giocatore può inserire a turno, un tassello del proprio colore,
- vince il gioco chi allinea per primo 4 tasselli del proprio colore.

è facilmente intuibile che è possibile codificare per tale gioco, un transition system che lo rappresenti, identificando con ogni stato, una possibile configurazione di gioco e con una transizione, il possibile passaggio da una configurazione ad un altra.

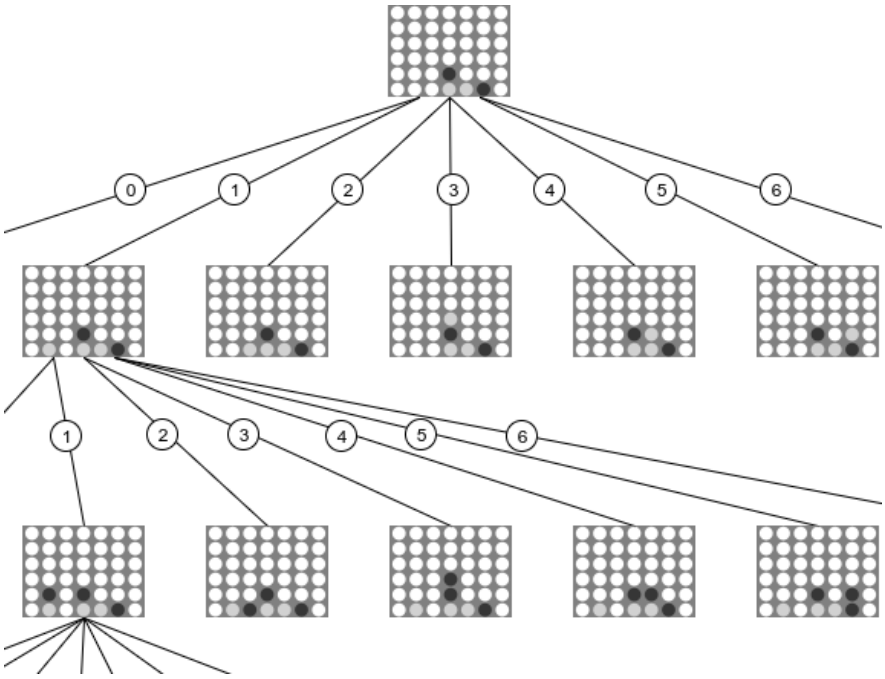


Figura 1.1: Parte di un albero degli stati di Forza Quattro. Immagine ottenuta da `https://medium.com/@gillesvandewiele/creating-the-perfect-connect-four-ai-bot-c165115557b0`

É inoltre possibile rappresentare con una formula in logica μ-calculus la proprietà di vittoria "allinea 4 tasselli del colore rosso, prima che lo faccia il giocatore giallo". A partire dalla formula e dal transition system si può generare un PG. Dato il gioco e uno stato iniziale del gioco (codificato in base ad un corrispondente stato del transition system e alla proprietà di vittoria), esisterà una strategia vincente per il giocatore "rosso" a partire da esso, se e solo se esiste una strategia vincente per il giocatore "rosso" per Forza Quattro.

1.3 Perché utilizzare una codifica simbolica

Anche per l'esempio giocattolo mostrato in precedenza, di Forza Quattro giocato su una griglia 6×7; il numero di stati del transition system è nell'ordine di $\sim 4.5 \times 10^{18}$. L'enormità di stati di cui è richiesta la codifica per un PG pone alcuni importanti problemi, da affrontare:

- **memoria**: ridurre il più possibile lo spazio di memoria necessario per rappresentare ciascuno stato risulta assolutamente fondamente, vista la mole di elementi da memorizzare,

- **tempi**: poter eseguire operazioni sugli stati in tempi ragionevoli nonostante ne siano il gran numero, è un aspetto importantissimo da tenere in conto in fase di risoluzione del PG.

In quest'ottica per codificare PG, risulta particolamente indicata allo scopo la rappresentazione simbolica. Tale metodologia, si pone come obiettivo di rappresentare non il singolo stato, ma trovare una codifica (ad esempio un'espressione booleana) che rappresenti insiemi di stati in un formato compresso. Così facendo, in primo luogo, se la codifica è efficiente, si può ottenere un significativo guadagno di spazio, potenzialmente sempre maggiore al crescere della taglia del problema; in secondo luogo, le operazioni da effettuare su ciascun singolo stato possono essere riformulate in termini insiemistici per essere eseguite direttamente sulle formule che rappresentano gli insiemi di stati[Bryant, 1992], vantaggio non da poco se si considera che gran parte dei metodi di compressione dei dati, richiedono che per eseguire operazioni su di essi, vengano prima decompressi.
Nella pratica sostanzialmente, avviene un cambio di prosettiva, non vengono più rappresentati i singoli elementi, ma viene creato un encoding compresso, che rappresenta insiemi di stati; inoltre per eseguire operazioni su di essi, non è necessario decomprimere gli elementi e ricomprimerli nel formato codificato, ma è possibile ideare operazioni che agiscano direttamente sulla rappresentazione degli insiemi.
Al giorno d'oggi il tipo di rappresentazione simbolica più diffuso, per l'encoding di PG fa uso di una particolare struttura dati, denominata Decision Diagram, la

quale è stata utilizzata anche per questo elaborato; per questo motivo, nei successivi capitoli ne verrà fornita una panoramica più esaustiva su come sono fatti i vari tipi di DD, e come possono essere utilizzati per codificare PG.

Capitolo 2

Parity Games

in questo capitolo verranno definiti, in termini più formali, tutti i contenuti relativi ai PG di cui si è voluto dare una prima intuizione, nell'introduzione; in ogni caso, una dissertazione più esaustiva a riguardo può essere trovata al seguente riferimento[Grädel et al., 2002]. Verranno inoltre presentati anche i principi e le nozioni alla base dei due algoritmi trattati nell'elaborato.

2.1 Definizione formale

Un Parity Game può essere definito come una tupla:

$$Game = \langle A, P, W_c \rangle$$

dove A, P e W_c rappresentano rispettivamente, l'arena del gioco, una funzione di mappiung e la condizione di vittoria del PG.

L'Arena: si definisce a sua volta come una tripla:

$$A = \langle V_0, V_1, R \rangle$$

V_0 e V_1 sono due insiemi disgiunti di stati, appartenenti rispettivamente al giocatore 0 e al giocatore 1 definiti come i giocatori del PG, con $V = (V_0 \cup V_1)$ inoltre

è inteso riassumere tutti i vertici del gioco.
$R \subseteq V \times V$ rappresenta la relazione di transizione tra gli stati del PG.

Funzione di priorità: Ad ogni stato inoltre è associata una priorità o ranking da una specifica funzione di mapping $P : V \longrightarrow Pr$ dove $Pr = \{1, 2, 3...n\}$ rappresenta un insieme finito di priorità.

Condizione di vittoria: Dato uno stato v_i possiamo definire i suoi successori come l'insieme $succ(v_i) = \{v^{'} \in V | (v, v^{'}) \in R\}$. Detto ciò, viene definita una giocata π, come una sequenza di stati $\pi = v_0, v_1...$ con $v_i \subseteq V$ e $v_{i+1} \in succ(v_i)$; possiamo dire che per un PG la condizione di vittoria della giocata π per il $giocatore_0/1$ è:

$$max(inf(P(\pi)) = even/odd$$

dove con $P(\pi)$ si vuole intendere la sequenza delle priorità Pr_π associate a ciascuno stato della giocata π, con inf definita come una funzione che data una sequenza di priorità, restituisce il sottoinsieme ∞-$Pr_\pi \in Pr_\pi$ contenente le priorità incontrate infinitamente spesso nella sequenze, con max funzione che dato un set di priorità restituisce, la priorità migliore e infine con $even/odd$ le parità associate rispettivamente al giocatore σ e $\bar{\sigma}$.
In altre parole, un giocatore vince una giocata, quando la priorità massima visitata infinitamente spesso, ha la stessa "parità" del giocatore.

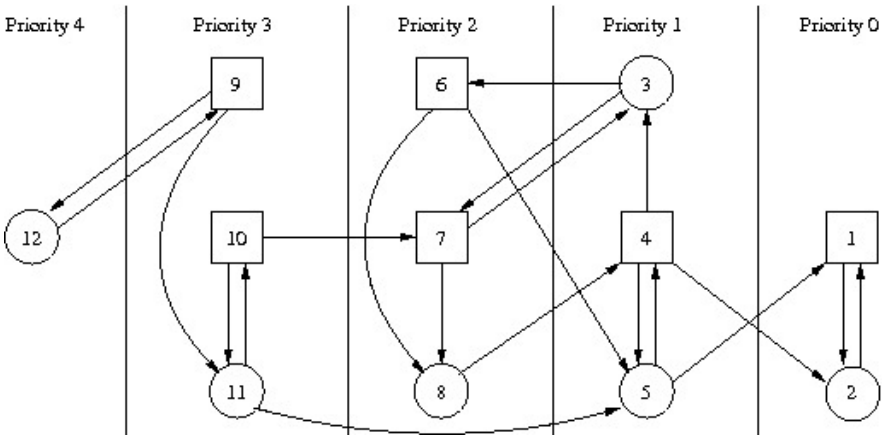


Figura 2.1: Esempio di Parity Game

Strategia: Per determinare cosa effettivamente significhi vincere un PG è necessario introdurre la nozione di strategia. Per strategia di un giocatore $\sigma \in \{0, 1\}$ su un PG, s'intende una funzione parziale $f_\sigma : V_\sigma \to V$ che mappa ogni σ-vertice v_i del gioco in una nuova posizione $f_\sigma(v_i) \in succ(v_i)$. La strategia f_σ si dice vincente per il giocatore σ su $Y \subseteq V$ se per tutte le giocate che partono da un vertice $v \in Y$, il giocatore σ determina tutte le sue mosse con la funzione f_σ e ottiene una vittoria.

A questo punto si può affermare che il giocatore σ vince un PG sul sottoinsieme dei vertici $Y \subseteq V$ se ha una strategia vincente su Y. Infine denominiamo "winning regions" gli insiemi WR_0 e WR_1 che contengono rispettivamente tutti i vertici v per cui il giocatore 0 e il giocatore 1 hanno stategie vincenti su un PG partendo da v stesso.

È molto importante notare, che le strategie per i PG sono memoryless e cioè la funzione $f_\sigma(v_i)$ determina il vertice successivo in cui muovere, basandosi esclusivamente su v_i, non si ha quindi bisogno di memorizzare informazioni relative allo storico delle mosse effettuate fino a quel momento, per determinare una strategia vincente.

Quanto appena detto è formalizzato dal **memoryless determinacy theorem** il quale afferma che:

In ogni PG i vertici sono partizionati nelle winning regions (WR_0, WR_1) per i due giocatori; inoltre essi sono sono in grado di vincere partendo da qualsiasi vertice

della loro WR, usando una strategia di tipo memoryless [Grädel et al., 2002].

2.2 Algoritmi di risoluzione

Gli algoritmi di risoluzione di PG possono essere racchiusi in due grandi categorie:

- algoritmi che tentano di risolvere il gioco nel loro insieme,
- algoritmi che invece cercano di identificare porzioni delle winning regions decomponendo il PG in sottogiochi più piccoli, per poi riaggregare i risultati ottenuti.

In questo elaborato ci concentreremo, sul secondo tipo di risolutore, in quanto più naturalmente si presta ad essere codificato in maniera simbolica. In particolare saranno presi in esame gli algoritmi di Zielonka[Zielonka, 1998] e Priority Promotion[Benerecetti et al., 2016]. Prima di fare ciò però, sono presentati alcuni importanti concetti, utilizzati all'interno degli algoritmi per ottenere le winning regions.

2.2.1 Nozioni fondamentali

σ-predecessore, σ-uscita, σ-attrattore e σ-trappola: dato un giocatore σ, e un sottoinsieme dei vertici del PG: $Y \subseteq V$, e denominato l'altro giocatore come $\bar{\sigma}$, si indica con $pred_\sigma(Y) = \{v \in V_\sigma | succ(v) \cap Y \neq \emptyset\} \cup \{v' \in V_{\bar{\sigma}} | succ(v') \subseteq Y\}$ l'insieme dei vertici del PG, in cui il giocatore σ può forzare il gioco su un vertice di Y in una mossa o perché ha l'opportunità di farlo personalmente da un nodo da esso controllato, o perché l'altro giocatore non ha altra scelta poiché tutti i suoi successori sono in Y. Il σ-attrattore generalizza il concetto di σ-predecessore, non limitandolo ad una sola mossa di gioco, se infatti l' σ-predecessore, viene utilizzato come segue:

1 Attractor(PG, $Y \subseteq V$, $\sigma \in \{0,1\}$)

```
Attr_0 ← Y
repeat
    Attr_i ← Attr_{i-1} ∪ pred_σ(Attr_{i-1})
until Attr_i = Attr_{i-1}
return Attr_i
```

si ottiene l'unione dell'insieme di partenza Y e dei vertici in cui il giocatore σ può forzare il gioco su un vertice di Y con un numero di mosse arbitrario.
Benché la funzione *Attractor* consti dei tre parametri PG, set di vertici Y e giocatore σ; spesso PG e σ saranno fissati dal contesto in cui è utilizzato l'attrattore, per questo motivo nei capitoli successivi ci si riferirà talvolta in forma abbreviata con $Attr(Y)$ all'attrattore operato su un dato PG e rispetto a un dato σ.

Con σ-uscita: $esc_\sigma(PG, Y) = pred_\sigma(V \backslash Y) \cap Y$ s'intende l'insieme dei vertici di Y da cui il giocatore σ può forzare l'uscita da Y in una sola mossa all'interno del gioco PG.
Infine, intuitivamente, per ogni vertice controllato dal $\bar{\sigma}$-giocatore contenuto in $V \backslash Attr(Y, \sigma)$, se tali vertici non sono all'interno di un σ-attrattore ciò vuol dire che l'$\bar{\sigma}$-giocatore possiede sempre tra i suoi successori, un vertice che può evitargli di essere "richiamato" all'interno dell'attrattore. Inoltre per ogni vertice controllato dal σ-giocatore contenuto in $V \backslash Attr(Y, \sigma)$, non esiste alcuna strategia che può in qualche modo portarli a far parte dell'attrattore. In altre parole, dato il complemento di un attrattore, il σ-giocatore ha una strategia per forzare il gioco, a rimanerne all'esterno; l'insieme di tali vertici prende il nome di σ-trappola [Zielonka, 1998], la quale formalmente è definita come:

$$K \subseteq V | \forall v_\sigma \in K, succ(v_\sigma) \cap K = \emptyset \wedge \forall v_{\bar{\sigma}} \in K, succ(v_{\bar{\sigma}}) \cap K \neq \emptyset$$

σ-dominio e σ-quasi dominio: per σ-dominio s'intende un insieme di vertici $Y \subseteq V$ tale per cui esiste una strategia le cui mosse sono in Y ed è vincente per

il giocatore σ. È ovvio constatare che l'unione di tutti gli σ-domini di un PG rappresenta una σ-winning region per quel gioco. La nozione di σ-quasi dominio non fa altro che rilassare, quella di σ-dominio, non imponendo che il giocatore σ debba necessariamente vincere a partire dai suoi nodi. Un quasi dominio infatti è tale se il giocatore σ ha una strategia vincente a meno che[Benerecetti et al., 2016]:

- il giocatore $\bar{\sigma}$ non possa scappare dal quasi dominio,
- l'unica scelta rimasta al giocatore σ sia quella di uscire dal quasi dominio.

Se nessuno di questi due casi si verifica, il quasi dominio, si dice chiuso, in quanto non consente vie di fuga, inoltre come si può facilmente immaginare, un σ-quasi dominio chiuso è a tutti gli effetti un σ-dominio.

sottogioco: Con la nozione, banale, ma non di meno importante di sottogioco si vuole intendere un $PG^{'}$ tale che:
dato un $PG = \langle A, P, W_c \rangle$ con $A = \langle V_0, V_1, R \rangle$; i suoi $V_0^{'}$, $V_1^{'}$ e $R^{'}$ hanno i seguenti vincoli:

$$V_0^{'} \subseteq V_0 \wedge V_1^{'} \subseteq V_1 \wedge R^{'} \subseteq (V^{'} \times V') \wedge R^{'} \subseteq R$$

Definiamo inltre l'operazione spesso usata di restrizione su sottogioco $PG^{'} = PG \backslash Y$ con PG e $PG^{'}$ giochi e Y un set di vertici e in cui $PG^{'}$ è un sottogioco di PG in cui:

$$V^{'} = V \backslash Y \wedge R^{'} \subseteq ((V \backslash Y) \times (V \backslash Y)) \wedge R^{'} \subseteq R$$

2.2.2 Zielonka

l'algoritmo di Zielonka è uno dei principali, affermatisi in letteratura, di tipo *divide and conquer*. Tale algortimo infatti, come idea generale, scompone il PG in sottogiochi sempre più piccoli finché non è in grado di ottenere per essi una winning region. Fatto ciò cerca di fondere insieme le winning region dei sottogiochi, al fine di generare quelle per il PG completo.
Più precisamente, dato un gioco e l'insieme V dei suoi vertici e sia $Y = P^{-1}(max(Pr))$ l'insieme dei vertici del PG con la massima priorità e con $max(Pr)$ priorità della

stessa "parità" di σ; facendo l'attrattore di tale insieme su PG e dividendo il gioco nelle due parti $A = Attr(Y, \sigma)$ e $V^{'} = V \backslash Attr(Y, \sigma)$ è possibile fare alcune importanti osservazioni. Innanzitutto come visto al capitolo precedente si evince che $V^{'}$ è una $\bar{\sigma}$-trappola in quanto complemento di un attrattore. Inoltre all'interno del sottogioco $PG^{'}$ di PG con insieme dei vertici $V^{'}$ ci sono due possibili casi che possono verificarsi:

- $PG^{'}$ è un sottogioco in cui σ è vincente in tutti gli stati,
- $PG^{'}$ è un sottogioco in cui σ non è vincente in tutti gli stati.

Nel primo caso σ vince anche per PG in quanto, se il giocatore $\bar{\sigma}$ forza il gioco a restare in $V^{'}$ perde per definizione. Nel caso in cui invece scelga di non fare ciò, sarà costretto a passare per A; se sceglie, però, di agire in questo modo infinitamente spesso per evitare di restare in $V^{'}$, allora attraverserà infinitamente spesso i vertici con la più alta priorità del gioco e della stessa parità di σ perdendo anche in questo scenario.
Nel secondo caso invece si avrà per $PG^{'}$, due winning region $(WR^{'}_{\sigma}, WR^{'}_{\bar{\sigma}})$ di cui non solo sappiamo che $WR_{\bar{\sigma}} \neq \emptyset$, ma anche che σ non può forzare il gioco ad uscire da $WR^{'}_{\bar{\sigma}}$; per questo motivo è possibile aggiungere $B = Attr(PG, WR^{'}_{\bar{\sigma}}, \bar{\sigma})$ alla $WR_{\bar{\sigma}}$ di PG e continuare risolvendo la parte $V^{'} = V \backslash B$ di cui non si sa ancora per certo chi è il vincitore [Friedmann, 2011].
Di seguito lo pseudocodice dell'algoritmo di Zielonka:

2 Zielonka($PG = \langle A = \langle V_0, V_1, R\rangle, P, W_c\rangle$)

```
if V = ∅ then
    return (∅, ∅)
else
    max_pr ← max(P(V))
    σ ← player_of_parity(max_pr)
    A ← P^{-1}(max_pr)
    A ← Attr(PG, A, σ)
    (WR'_0, WR'_1) ← Zielonka(PG\A)
    if WR'_σ̄ = ∅ then
        (WR_σ, WR_σ̄) ← (WR'_σ ∪ A, ∅)
    else
        Y ← Attr(PG, WR'_σ̄, σ̄)
        (WR'_0, WR'_1) ← Zielonka(PG\Y)
        (WR_σ, WR_σ̄) ← (WR'_σ, WR'_σ̄ ∪ Y)
    end if
end if
return (WR_0, WR_1)
```

2.2.3 Priority Promotion

Così come Zielonka anche Priority Promotion cerca di combinare insieme risultati parziali al fine di ottenere le winning regions dei giocatori; in particolare PP si concentra sul riconoscimento di σ-domini da aggregare man mano al risultato finale. Per fare ciò utilizza alcuni concetti fondamentali che introduciamo di seguito:

- con σ-regione (σ-R) s'intende un σ–quasi dominio in cui l'eventuale set di vertici che forma la $\bar{\sigma}$-uscita, è costituito da soli elementi con la priorità massima nel gioco,

- con spazio dei domini invece viene inteso un ordinamento parziale di *stati* che tiene traccia delle sia delle σ-regioni, man mano che vengono calcolate, sia degli "aggiornamenti" alla funzione di priorità $P : V \longrightarrow Pr$ che vengono fatti durante l'esecuzione dell'algoritmo.

Dato un PG, Priority Promotion inizia col costruire una σ-R raccogliendo in essa i vertici etichettati con priorità massima p e facendone l'attrattore. Alla regione è infine associata la priorità p. A questo punto possono verificarsi due possibilità:

- la regione appena calcolata è chiusa nell'intero gioco PG. Ricordando che una σ-regione è un tipo particolare di σ-quasi dominio e sapendo che è chiusa nell'intero gioco se ne deduce che è anche un σ-dominio, per cui ne viene calcolato l'attrattore per ottenere un σ-dominio massimale che viene aggiunto al risultato finale; dopodiché l'algoritmo riparte alla ricerca del dominio successivo su $PG = PG \backslash Attractor(PG, \sigma\text{-}R, \sigma)$ resettando lo spazio dei domini.

- la regione appena calcolata è aperta nell'intero gioco PG. Se ciò accade ancora una volta possono verificarsi due casi:

  - σ-R è aperta anche in $PG^{'}$, dove per $PG^{'}$ in questo caso viene inteso il sottogioco di PG epurato di tutte le regioni presenti nello spazio dei domini tranne la σ-R appena calcolata(è ovvio constatare che alla prima iterazione oppure subito dopo il ritrovamento di un dominio, non esisteranno regioni nello spazio dei domini diverse da quella corrente e quindi PG risulta uguale a $PG^{'}$).
    in questo caso si prosegue a popolare lo spazio dei domini computando la regione successiva in $PG^{'} \backslash \sigma\text{-}R$.

  - la regione appena calcolata è chiusa in $PG^{'}$. Ciò significa che in PG il $\bar{\sigma}$-giocatore, può forzare l'uscita dalla regione corrente non in $PG^{'}$, ma in almeno un'altra regione, già calcolata, con priorità associata più alta. In questo caso la regione con "priorità di fuga più bassa" (*best exit priority*) e la regione corrente vengono fuse insieme e alla nuova regione computata, viene associata la priorità migliore tra le due. Inoltre poiché la ricerca riparte dalla regione appena creata, viene effettuato un reset nello spazio dei domini di quelle precedentemente computate con priorità associata inferiore [Benerecetti et al., 2016].

Di seguito verrà esposto lo pseudocodice di Priority Promotion insieme ad alcune note per meglio comprenderne la notazione:

- Con la struttura Reg (assimilabile concretamente ad un array associativo) s'intende una struttura che tiene traccia di tutte le regioni create ed ha come indice, la priorità con la quale ciascuna regione è etichettata,

- La funzione *reset* si occupa di resettare il mapping priorità-vertici delle regioni al di sotto del valore di best exit priority,

- La funzione $update_priority_function$ si occupa di aggiornare il mapping priorità-vertici delle regioni e del sottogioco $PG^{'}$ seguendo il mapping delle regioni per i vertici per cui è definito (si ricorda che tale mapping è parziale e dunque non definito su tutti i vertici del gioco), quello del gioco $PG.P$ ove invece non è definito.

3 Priority_Promotion($PG = \langle A = \langle V_0, V_1, R\rangle, P, W_c\rangle$)

1: **while** $PG.V \neq \emptyset$ **do**
2: $\quad max_pr \leftarrow PG.max_pr$
3: $\quad (WR_0^{'}, WR_1^{'}) \leftarrow Searcher(PG, PG, max_pr)$
4: $\quad (WR_0, WR_1) \leftarrow (WR_0^{'} \cup WR_0, WR_1^{'} \cup WR_1)$
5: $\quad PG.V \leftarrow PG.V \backslash (WR_0, WR_1)$
6: **end while**
7: **return** (WR_0, WR_1)

4 Searcher($PG = \langle A = \langle V_0, V_1, R\rangle, P, W_c\rangle$,$PG^{'}$, max_pr)

```
1:  Reg[max_pr] ← PG.V[max_pr]
2:  Reg[max_pr] ← Attr(PG', Reg[max_pr], σ)
3:  while true do
4:      σ ← player_of_parity(max_pr)
5:      if Esc_σ̄(PG, Reg[max_pr]) = ∅ then
6:          return (Attr(PG, Reg[max_pr], σ), ∅)
7:      else
8:          if Esc_σ̄(PG', Reg[max_pr]) = ∅ then
9:              bep ← best_exit_priority(Reg, Reg[max_pr]))
10:             Reg[bep] ← merge(Reg[bep], Reg[max_pr])
11:             reset(Reg, bep)
12:             update_priority_function(PG.P, PG'.P, Reg)
13:             PG'.V ← ∅
14:             for i in PG.min_pr...bep do
15:                 PG'.V ← PG'.V ∪ PG'.P^{-1}[i]
16:             end for
17:             max_pr ← bep
18:         else
19:             PG' ← PG'.V\Reg[max_pr]
20:             max_pr ← PG'.max_pr
21:         end if
22:     end if
23: end while
```

2.3 Codifica Simbolica

Come anticipato nell'introduzione, un modo valido dal punto di vista dell'efficienza, di memorizzare i PG è quello di utilizzare una codifica simbolica che permetta di formulare il problema in forma compressa. A tale scopo è possibile utilizzare una particolare struttura dati denominata Binary Decision Diagram.

Per **Binary Decision Diagrams** s'intende una particolare classe di grafi radicati diretti e aciclici i cui nodi hanno le seguenti caratteristiche:

- ciascun nodo interno è etichettato con una variabile e ha due archi uscenti:
  - un arco di tipo low che, se navigato implica, valore falso, per la variabile da cui è originato,

  - un arco di tipo high che dualmente, se navigato, implica valore vero per la variabile d'origine.

- ciascun nodo terminale ha due possibili valori: true e false.

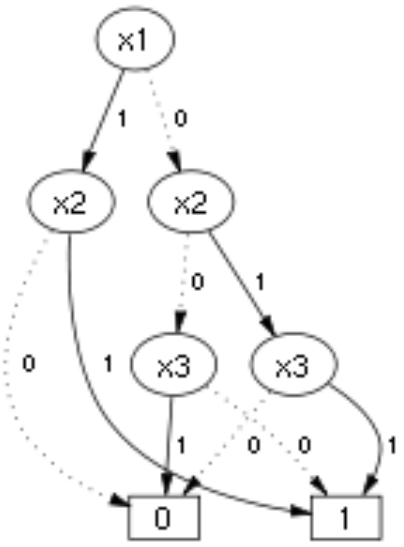
I BDD inoltre tentano di mantenere il loro carico informativo nel modo il più sintetico possibile, per questo motivo ad essi sono applicate delle specifiche regole di riduzione, per contenere il più possibile la dimensione del grafo:

- vengono rimossi tutti i nodi x il cui valore vero e negato porta allo stesso risultato y: tutti gli archi entranti in x vengono ridirezionati, più concisamente, su y;

- vengono condivisi tutti i sotto-alberi identici tra loro: nel caso in cui un nodo x e un nodo y puntino con uno dei loro archi ad uno stesso sotto-albero, non ne vengono mantenute due copie identiche, ma entrambi i nodi radice condivideranno il sotto-albero in comune;

- viene evitata la duplicazione dei nodi terminali: se due nodi terminali hanno lo stesso valore, allora uno dei due viene cancellato e gli archi entranti su di esso, vengono ridirezionati sull'altro.

Binary Decision Diagrams e **Parity Games** Come è possibile intuire dalla loro struttura base, una classica applicazione dei BDD è quella di codificare, in modo sintetico, funzioni booleane, inoltre il teorema di canonicità ci dice che: se le variabili della funzione booleana appaiono in un ordine fissato all'interno di un BDD (Ordered BDD) allora abbiamo la garanzia che la codifica di tale funzione in OBDD è canonica (ovvero unica) [Bryant, 1986].



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figura 2.2: Esempio di codifica di un'espressione booleana tramite BDD

A loro volta espressioni booleane possono essere utilizzate per codificare insiemi: dato infatti un set di n elementi è possibile ottenerne una codifica booleana per ciascuno stato con $\lceil log_2(n) \rceil$ variabili, e legarne l'appartenenza o meno all'insieme, in base al risultato di una funzione booleana. Tale ragionamento è alla base dell' utilizzo di BDD nell'ambito della codifica di PG. Ricordando infatti che i PG sono costituiti, tra le altre cose, da insiemi di vertici su cui sono definite una funzione di priorità e una relazione di transizione, è possibile ideare, di ciascuno di questi elementi costituenti di un PG, una codifica booleana che a sua volta può essere riportata in BDD.

Capitolo 3

Il tool LTSmin

Il Labelled Transition System minimization (in breve LTSmin) è un applicativo open-source, completamente scritto nel linguaggio C, sviluppato presso l'università di Twente (Olanda) che in origine si proponeva di offrire strumenti per la manipolazione di transition system; nel tempo però con l'aggiunta di funzionalità sempre nuove ha finito col diventare un ampio sistema di model-checking. Tra le tante funzionalità offerte, LTSmin è uno dei pochi sistemi di model-checking noti in letteratura, a proporre anche una codifica simbolica di Parity Game e una versione base di risolutore di Parity Game simbolici, modellata sull'algoritmo di Zielonka. Tale applicativo risulta centrale per il presente elaborato di tesi, in quanto si è scelto di implementare tutti gli algoritmi proposti all'interno di tale sistema estendendo e aggiungendovi nuove funzionalità. Per questo motivo, di seguito viene fornita una panoramica dell'infrastruttura, utilizzata da LTSmin per manipolare PG in rappresentazione simbolica.

3.1 Struttura per Parity Games simbolici

Per PG di tipo simbolico, LTSmin adopera una struttura parity_game che al suo interno contiene tutti gli elementi fondanti di un PG (insiemi di stati, relazione di priorità, relazione di transizione). Tale codifica è di alto livello, la struttura infatti è usata come una *black box* e nessuna conoscenza sull'effettiva implementazione interna, viene richiesta al "client" di parity_game.

Internamente LTSmin prevede diversi tipi di backend per la codifica simbolica di PG. Ciascuno di essi, consiste sostanzialmente in una libreria per la rappresentazione simbolica (principalmente varianti di Decision Diagram), la quale viene sfruttata per garantire un'implementazione concreta delle strutture definite in parity_game. Allo stesso modo sono forniti dei wrapper[1] per incapsulare le strutture di alto livello e le operazioni di manipolazione sui PG i quali devono essere ugualmente concretamente implementati da ognuno dei backend.
Di seguito viene illustrata nel dettaglio la struttura *parity_game* con la quale vengono espressi esternamente i PG simbolici in LTSmin:
l'idea alla base della codifica dei PG è quella di fornire tre tipi astratti fondamentali, i quali saranno specializzati internamente da ciascun backend:

- vset_t per la codifica di insiemi di vertici,
- vrel_t per la codifica delle transizioni,
- vdom_t per conservare info utili rispetto al backend che va ad implementare concretamente la codifica simbolica.

3.1.1 I vertici in LTSmin

Come detto in precedenza, la struttura *vset_t* si occupa della codifica degli insiemi di vertici. *parity_game* nello specifico codifica al suo interno i tre insiemi: v, $v_player[0]$, $v_player[1]$ i quali rispettivamente codificano i vertici dell'intero gioco, i vertici controllati dallo $0-$giocatore e i vertici controllati dall' $1-$giocatore. In questo caso abbiamo una ridondanza di informazioni, in quanto il set v risulta direttamente calcolabile dagli altri due. Tale penalità sulle dimenzioni del gioco è sopportata poichè avere questi tre insiemi già pronti, fornisce un disceto vantaggio in fase di risoluzione, visto il loro frequente utilizzo.

La funzione di priorità in parity_game Per assegnare a ciascun vertice del PG una priorità, viene utilizzato l'array di insiemi *v_priority* di tipo *vset_t*.

[1]wrapper: un elemento, il cui scopo è incapsulare e nascondere un altro elemento al suo interno, solitamente viene utilizzato per fornire un interfaccia a chi utilizza il wrapper così da astrarsi da come l'oggetto rivestito è concretamente costituito.

LTSmin esprime il mapping operato dalla funzione, raggruppando i vertici in insiemi divisi per priorità, i quali vengono assegnati alla posizione di *v_priority* che ha per indice la priorità corrispondente a quella dell'insieme. Vengono forniti inoltre *min_priority* e *max_priority* per definire il range di priorità espresso in *v_priority*. Da notare inoltre, che LTSmin risolve giochi di tipo "*minparity*" e cioè in cui la parità migliore è considerata come quella più bassa; per questo motivo la priorità massima è da intendere *min_priority* e non *max_priority* come si potrebbe naturalmente essere portati a pensare.

3.1.2 La relazione di transizione in LTSmin

Nella codifica della relazione di transizione si evince l'attenzione di LTSmin per ottenere una valida performance in termini di risoluzione dei PG e di dimensionalità del problema.
Per codificare le transizioni, viene utilizzata la struttura *vrel_t*. A differenza di quanto si possa banalmente intuire, non viene però generato un unico set di transizioni che codifichi la relazione per intero, ma viene invece introdotto il concetto di gruppo di transizioni.
In pratica viene operata una scomposizione delle transizioni in insiemi indipendenti, ciascuno dei quali è responsabile di una porzione della codifica della relazione di transizione. In questo modo si è costretti a riformulare ogni operazione relativa a tutte le transizioni, per ciascun gruppo, e unire i risultati parziali ottenuti per computare il risultato dell'operazione sulla relazione di transizione complessiva. Ciascuna di queste operazioni però, internamente, si troverà ad agire su un Decision Diagram di dimensioni enormemente ridotte, rendendo l'utilizzo della relazione di transizione, più veloce con l'aumentare della taglia del problema e quindi consentendo al PG di guadagnarne in scalabilità [van Dijk and van de Pol, 2017].
Più nel dettaglio la codifica interna dei gruppi è definita come segue:
come illustrato in [2.3] gli insiemi di stati sono codificati tramite decision Decision Diagrams i quali a loro volta, sono definiti sulle variabili di una funzione booleana che determina l'appartenenza all'insieme. Per quanto riguarda le transizioni, dato un un insieme $vars = \{x_0, x_1, x_2, ...x_n\}$ di n variabili su cui sono definiti

gli insiemi di stati, viene creato contestualmente ad esso un insieme parallelo $vars^{'} = \{x_0^{'}, x_1^{'}, x_2^{'}, ...x_n^{'}\}$ denominato insieme delle variabili primate. L'insieme $vars^{'}$ delle variabili primate è esattamente identico a $vars$, concretamente esso è un semplice duplicato, in cui ciascuna variabile è etichettata con un identificativo diverso rispetto al set di variabili originale. Detto ciò, indichiamo con s uno stato definito su $vars$ e con $s^{'}$ uno stato definito su $vars^{'}$. Possiamo esprimere una transizione $s \rightarrow s^{'}$ con una funzione booleana $f : vars \times vars^{'} \longrightarrow \{0, 1\}$ in cui $f(s, s^{'}) = 1$.
Dati m gruppi di transizioni, LTSmin divide gli insiemi di variabili $vars$ e $vars^{'}$ in m sottoinsiemi $group_vars_i$ e $group_vars_i^{'}$ accoppiati tra loro. I $group_vars_i$ non sono necessariamente disgiunti tra loro, ma per ogni $i, j = 0, ...m - 1$ non può accadere che $group_vars_i \subseteq group_vars_j$, banalmente vale lo stesso anche per i sottoinsiemi $group_vars_i^{'}$. Una volta fatto ciò LTSmin definisce ciascun gruppo di transizioni non su $vars \times vars^{'}$, ma su $group_vars_i \times group_vars_i^{'}$ riducendo di gran lunga la grandezza del suo Decision Diagram. La semantica utilizzata per ciascun gruppo è la seguente:

- per ogni coppia $(x_i, x_i^{'})$ definita nel gruppo, il valore di x_i e di $x^{'}$ corrisponde al valore dell'i-esima variabile delle codifiche degli stati di "partenza" e di "arrivo",

- per ogni coppia $(x_i, x_i^{'})$ non definita nel gruppo, il valore per lo stato di "partenza" e di "arrivo" di x_i e $x_i^{'}$ è invariante $(x_i = x_i^{'})$.

In questo modo indirettamente ogni gruppo è in grado di esprimere transizioni tra stati, pur non essendo definito su tutte le variabili degli stati (e delle corrispondenti variabili primate).
infine l'unione di tutte le transizioni di ciascun gruppo, determina l'intera relazione di transizione.

3.1.3 Criteri implementativi

Le scelte implementative illustrate in precedenza, per la codifica dei Parity Game, sono frutto di importanti valutazioni, riguardo i seguenti aspetti:

- **ridondanza:** al fine di contenere le dimensioni del PG in memoria, è desiderabile non esprimere più volte lo stesso contenuto informativo. Codificare ad esempio gli insiemi V e V_0, V_1 contenenti rispettivamente tutti gli stati del gioco e gli stati controllati dal giocatore 0 e dal giocatore 1, risulta in una grossa ridondanza in quanto $V = V_0 \cup V_1$ e dunque le informazioni da esso fornite sono già determinate dalle altre strutture della codifica,

- **strumenti utili alla risoluzione:** oltre alla dimensione del problema in memoria, è sicuramente molto importante rendere la risoluzione dei PG il più veloce possibile; a tale scopo può essere utile creare delle strutture che aiutino a rendere più veloci le operazioni più spesso eseguite durante la richerca della soluzione,

- **dimensioni contenute delle strutture codificanti:** Come si vedrà più approfonditamente nella seconda parte di questo capitolo, LTSmin utilizza Decision Diagrams per effettuare la sua codifica simbolica; per questo motivo, la velocita di risoluzione di un qualsiasi algoritmo simbolico, è direttamente influenzata dalle dimensioni dei DD sui cui esso si trova ad operare. Al fine di rendere la risoluzione il più veloce possibile, può essere quindi utile non solo pensare una struttura del PG che eviti ridondanze e/o riduca la complessità di operazioni frequentemente effettuate in fase risolutiva, ma che permetta anche di lavorare spesso con DD il cui rapporto tra dimensioni in memoria e numero di stati codificati sia migliore possibile.

Ovviamente risulta impossibile nella pratica ottenere una struttura del PG che sia ottimale in ciascuna delle tre caratteristiche sopraelencate, è importante però effettuare un corretto *trade-off*, al fine di ottimizzare le prestazioni dell'intero sistema.
LTSmin come visto in precedenza, sceglie di sacrificare la ridondanza dei dati, al

fine di velocizzare le operazioni di risoluzione; inoltre, solo per la codifica delle transizioni, opta per una soluzione che tenga conto della dimensione dei Decision Diagram al fine di manipolare DD con grandezze le più contenute possibile.

3.1.4 Le informazioni di supporto

Ciascuna delle librerie per Decision Diagram utilizzate da LTSmin per la codifica simbolica concreta, richiede un bagaglio di informazioni specifiche leggermente diverso, necessario per tenere traccia della strutturazione dei DD e poter effettuare operazioni su di essi. Per rispondere a tale necessità, mantenendo però l'astrazione dalla codifica concreta dei DD per chi effettua operazioni sui PG, LTSmin utilizza la struttura *vdom_t*. Essa, di volta in volta, a seconda della libreria, viene popolata all'avvio del risolutore, con i dati necessari per gestire tutte le operazioni sui DD e per la codifica simbolica degli stati.
Dall'estratto di codice esposto di seguito è possibile inoltre notare due campi: *state_length* e *num_groups*, non direttamente correlati ai concetti di PG. Tali campi però sono generici per una qualsiasi codifica simbolica, per cui, essendo collocati idealmente ad un livello di astazione più alto, rispetto alle informazioni dipendenti dai singoli backend, vengono esposte esternamente al campo *vdom_t domain*.
Per quanto riguarda il loro significato semantico:

- *state_length*: poichè la rappresentazione di insiemi di stati, come espresso in [2.3], avviene tramite la codifica di variabili di funzioni booleane, *state_length* fornisce ai backend l'infomazione sul numero di variabili necessarie per rappresentare ciascun insieme del PG con una funzione booleana e quindi con un BDD.

  LTSmin prevede inoltre, per semplicità che uno stato gli possa essere dato in input con una sequenza di interi piuttosto che con una di valori booleani; in questo caso *state_length* non fornirà direttamente il numero delle variabili su cui è definito il DD, ma un numero di gruppi di variabili booleane, in cui ciascun gruppo sarà designato a codificare uno degli interi nella se-

quenza di input. Quante variabili invece sono designate per la codifica di un gruppo viene espresso tramite un valore di default dai backend che, in caso di bisogno, può essere opportunamente cambiato.

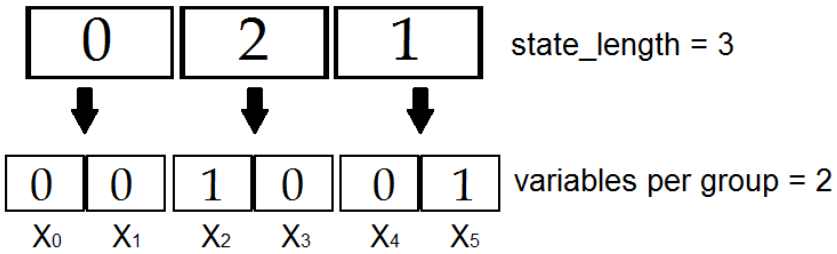


Figura 3.1: Esempio di codifica di uno stato codificato con una sequenza di interi in espressione booleana

- *num_groups*: come illustrato in precedenza LTSmin codifica la relazione di transizione di un PG tramite la definizione di gruppi di relazioni, con tale parametro si tiene traccia della quantità dei gruppi generati.

Di seguito l'estatto di codice che definisce la struttura *parity_game* fornita da LTSmin comprensiva di tutti gli elementi spiegati fin ora:

```
1   typedef struct {
2       vset_t    v;
3       vset_t    v_player [2];
4       vset_t   *v_priority ;
5       int       min_priority;
6       int       max_priority;
7       int       state_length;
8       int      *src;
9       int       num_groups;
10      vrel_t   *e;
11      vdom_t    domain;
12  } parity_game;
```

Codifica di Parity Game in LTSmin

3.2 Backend per la codifica simbolica

3.2.1 Gestione dei wrappers

Essendo LTSmin scritta in C, il sistema dei wrappers per incapsulare i diversi tipi di backend, viene implementato grazie ad un intricato sistema di specializzazione dei tipi e all'utilizzo di puntatori a funzione.
In primo luogo, per rendere più chiara l'esposizione successiva, di seguito è fornito un semplice schema dei file coinvolti nella definizione dei backend e dei wrappers.

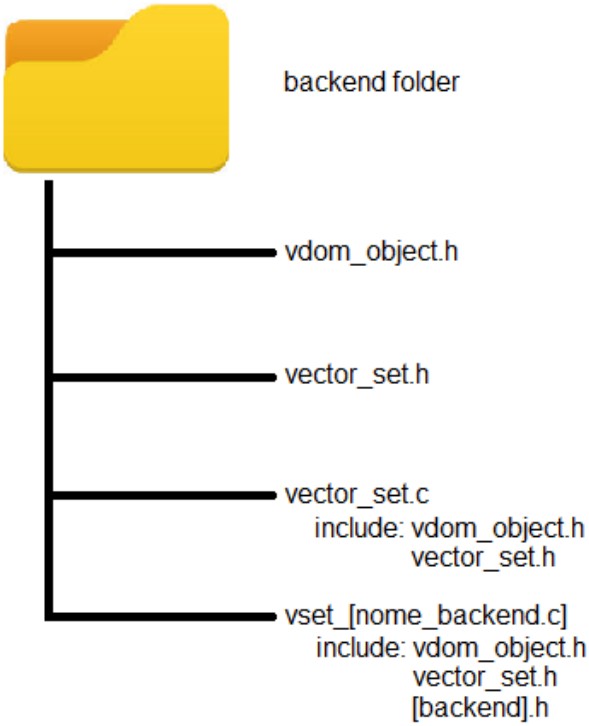


Figura 3.2: File coinvolti nell'implementazione dei backend

L'intero meccanismo si basa sulla definizione dei tipi astratti vrel_t,vset_t e vdom_t, tramite delle strutture appositamente non definite ad alto livello, ma la cui implementazione concreta, è effettuata in modo "custom" e indipendente da ciascuno dei Backend.

File: vector_set.h

```
1    typedef struct vector_relation *vrel_t;
```

```
typedef struct vector_set *vset_t;
typedef struct vector_domain *vdom_t;
```

Definizione dei tipi astratti per la codifica di PG

Inoltre viene definita la struttura *vector_domain_shared* il cui scopo è unicamente quello di contenere dei puntatori a funzione per ogni operazione necessaria a LTSmin.

File: vdom_object.h

```
struct vector_domain_shared {
    vset_t (*set_create)(vdom_t dom,int k,int* proj);
    void   (*set_save)(FILE* f, vset_t set);
    vset_t (*set_load)(FILE* f, vdom_t dom);
    void   (*set_add)(vset_t set,const int* e);
    int    (*set_equal)(vset_t set1,vset_t set2);
    ...
};
```

Implementazione di *vector_domain_shared*

Tale struttura è quindi inclusa in ciascuna implementazione concreta di *vdom_t domain*

File: vset_sylvan.c

```
struct vector_domain {
    struct vector_domain_shared shared;

    BDDVAR *vec_to_bddvar;
    BDDVAR *prime_vec_to_bddvar;
    size_t bits_per_integer;
    BDD universe;
    BDD prime_universe;
};
```

Implementazione "custom" di *vdom_t domain* del backend sylvan

quando LTSmin opera utilizzando la codifica simbolica, richiede di sapere in input, quale dei backend utilizzare. In base a ciò il sistema sceglie ed inizializza il *vdom_t domain* corrispondente. Durante questa fase di inizializzazione, il backend prescelto chiama una sua funzione denominata *dom_set_function_pointers*, la quale associa a ciascun puntatore a funzione, la sua implementazione concreta.

File: vset_sylvan.c

```
1  static void dom_set_function_pointers(vdom_t dom) {
2      dom->shared.set_create=set_create;
3      dom->shared.set_destroy=set_destroy;
4      dom->shared.set_add=set_add;
5      dom->shared.set_add_subset=set_add_subset;
6      dom->shared.set_member=set_member;
7      dom->shared.set_is_empty=set_is_empty;
8       ...
9  };
```

Implementazione di *dom_set_function_pointers* per il backend sylvan

A questo punto, poichè *vdom_t domain* è sempre disponibile, in quanto ne viene conservata una copia in cascuna implementazione concreta di *vset_t* e *rel_t*, richiamando all'interno in una funzione contenitore *domain→shared.nome_funzione* è possibile accedere all'implementazione concreta del backend prescelto, senza doverlo "conoscere" esternamente.

File: vector_set.c

```
1  vset_t vset_create(vdom_t dom, int k, int* proj){
2      return dom->shared.set_create(dom,k,proj);
3  }
```

Implementazione del wrapper *vset_create*

Di seguito l'elenco delle firme dei wrapper necessari per operare sui PG corredate di una breve descrizione dell'operazione:

```
/** Crea il dominio specifico per il backend
  n lunghezza del vettore per codificare uno stato.
  impl identificativo del backend*/
extern vdom_t vdom_create_domain(int n, vset_implementation_t impl);
extern vdom_t vdom_create_domain_from_file(FILE *f, vset_implementation_t
    impl);
/** Crea un insieme di stati/transizioni vuoto.
  k numero di variabili su cui il set di stati/transizioni viene definito (se
    negativo indica che il set viene definito su tutte le variabili).
  proj array di indici delle variabili su cui viene definito il set (se k ha
    valore negativo pu essere lasciato con valore NULL) */
extern vset_t vset_create(vdom_t dom,int k,int* proj);
extern vrel_t vrel_create(vdom_t dom,int k,int* proj);
/** Salva/Carica set di stati o di transizioni su/da un file.
    f file dove salvare/da cui caricare
    set insieme da salvare */
extern void vset_save(FILE* f, vset_t set);
extern vset_t vset_load(FILE* f, vdom_t dom);
extern void vrel_save(FILE* f, vrel_t rel);
extern void vrel_load(FILE* f, vrel_t rel);
/** Aggiunge uno stato all'insieme.
    e array in cui viene codificato lo stato */
extern void vset_add(vset_t set,const int* e);
/** Aggiunge una transizione all'insieme.
    src array in cui viene codificato lo stato di partenza della transizione
    dst array in cui viene codificato lo stato d'arrivo della transizione*/
extern void vrel_add(vrel_t rel,const int* src,const int* dst);
/** Verifica che uno stato appartenga a un insieme.*/
extern int vset_member(vset_t set,const int* e);
/** Verifica che due set siano uguali*/
extern int vset_equal(vset_t set1,vset_t set2);
/** Verifica che un set sia vuoto*/
extern int vset_is_empty(vset_t set);
/** Svuota un insieme*/
extern void vset_clear(vset_t set);
/** Svuota un insieme e lo dealloca*/
extern void vset_destroy(vset_t set);
```

```
35  /** Copia il contenuto di un set in un altro.*/
36  extern void vset_copy(vset_t dst,vset_t src);
37  /** Unisce i set in input e salva il risultato in dst */
38  extern void vset_union(vset_t dst,vset_t src);
39  /** Fa l'intersezione dei set in input e salva il risultato in dst*/
40  extern void vset_intersect(vset_t dst,vset_t src);
41  /** Sottrae a dst gli stati presenti in src e salva il risultato in dst*/
42  extern void vset_minus(vset_t dst,vset_t src);
43  /** Conta il numero di nodi del DD e il numero di stati/transizioni del PG
44    nodes conterr il numero di nodi
45    elements conterr il numero di stati/transizioni del PG*/
46  extern void vset_count(vset_t set,long *nodes,bn_int_t *elements);
47  extern void vrel_count(vrel_t rel,long *nodes,bn_int_t *elements);
48  /** Codifica succ(src) basandosi sulle transizioni contenute in rel e salva il
      risultato in dst*/
49  extern void vset_next(vset_t dst,vset_t src,vrel_t rel);
50  /** Codifica pred(src) i cui stati sono contenuti nel set univ basandosi sulle
      transizioni contenute in rel e salva il risultato in dst*/
51  extern void vset_prev(vset_t dst,vset_t src,vrel_t rel, vset_t univ);
```

3.2.2 I tipi di Backend

tra le librerie utilizzate come backend simbolici per LTSmin possiamo annoverare:

- **BuDDy:** un importante package per la gestione dei Decision Diagram classici, corredato di tutte le principali operazioni effettuabili su BDD e fornito di alcuni importanti accorgimenti ad oggi spesso utilizzati su BDD come: il riordino delle variabili (al fine di ridurre la grandezza dei BDD) e un sistema di *garbage collection* automatizzato [Nielsen, 2006].

- **sylvan:** la particolarità di questo package è quella di essere sviluppato per eseguire le operazioni sui BDD in modo parallelo, accelerando così i tempi d'esecuzione. Nella pratica sylvan scompone al suo interno ciascuna operazione, in sotto task, i quali sono inseriti in una coda dalla quale i worker

thread designati per la parallelizzazione possono attingere i loro compiti per dare contributo al calcolo delle operazioni [van Dijk and van de Pol, 2017].

- **lddmc:** sylvan è un package open source e quindi aperto ad estensioni. lddmc appunto è un estensione di sylvan che conserva la parallelizzazione delle operazioni, ma invece di operare su classici BDD opera su List Decision Diagram. Per esprimere il concetto di List Decision Diagram, è necessario introdurre precedentemente una distinzione tra due tipi di BDD:
  - i **BDD completamente ridotti** vietano la loro interno la presenza di nodi in cui tutti gli archi uscenti puntano allo stesso nodo del livello successivo. In questo modo l'altezza del BDD è soggetta a contrazione, se infatti tutti i nodi che normalmente comparirebbero ad un certo livello dell'albero vengono sottointesi, poiché violano la proprietà appena espressa, quel livello viene sostanzialmente cancellato. Se da una parte ciò offre vantaggi in termini di memoria occupata, dall'altra rende più complessa la gestione dei BDD in quanto la loro altezza non è direttamente determinabile dalle variabili su cui sono definiti;
  - i **BDD parzialmente ridotti** invece sottointendono nodi come i completamente ridotti solo se questo non comporta una contrazione dei livelli (e quindi dell'altezza complessiva) del BDD.

Inoltre è importante definire anche i cosiddetti Multivalued Decision Diagrams. Possiamo definire gli MDD come una naturale estensione dei BDD translati dal dominio booleano a quello intero. Se infatti i BDD prevedono da ogni nodo non terminale l'uscita di esattamente due archi, i nodi degli MDD prevedono n archi in uscita con valore intero compreso tra 0 ed $n-1$. Allo stesso modo i nodi terminali non avranno solo due valori possibili, ma n.

Parimenti ai BDD parzialmente e completamente ridotti esistono banalmente anche gli MDD parzialmente e completamente ridotti.

I List Decision Diagram su cui è definito lddmc, altro non sono che una rappresentazione tramite linked-list di MDD parzialmente ridotti [van Dijk and van de Pol, 2017

- **ATerm:** il package ATerm esula in parte da quanto fin ora spiegato in quanto: è si una libreria per la rappresentazione simbolica, ma che invece di basarsi su DD opera su una differente struttura di "termini annotati". Per un maggiore approfondimento di cosa siano esattamente i termini annotati si invita a consultare [de jong et al., 2000]. Brevemente, gli AT (*annotated terms*) si possono intendere come un tipo di dato astratto, che appartiene alla famiglia degli alberi sintattici annotati, atto alla rappresentazione concisa di dati che abbiano una struttura ad albero. Negli AT ogni elemento è denominato termine ed ogni informazione utile, ma non direttamente ottenibile dai termini è raccolta in pezzi di informazioni di corredo denominate annotazioni.

3.3 Da LPS a Parity Game

In questa sezione, si vuole fornire un'idea di base di come è possibile, tramite LTSmin, passare tra varie rappresentazioni, al fine di ottenere dalle specifiche di un sistema e da una proprietà modellata su quel sistema, un Parity Game. Per una spiegazione più esaustiva dei procedimenti attuabili in LTSmin e della sintassi dei vari formati, si rimanda a [Kant and van de Pol, 2014] su cui la seguente sezione si basa.

Nell'introduzione [1.1] si è già parlato di come, a partire da un *transition system* e da una formula in logica μ-calculus che esprime una proprietà sul sistema, si può ottenere un PG. In realtà tale approccio, benchè possibile, risulta difficoltoso a causa della natura stessa dei *Transition Systems*. Un *TS* infatti, si può definire come un set di configurazioni di un sistema e una collezione di transizioni tra una configurazione e l'altra. Prendendo in esempio il gioco forza quatto (figura: 3.3) si può immediatamente immaginare come la quantità di configurazioni e transizioni, pur per un sistema così semplice, risulta molto elevata; basti pensare a tutte le possibili combinazioni di gioco attuabili sulla *board*.

In qualche modo questo approccio fornisce una codifica totalmente esplicita di come un sistema agisce, generando tuttavia un grafo di enormi dimensioni, che in

fase di traduzione in PG, funge da "collo di bottiglia" per i tempi di generazione. Un modo alternativo di procedere, è quello di non passare dalla codifica in *Transition System*, ma esprimere il comportamento di un sistema, tramite una serie di regole di funzionamento.

LPS: in LTSmin, è possibile codificare le specifiche di un sistema in Linear Process Specification (LPS) tramite un linguaggio algebrico dei processi denominato mCRL2. Le istruzioni in mCRL2 possono avere principalmente i seguenti elementi:

- un'azione di processo, la quale può avere o non avere argomenti,
- un operatore di concatenazione di azioni,
- un costrutto condizionale, per influenzare il corso degli eventi sui processi,
- un operatore *sum* per scorrere una selezione di processi.

Riprendendo l'esempio di prima possiamo, a titolo dimostrativo, definire alcune specifiche del sistema forza quattro come segue:

- una casella può essere vuota, gialla o rossa,

```
sort Piece = struct Red | Yellow | None;
```

- l'azione At(x,y,B) prende in ingresso due coordinate e una Board di gioco. Se tali valori in input sono validi, tramite il costrutto condizionale "→", restituisce il valore *Piece* che occupa la posizione indicata,

```
map At:Nat#Nat#Board->Piece;
```

- la specifica della condizione di vittoria (quattro pezzi in fila dello stesso tipo) per colonna strutturata come segue:
  - vengono scorse tutte le posizioni di gioco grazie all'operatore *sum*,

– ogni posizione è concatenata con l'operatore "." alla verifica della condizione di vittoria,

– se la condizione è vera, viene dichiarata la vittoria con l'azione *Wins*.

```
1  proc sum x,y:Pos .(At(x,y,b)==At(x,y+1,b) &&
                      At(x,y+1,b)==At(x,y+2,b) &&
                      At(x,y+2,b)==At(x,y+3,b))
2                         -> Wins(At(x,y,b))
```

μ-calculus: una volta espresse le specifiche di un sistema tramite LPS, è necessario esprimere una proprietà su di esso. A tale scopo viene utilizzata la logica modale del primo ordine μ-calculus, la quale è definita come un estensione della logica di Hennesy-Milner corredata con gli operatori di punto fisso e quantificatori [Kant and van de Pol, 2014]. Le formule di tale logica sono definite come segue:

$$\phi = b \mid \bar{b} \mid \phi \wedge \phi \mid \phi \vee \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi \mid Qd : D.\phi \mid Z(e) \mid \rho Z(x : D = d).\phi$$

dove b è un espressione booleana, $Q \in \{\forall, \exists\}$, $\rho \in \{\mu, \nu\}$ è un punto fisso minimale (μ) o massimale (ν) con la restrizione che ogni variabile proposizionale Z occorre positivamente in γ in un'equazione del tipo $\rho Z(x : D = d).\gamma$ e α è un linguaggio di specifica di predicati sulle azioni.
Riprendendo il classico esempio di forza quattro, tramite il μ-calculus è ad esempio possibile esprimere la proprietà: "il giocatore giallo ha una strategia vincente" come segue:

$$\mu X.[Wins(Red)]false \wedge \langle Move\rangle(\langle Wins(Yellow)\rangle true \vee [Move]X)$$

PBES Una volta codificata sia la proprietà, tramite la formula del μ-calculus che le specifiche del sistema, tramite LPS, LTSmin combina questi due elementi per ottenere una nuova codifica in Parameterised Boolean Equation System.
Tale formato non è altro che un sistema di equazioni booleane, formalizzate nella

logica del primo ordine. Un sistema PBES può essere poi normalizzato in Parametrised Parity Game (PPG) che è ancora un PBES, ma in cui ogni equazione è una congiunzione o una disgiunzione. Riprendendo l'esempio di forza quattro ed in particolare, la formula del μ-calculus riportata in precedenza, e facendo le seguenti assunzioni:

- siano $b_1 ... b_{49}$ le variabili che rappresentano ciascuna delle 6×7 caselle della *board* di forza quattro,
- sia p la variabile che codifica il turno di un giocatore,
- sia la formula esaminata in precedenza, scomposta nel seguente sistema di formule del μ-calculus:

$$\mu X = A \wedge B$$

$$\mu B = \langle Move(Yellow) \rangle C$$

$$\mu A = [Wins(Red) false]$$

$$\mu C = \langle Wins(Yellow) \rangle true \vee [Move(Red)]X$$

Il PBES risultante assume la seguente forma:

$$\texttt{pbes}\ \mu X(b_1, ...b_{49}, p) = A(b_1, ...b_{49}, p) \wedge B(b_1, ...b_{49}, p)$$

$$\mu A(b_1, ...b_{49}, p) = A_1(b_1, ...b_{49}, p) \wedge A_2(b_1, ...b_{49}, p) \vee ...$$

$$...$$

$$\mu B(b_1, ...b_{49}, p) = B_1(b_1, ...b_{49}, p) \vee B_2(b_1, ...b_{49}, p) \vee ...$$

$$\mu B_1(b_1, ...b_{49}, p) = (b_1 = -) \wedge C(b_1, ...b_{49}, Opponent(p))$$

$$...$$

$$\mu B_{49}(b_1, ...b_{49}, p) = (b_{49} = -) \wedge C(b_1, ...b_{49}, Opponent(p))$$

$$\mu C(b_1, ...b_{49}, p) = ...$$

$$...$$

$$\texttt{init}\ X(-, -, ..., -, -, -, Yellow);$$

Da questo nuovo formato, viene infine effettuata la traduzione e instanziazione in Parity Game.

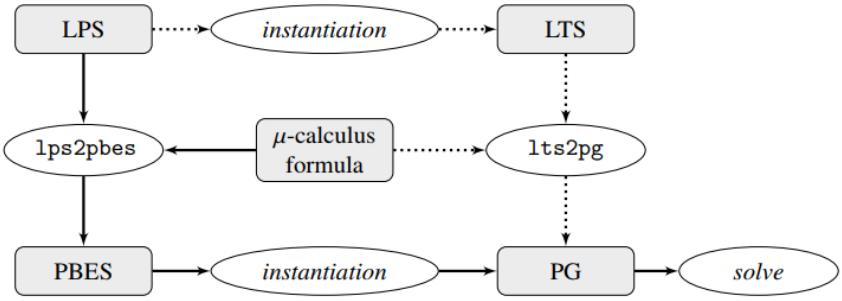


Figura 3.3: Mappa delle trasformazioni possibili per ottenere un Parity Game simbolico [Kant and van de Pol, 2014]

Capitolo 4

La libreria CuDD

4.1 Caratteristiche

Il Colorado university Decision Diagram package (in breve CuDD) è forse la più affermata libreria per la manipolazione di Binary Decision Diagrams (nonchè di due sue varianti quali gli Algebraic Decision Diagrams e i Zero Suppressed Decision Diagrams). CuDD è una libreria Open Source, di basso livello, completamente scritta in C, la quale espone esternamente un numero molto esteso di operazioni con le quali agire su BDD. Di seguito viene riportata una breve introduzione alle caratteristiche più importanti di tale libreria, basata sulla documentazione ufficiale disponibile a [Somenzi, 2016].

4.1.1 La struttura base

I BDD implementati con CuDD hanno alla loro base, la struttura *DdNode*; la quale contiene tutte le informazioni relative ai singoli nodi dei BDD come: id del nodo, conteggio dei riferimenti del nodo e puntatori ad altri *DdNode* per permettere la connessione tra di essi. Internamente tali elementi sono immagazinati in un'*hash table* denominata *unique table* il cui compito è quello di garantire l'unicità della coppia nodo "padre"-nodo "figlio", verificando in questo modo la canonicità dei DD [2.3]. Tale tabella principale è a sua volta divisa in sottotabelle, ognuna delle quali è demandata a contenere i nodi generati a partire da una specifica

variabile.
la *unique table* è conservata all'interno di un *DdManager* al quale è necessario fare riferimento, ogni volta che si vuole effettuare un'operazione sui BDD. All'interno del *DdManager* infatti, sono presenti anche tutte le informazioni generali relative all'ambiente CuDD, come: sequenza delle variabili dopo eventuali riordini, quantità di memoria dedicata alle varie strutture, numero di variabili e così via.

4.1.2 Tecniche per il miglioramento della performance

Uno degli obiettivi di CuDD è ovviamente quello di preservare la performace dell'intero sistema, tenendo il consumo di memoria più basso possibile. Per fare ciò si avvale di alcuni importanti strumenti:

- **Cache:** gran parte delle funzioni per manipolare i BDD sono, per questioni di efficienza, di tipo ricorsivo. Per far si che tale metodologia sia efficace, CuDD prevede l'utilizzo di una tabella ausiliaria per il salvataggio dei risultati calcolati. Tale tabella viene appunto denominata Cache all'interno del sistema. Normalmente essa è tenuta, almeno inizialmente, di dimensioni ridotte e, a meno che l'utente che utilizzi CuDD non imposti diversamente, le dimensioni vengono decise autonomamente dal sistema. Man mano che vengono effettuate operazioni, la Cache aumenta gradualmente di dimensioni. Per impedire un degradamento dell'efficienza, tale crescita avviene però fino ad un valore di tolleranza massimo, oltre il quale la Cache non aumenta in grandezza.

- **Shared BDD:** Al fine di risparmiare memoria CuDD ricorre ad alcuni importanti accorgimenti, uno di questi è l'utilizzo dei cosidetti "BDD condivisi". Ogni volta che viene richiesta la generazione di un BDD, si verifica se parte di esso sia già stato creato all'interno di un altro precedentemente generato; se ciò avviene, la parte comune non viene inutilmente duplicata, ma è riutilizzata da ambo i Decision Diagrams.

- **garbage collection:** CuDD permette di utilizzare la tecnica della *garbage collection* per risparmiare ulteriormente in termini di spazio. In particolare, se abilitato a farlo, il sistema tenta di liberare periodicamente e in autonomia la memoria allocata per strutture che, durante l'esecuzione del programma, non verranno più utilizzate. Tale attività è svolta automaticamente, l'utente però può controllarla nel seguente modo:

  – può disattivare o attivare la garbage collection

```
void Cudd_DisableGarbageCollection(DdManager *dd)
void Cudd_EnableGarbageCollection(DdManager *dd)
```

  – può forzare l'esecuzione della garbage collection in uno specifico punto

```
int cuddGarbageCollect(DdManager *unique, int clearCache)
```

Inoltre sempre all'utente è affidata la designazione di cosa può essere "raccolto" dal garbage collector e cosa no. Come detto in precedenza, infatti, ciascun nodo è corredato con un conteggio dei riferimenti. Tramite alcune funzioni di referenziazione e dereferenziazione è possibile incrementare o decrementare i conteggi dei riferimenti dei nodi creati. Nel momento in cui il contatore di un nodo raggiunge lo zero a seguito di una dereferenziazione, esso viene automaticamente dichiarato "morto". Ogni volta che il *garbage collector* interviene, provvederà dunque a reclamare e liberare la memoria di tutti i nodi dichiarati morti. Da notare è che nel caso in cui un BDD venga referenziato e dereferenziato lo stesso numero di volte, non è detto che il conteggio dei riferimenti di ciascun nodo del suddetto BDD sia a zero. Infatti, a causa dell'utilizzo della politica degli "shared" BDD, un nodo può essere condiviso da altri Decision Diagrams, e aver ricevuto una referenziazione da uno di essi.

4.2 Integrazione di CuDD in LTSmin

Caso stano è che CuDD, probabilmente la libreria più affermata e utilizzata dalla comunità scientifica per la manipolazione di Binary Decision Diagrams, non sia stata contemplata tra i Backend usati da LTSmin per codificare simbolicamente PGs. Al fine di verificare le performance degli algoritmi in esame anche su un codificatore di BDD di riferimento com'è CuDD, parte del lavoro di questo elaborato ha previsto la sua integrazione tra i Backend simbolici utilizzati in LTSmin.

4.2.1 Configurazione e installazione

Come la stragrande maggioranza dei tool di medie e grandi dimensioni, scritti in C, per i sistemi operativi Unix; LTSmin utilizza i file di *configure* e di *makefile* per la sua installazione.

configure: lo script di configurazione si occupa di verificare che sul sistema in essere sia presente un compilatore C adeguato e di rendere l'applicativo installabile sullo specifico sistema in cui si trova. Gran parte del suo lavoro consiste quindi nella verifica e nel *matching* dei *path* di tutte le librerie da cui dipende il sistema da installare.

makefile: una volta finito il lavoro del file di configurazione, il software viene successivamente costruito invocando il *makefile*, il cui scopo principale è quello di eseguire effettivamente tutte le operazioni di *linking* e di costruzione necessarie ad ottenere gli "eseguibili" del prodotto.
Grandi applicazioni solitamente non prevedono un unico makefile, ma scompongono la fase di costruzione in più makefile diversi, ciascuno con i propri compiti specifici. Inoltre LTSmin, non include un makefile precostituito, ma piuttosto un *makefile.in*: una sorta di template per *makefile*, che al passo di configurazione viene utilizzato per generare i makefile veri e propri.

L'integrazione di CuDD in LTSmin quindi si è basata su una serie di passi fondamentali:

- in primo luogo è stato scaricato e installato CuDD all' interno della cartella principale di LTSmin, dove sono presenti anche gli altri backend utilizzati dal tool,
- in secondo luogo è stato modificato il configure file, per permettere all'applicativo di ritrovare le librerie di CuDD da utilizzare al suo interno,
- è stato creato il file vset_cudd.c per l'implementazione dei wrapper "custom" nella sintassi di CuDD,
- infine sono stati modificati il makefile responsabili di costruire i backend simbolici, al fine di prevedere e includere anche la *build* di *vset_cudd.c* tra le librerie da costituire e integrare nel sistema.

4.2.2 Implementazione degli elementi di base

Tramite i passi spiegati in [4.2.1] si è potuto aggiungere CuDD tra le librerie da cui il sistema è dipendente per la rappresentazione simbolica. Ovviamente però, per far si che CuDD fosse adoperato come tutti gli altri backend simbolici come illustrato in [3.2] è stato necessario:

- implementare le strutture specializzate, *vector_set*, *vector_relation* e *vector_domain* che estendono rispettivamente i tipi astratti dei wrapper *vset_t*, *vrel_t* e *vdom_t* con un implementazione concreta per ciascun backend;
- implementare concretamente le funzionalità atte ad operare su parity_game;

utilizzando le strutture, le primitive e la sintassi proprie dalla libreria.
Per meglio comprendere come si comportano, le strutture che si andranno a presentare, è qui definito, un esempio banale di PG che verrà usato per rappresentare man mano, quanto sarà spiegato di seguito:

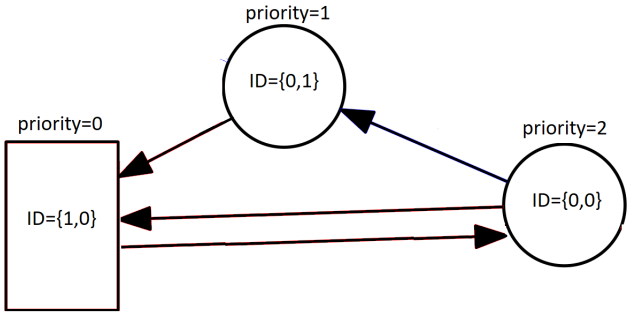


Figura 4.1: Esempio di parity game

Il PG nell'esempio è costituito dai tre nodi {(0,0), (0,1) e (1,0)} rispettivamente con priorità 2,1 e 0. Il sottoinsieme di nodi {(0,0), (0,1)} è controllato dal *player_0* mentre {(1,0)} dal *player_1*. Ciascun nodo ha un identificativo formato da una sequenza di due interi compresi tra zero e uno, codificabile con una variabile booleana (X_0, X_1). Inoltre sono presenti quattro transizioni {((0,0),(1,0)), ((0,0),(0,1)), ((1,0),(0,0)), ((0,1),(1,0))}. Ogni transizione è identificabile con una sequenza di quattro viariabili $(X_0, X_1), (X_0', X_1')$. Infine una piccola nota sull'arco ((1,0),(0,0)): esso infatti può essere definito anche sulla sola coppia $((X_0)(X_0'))$ in quanto tra X_1 e X_1', nel suo caso, vi è invarianza [3.1.2] $(X_1 = X_1' = 0)$.

4.2.2.1 vector_set:

le due strutture fondanti per operare con CuDD sono [4.1.1]: *DdNode* per codificare i BDD e *DdManager* che è il "contenitore" di tutte le info necessarie per operare sui BDD, per questo motivo, l'implementazione concreta di *vset_t* deve almeno contenere il manager e il *DdNode* che codifichi l'insieme degli stati del PG.

Per effettuare inoltre molte delle operazioni sui BDD, CuDD richiede una serie di informazioni ausiliarie, specifiche per ogni singolo Decision Diagram. Tali informazioni, poichè immediatamente ottenibili alla creazione del set, vengono salvate anch'esse nella struttura *vset_t* per evitare che in futuro si abbia bisogno di ricalcolarle, rallentando l'esecuzione dei *solver*. Le informazioni aggiuntive necessarie sono le seguenti:

- *vec_to_bddvar*: poichè molte funzionialità richiedono di sapere ciascun set su quali variabili è definito, ciscuna di esse viene salvata in un array di variabili apposito. Nel caso dell'esempio in figura: 4.1 il set di stati dell'intero gioco avrà l'array vec_to_bddvar=$[X_0, X_1]$ con X_0 e X_1 rappresentate tramite $DdNode*$,

- *prime_vec_to_bddvar*: lo stesso discorso fatto al punto precedente, è valido anche per la versione rinominata (primata) delle variabili, in quanto le operazioni come il calcolo dei predecessori di uno stato, che coinvolgono sia transizioni che stati necessitano di sapere qual'è la versione primata delle variabili su cui il set è definito. Nel caso dell'esempio in figura: 4.1 il set di stati dell'intero gioco avrà l'array vec_to_bddvar=$[X_0', X_1']$ con X_0' e X_1' rappresentate tramite $DdNode*$,

- *projection*: tutte le operazioni che necessitano di una quantificazione, hanno bisogno di sapere quali sono le variabili su cui il set invece non è definito, una congiunzione di queste variabili, viene conservata nel BDD projection. Nel caso dell'esempio in figura: 4.1 per qualsiasi insieme definito projection=$false$ in quanto tutti i vertici del PG sono definiti su tutte variabili definite per gli stati (nello specifico X_0 e X_1)

- *variables*: talvolta per alcune operazioni, CuDD richiede le variabili su cui un set è definito, non separatamente, ma in congiunzione tra loro in un unico BDD. Per questo motivo, pur di non ricalcolare tale congiunzione ogni volta quando richiesto, viene tenuta una copia delle variabili su cui il set è definito, anche in questo formato. Nel caso dell'esempio in figura: 4.1 in *variables* è contenuto il $DdNode*$ che codifica il BDD:

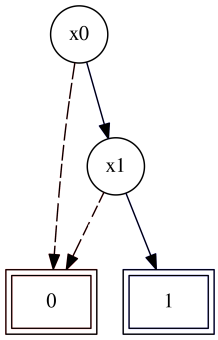


Figura 4.2: BDD con concatenazione delle variabili X_0 e X_1

- *prime_variables*: per gli stessi motivi delle variabili non rinominate illustrati al punto precedente, viene creato anche un duplicato delle variabili primate in un unico BDD.Nel caso dell'esempio in figura: 4.1 in *prime_variables* è contenuto il $DdNode*$ che codifica il BDD:

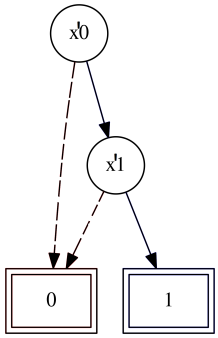


Figura 4.3: BDD con concatenazione delle variabili X_0' e X_1'

- infine viene fornito *vector_size*, per poter conoscere la dimensione degli array *vec_to_bddvar* e *prime_vec_to_bddvar*. Nel caso in figura: 4.1 banalmente *vector_size*=2

Di seguito il codice dell'implementazione di *vector_set* (estensione concreta di *vset_t*) illustrata in precedenza:

File: vset_cudd.c

```
1   struct vector_set {
2       vdom_t      dom;
3       DdManager *manager;
4       DdNode     *dd;
```

```
5        size_t      vector_size ;
6        DdNode     **vec_to_bddvar;
7        DdNode     **prime_vec_to_bddvar;
8        DdNode      *projection;
9        DdNode      *variables;
10       DdNode      *prime_variables;
11   };
```

Definizione concreta di *vset_t* in CuDD

4.2.2.2 vector_relation:

Gran parte delle informazioni necessarie per i set sono utilizzate anche per la codifica delle transizioni in *vrel_t*. Si ricorda che chiaramente il BDD su cui è definito la transizione, comprenderà non solo le variabili "normali", ma anche le primate, al fine di indicare con esse lo stato "d'arrivo" della transizione.
Benchè strutturalmente molto simile, *vrel_t* necessita rispetto a *vset_t* delle seguenti informazioni aggiuntive:

- *all_variables*: come per le operazioni sui set, anche le operazioni con le transizioni necessitano spesso di conoscere le variabili su cui esse sono definite, in forma di congiunzione di variabili. *all_variables* contiene tale congiunzione sia di quelle primate che non. Nel caso in figura: 4.1 si può definire un insieme di transizioni che contenga il solo l'arco ((1,0),(0,0)), senza X_1 e X_1' in quanto invarianti. Per questo motivo *all_variables* sarà così costituito:

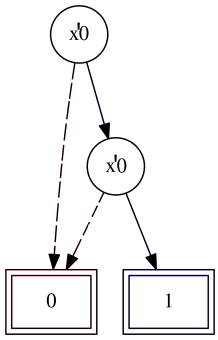


Figura 4.4: BDD con concatenazione delle variabili X_0 e X_0' su cui è definita la transizione ((1,0),(0,0)) presente in figura 4.1

- *invariant_variables*: come spiegato in [3.1.2] le variabili su cui non sono definite le transizioni, hanno in se il valore semantico di invarianza di valore tra stato di "partenza" e stato "d'arrivo", tale informazione sulle variabili, che esprimono questa proprietà, è conservata nel BDD *invariant_variables*. Nel caso in figura: 4.1 nel insieme di transizioni che contenga il solo l'arco ((1,0),(0,0)), *invariant_variables* sarà così costituito:

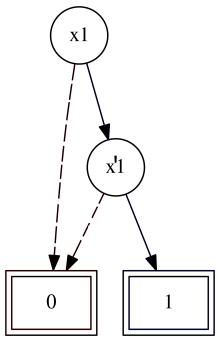


Figura 4.5: BDD con concatenazione delle variabili X_1 e X_1' su cui è invariante la transizione ((1,0),(0,0)) presente in figura 4.1

Di seguito il codice dell'implementazione di *vector_relation* (estensione concreta di *vrel_t*) spiegata in precedenza:

File: vset_cudd.c

```
1   struct vector_relation {
2       vdom_t      dom;
3       size_t      vector_size ;
4       DdManager   *manager;
5       DdNode      *dd;
6       DdNode      **vec_to_bddvar;
7       DdNode      **prime_vec_to_bddvar;
8       DdNode      *variables;
9       DdNode      *prime_variables;
10      DdNode      *all_variables ;
11      DdNode      *projection;
12      DdNode      *invariant_variables
13  };
```

Definizione concreta di *vrel_t* in CuDD

4.2.2.3 vector_domain:

il dominio infine, come detto in [3.1], conserva un campo *shared* contenente i puntatori a funzione tramite i quali si potrà accedere durante l'esecuzione del programma, alle implementazioni concrete del backend, e inoltre a tutte le informazioni "globali" specifiche di CuDD. Nel particolare, come informazioni globali sono previsti i campi:

- *bits_per_integer*: non sempre gli stati vengono dati in input a LTSmin in forma booleana, ma talvolta intera. Ciò implica che la codifica di un intero, non sia responsabilità di una singola variabile, ma di un gruppo di variabili sufficientemente ampio da poter codificare il valore intero [3.1], a questo proposito, *bits_per_integer* fornisce il dato relativo al numero di variabili necessarie per codificare ciascun numero in input,

- *universe*, *prime_universe* e *complete_universe*: il dominio infine, tiene traccia delle congiunzioni di tutte le variabili semplici, tutte le variabili primate e tutte le variabili sia primate che non definite nell'intera sessione CuDD al fine di favorire le operazioni che fanno richiesta di questi dati. Nel gioco in figura: 4.1 *universe*, *prime_universe* e *complete_universe* saranno cosi strutturati:

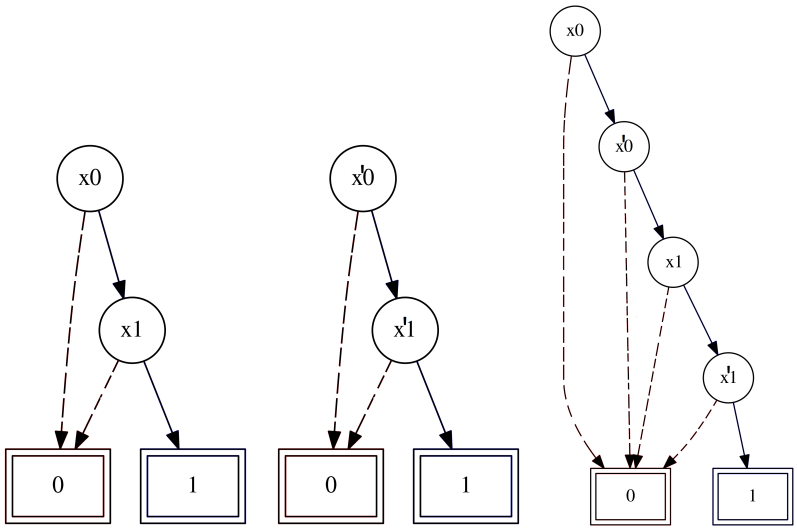


Figura 4.6: *universe*, *prime_universe* e *complete_universe* per un gioco modellato sull'esempio in figura 4.1

Di seguito il codice dell'implementazione di *vector_domain* (estensione concreta di *vdom_t*) illustrata in precedenza:

File: vset_cudd.c

```
1   struct vector_domain {
2       struct vector_domain_shared shared;
3       DdNode **vec_to_bddvar;
4       DdNode **prime_vec_to_bddvar;
5       DdNode *universe;
6       DdNode *prime_universe;
7       DdNode *complete_universe;
8       size_t bits_per_integer ;
9   };
```

Definizione concreta di *vdom_t* in CuDD

4.2.3 Implementazione dei Wrapper

Di seguito verrà fornita, una breve spiegazione di come sono state impostate le funzionalità più interessanti, tramite CuDD, per ottenere i wrapper.

Gestione del GC nei wrappers la politica utilizzata da CuDD per la *garbage collection* [4.1.2] risulta particolarmente aggressiva. Se da un lato infatti è molto efficace, consentendo un grosso risparmio in termini di memoria, dall'altro richiede grandissima attenzione da parte del programmatore, per evitare che nodi effettivamente ancora in uso vengano reclamati improvvisamente dal GC e deallocati, portando successivamente ad un errore di memoria.
Inoltre è anche possibile eventualmente disattivare il GC; effettuare una scelta del genere porta però ovviamente, un degrado delle prestazioni consistente, non appena i BDD su cui si lavora raggiungono dimensioni significative.
Per questo motivo all'interno di ciascun wrapper, si è provveduto ad un attenta referenziazione manuale dei nodi dei BDD al fine di usare il GC con efficacia. Di seguito è riportato un esempio banale di creazione di un BDD tramite il quale verrà illustrato come istruire il GC ad intervenire nel giusto modo. Al suo interno

viene generato un BDD molto semplice che genera quattro variabili e assegna *true* alla concatenazione delle loro negazioni:

```
DdManager *manager;
DdNode *f, *var, *tmp;
int i;

...

f = Cudd_ReadOne(manager);
Cudd_Ref(f);
for (i = 3; i >= 0; i--) {
    var = Cudd_bddIthVar(manager,i);
    tmp = Cudd_bddAnd(manager,Cudd_Not(var),f);
    Cudd_Ref(tmp);
    Cudd_RecursiveDeref(manager,f);
    f = tmp;
}
```

Figura 4.7: Un esempio di corretta referenziazione dei BDD [Somenzi, 2016]

In [4.7] sono presenti innanzitutto alcune funzioni fondamentali, il cui comportamento viene spiegato di seguito:

- *Cudd_ReadOne* è la funzione responsabile dell'inizializzazione dei nodi terminali di un BDD con il valore di *true*, dualmente esiste la funzione *Cudd_ReadLogicZero* la quale inizializza i nodi terminali con *false*,
- *Cudd_bddAnd* esegue l'*and* logico tra le variabili. Da notare che la sua invocazione, manipola i BDD in ingresso generando nuovi nodi e ritornando un nuovo BDD in output,
- *Cudd_bddIthVar* prende in input un intero i e verifica se una variabile etichettata con tale indice sia stata già creata, se si la restituisce, altrimenti provvede a creare una nuova variabile,
- *Cudd_Ref* e *Cudd_RecursiveDeref* provvedono a modificare il conteggio dei riferimenti, sui nodi del BDD passatogli in input.

Dall'esempio riportato si possono evincere le nozioni fondamentali per agire con il GC:

- il GC non viene invocato direttamente dal programmatore; il suo intervento non è determinabile, nell'esempio infatti, non è possibile sapere il punto in cui il GC interverrà,
- diretta conseguenza del punto precedente è che una referenziazione va effettuata subito dopo un'operazione (es. *Cudd_bddAnd*) ogni volta che essa genera nuovi nodi. Nell'esempio precedente infatti, anche se il BDD completo si otterrà alla fine del ciclo, poichè non è possibile predire l'intervento del GC, vanno referenziati e dereferenziati anche i risultati parziali,
- le variabili sono trattate come un eccezione dal GC in quanto non necessitano di referenziazione e dereferenziazione.

Codifica di uno stato in un BDD LTSmin come detto, provvede ad inviare ai suoi backend gli stati codificati in array di interi. Il processo che viene effettuato per tradurli in BDD è il seguente:

- innanzitutto, dato in input un array di n interi che codifica uno stato, esso viene convertito in un array di valori booleani di grandezza $n \times bits_per_integer$;
- in secondo luogo viene creato un BDD con valore terminale *true* corrispondente alla sequenza booleana ottenuta al passo precedente. Per fare ciò CuDD prevede la funzione

```
1  DdNode *Cudd_bddComputeCube(DdManager *dd, DdNode **vars, int
       *encode, int  n)
```

  la quale prende in input un array di variabili, un array di interi con la codifica booleana di ciascuna variabile e un intero contenente la dimensione dei due array (che deve essere concorde) e restituisce un BDD con la codifica booleana translata in BDD e mappata sull'array delle variabili in input;

Funzionalità di base CuDD offre le funzioni per effettuare le principali operazioni booleane sui BDD come:

```
DdNode *Cudd_bddOr(DdManager * dd, DdNode * f, DdNode * g)
DdNode *Cudd_bddAnd(DdManager * dd, DdNode * f, DdNode * g)
DdNode *Cudd_Not(DdNode * f)
```

Proprio come ci si aspetta, *Cudd_bddOr* computa la disgiunzione tra due BDD e ritorna un puntatore al nuovo BDD risultante.
Cudd_bddAnd computa la congiunzione tra due BDD e ritorna un puntatore al nuovo BDD risultante.
Cudd_Not ritorna un puntatore al complemento del BDD in ingresso.
Ricordando che i BDD in questo caso rappresentano, nient'altro che funzioni booleane, che determinano l'appartenenza o meno ad un un insieme di una specifica codifica di uno stato [2.3], è facile notare che effettuare l'operazione di OR su due BDD equivale a creare una nuova funzione che determina l'appartenenza ad un insieme di tutte le codifiche dei due set su cui si opera.

X_0	X_1	X_2	f	g	f OR g
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	1	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	1	1	0	1

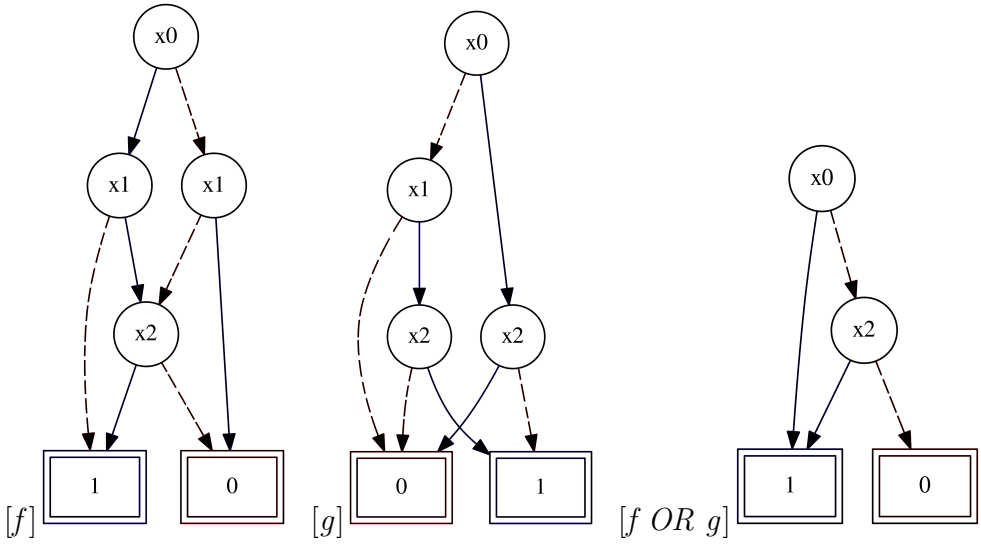


Figura 4.8: OR di due generiche funzioni booleane f e g con relativa codifica in BDD

Grazie a ciò tramite le operazioni di base fornite da CuDD, dati due BDD B_1 e B_2 che codificano due set di stati S_1 e ${}_2S$ è possibile ottenere banalmente le operazioni insiemistiche di unione di intersezione e sottrazione su BDD nel seguente modo:

- $S_1 \cup S_2 = B_1 \vee B_2$
- $S_1 \cap S_2 = B_1 \wedge B_2$
- $S_1 \setminus S_2 = B_1 \wedge \neg B_2$

Predecessori di un insieme di stati un altra operazione fondamentale, di cui LTSmin richiede l'implementazione è quella del calcolo dei predecessori di un set di stati S, dato un determinato insieme di trasizioni R [2.2.1].
Vista la struttura interna delle transizioni [3.1.2] tale operazione consta di tre passaggi fondamentali:

- *rinomina*: poichè in R le informazioni relative agli stati di arrivo delle transizioni sono contenute dalla porzione di BDD codificata dalle variabili primate $vars^{'}$, si necessita di un'operazione di rinomina, che permetta di mappare il set di stati S, definito sulle variabili non primate $vars$, con una codifica $S^{'}$ corrispondente nelle variabili primate,

- *restrizione*: una volta rinominati gli stati è necessario restringere R alle sole transizioni che terminano in $S^{'}$,

- *quantificazione*: infine per ottenere i predecessori, basta quantificare via la parte delle transizioni relativa agli stati di terminazione, contenuta nelle variabili primate.

Combinando i vari passi insieme è possibile ottenere l'operazione completa formalizzata come segue:

$$pred(S, R) \leftarrow \exists vars^{'}(R \wedge rename(S, vars^{'}))$$

CuDD a riguardo fornisce le seguenti utili funzioni:

- *Cudd_bddSwapVariables* prende in ingresso un BDD e due set di variabili e restituisce un BDD identico a quello in input, ma scambiando gli identificativi delle variabili del BDD contenuti nel primo array x con quelli presenti nel secondo array y. Tramite tale primitiva è possibile ottenere la funzionalità di rinomina di uno stato, scambiando sue le variabili "normali" con le non primate. Ad Esempio per il gioco in figura: 4.1 per effettuare la rinomina dei vertici è necessario porre $x = [X_0, X_1]$ e $y = [X_0', X_1']$

```
1  DdNode *Cudd_bddSwapVariables(DdManager *dd, DdNode *f, DdNode
       **x, DdNode **y, int n)
```

- *Cudd_bddAndAbstract* unisce in un solo colpo il passo di restrizione e di quantificazione, infatti computa l'intersezione tra due BDD e dal risultato quantifica via la congiunzione delle variabili presenti nell'ultimo parametro in input. Concretamente, passando a tale funzione i BDD di:

- un insieme di stati S rinominato in S' con $Cudd_bddSwapVariables$,
- della relazione di transizione,
- del campo $prime_variables$ di S che contiene la congiunzione delle variabili primate su cui è definito S'

si ottiene esattamente $pred(S, R)$.

```
DdNode *Cudd_bddAndAbstract(DdManager *manager, DdNode *f,
    DdNode *g, DdNode *cube)
```

Infine un ultimo inciso: come spiegato in precedenza [3.1.2] le relazioni non sono sempre definite su tutte le variabili degli stati. Se ciò accade viene assunta implicitamente l'invarianza di valore tra le variabili e le corrispondenti primate non presenti nell'insieme di transizioni. Tale proprietà però, non viene presa in considerazione nei passaggi precedenti. Per ottenere questo ulteriore vincolo, bisogna "arricchire" l'insieme delle transizioni R con delle clausole sulle variabili che esprimano tale condizione.
L'invarianza può essere espressa nell'algebra booleana con il costrutto $XNOR$

X_0	X_1	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

Figura 4.9: XNOR

CuDD fornisce a riguardo una funzione apposita per effettuare lo XNOR tra BDD.

```
DdNode *Cudd_bddXnor(DdManager *dd, DdNode *f, DdNode *g)
```

Richiamando tale funzione sulle coppie di variabili semplici e primate non presenti nella relazione R delle transizioni per ottenere delle clausole, e restringendo appunto R a rispettarle, è possibile aggiungere la proprietà di invarianza sulle variabili non definite. Più precisamente seguendo la semantica di [3.1.2], dato un gruppo di transizioni R definito tramite $vector_relation$, per tutte le variabili x,$x^{'}$ non presenti negli array vec_to_bddvar e $prime_vec_to_bddvar$ di R, ma definite nel gioco (presenti nelle congiunzioni $universe$ e $prime_universe$ del $vector_domain$), bisogna eseguire $Cudd_bddXnor(manager, x, x^{'})$ e intersecare il risultato ottenuto con il BDD di R.

Successori di un insieme di stati un'operazione da implementare molto simile al calcolo dei predecessori, è quella relativa ai successori.
Essa necessita esattamente delle stesse operazioni intermedie di $pred$, ma eseguite con un diverso ordine:

- *restrizione*: poichè nei set di transizioni le informazioni relative agli stati di partenza, sono contenute nelle variabili non primate; nel caso del calcolo dei successori, avremo in input un set degli stati di "partenza" che non ha bisogno di rinomina, in quando già definito sulle non primate. Il primo passo è dunque, restringere le transizioni per ottenere solo quelle che partono dal set di stati in input,

- *quantificazione*: per ottenere i successori, bisogna quantificare via la parte delle transizioni relativa agli stati di partenza, contenuta nelle variabili non primate,

- *rinomina*: il risultato del passo precedente sarà l'insieme di stati successori del set in input, definito però sulle variabili rinominate. Per ripristinare gli identificativi propri delle variabili degli stati a questo punto, è necessario un ulteriore passo di rinomina per passare dalle variabili primate agli identificativi "normali".

$$succ(S, R) \leftarrow rename(\exists vars(R \wedge S), vars)$$

Capitolo 5

Implementazione dei solver

In questa sezione si introdurranno, i tratti distintivi dell'implementazione di Zielonka utilizzata da LTSmin per risolvere i PG. In seguito verrà posto l'accento sulle varie migliorie strutturali, aggiunte col presente lavoro di tesi. Successivamente verrà illustrata un'implementazione simbolica di Priority Promotion, anch'essa accessoriata di alcuni validi accorgimenti implementativi.

5.1 Algoritmo di Zielonka

L'implementazione fornita in LTSmin rispecchia molto fedelemente quanto illustrato in [2.2.2], di seguito sono riportate nel dettaglio, le parti dell'implementazione che più si distaccano dalla rappresentazione astratta dell'algoritmo.

Deadlocks una prima osservazione da fare è che fino ad ora, si è scelto volutamente di non considerare un caso particolare verificabile nei PG, in quanto poco interessante dal punto di vista algoritmico. Può succedere infatti che all'interno di un PG siano presenti stati isolati[1] o comunque stati che non abbiano archi uscenti, i cosidetti "vicoli ciechi" (*deadlocks*). In queste situazioni di gioco non si può effettuare nessun'altra mossa e convenzionalmente viene assegnata la sconfitta (quindi la vittoria all'avversario), al giocatore che non può effettuare

[1]stati isolati: stati non connessi a nessun altro vertice del grafo.

alcuna mossa.

La versione LTSmin di Zielonka gestisce questa evenienza come segue:

- ad ogni iterazione ricorsiva sono recuperati gli stati che hanno sicuramente un successore nel PG(*no_deadlock_states*). Essi successivamente sono sottratti ai vertici del gioco divisi per giocatore, così da ottenere i *deadlocks* separati per ciascun *player* (Appendice A [3]righe 5-10),

- a questo punto:

  – se non sono presenti *deadlocks* si stabilisce semplicemente qual'è la priorità migliore nel gioco e qual'è il giocatore corrente ad essa associato (Appendice A [3] righe 16-19 e 25),

  – se sono presenti *deadlocks* gli viene data precedenza per cercare di assegnarli prima possibile ad una *winning region* e toglierli dalla computazione. Il solver assegna sostanzialmente al giocatore corrente i "vicoli ciechi dell'avversario" in cui egli è sicuramente vincente (Appendice A [3] righe 20-24 e 27-29).

Attrattore fatto ciò Zielonka procede classicamente con l'operazione di attrattore [2.2.1] che LTSmin struttura nel seguente modo (Allegato A [2]):

- innanzitutto prende classicamente in input tre parametri: PG, insieme di stati su cui fare l'attrattore e un giocatore ed esegue ciclicamente i seguenti passi,

1 trova gli stati predecessori dell'insieme in ingresso controllati dal giocatore che sta effettuando l'operazione di attrazione. Annette tali vertici al risultato finale, insieme al set di stati in input(Allegato A [2] righe 6-10),

2 tutti gli altri passi da ora in poi, sono invece dediti a recuperare i predecessori controllati dal giocatore avversario, che non possono evitare di essere attratti. Con un primo passaggio, in particolare, vengono recuperati tutti i predecessori della regione correntemente attratta manovrati dall'avversario,

che possiamo definire come "nodi candidati ad essere attratti"(Allegato A [2] righe 11-15),

3 col risultato ottenuto al passo precedente, vengono poi calcolati i successori di tali "nodi candidati". Sottraendo successivamente tali nodi alla parte di attrattore calcolato fino ad ora, si ottengono le "vie di fuga" dei nodi candidati (Allegato A [2] righe 16-23),

4 ricavando i predecessori di quelle che abbiamo definito come le "vie di fuga" e rimuovemndoli dall'insieme dei nodi candidati ad essere attratti, si ottengono i vertici candidati appunto all'attrazione e che in oltre non hanno archi che gli permettano di sfuggire (Allegato A [2] righe 24-30),

5 infine vengono semplicemente aggiunti al risultato i nodi computati (Allegato A [2] riga 31),

- L'algoritmo ripete tutti i passi descritti in precedenza, finchè dopo un iterazione completa, non riesce ad attrarre nessun nuovo nodo(Allegato A [2] riga 34).

Calcolo del sottogioco Zielonka utilizza poi l'attrattore ricavato come al punto precedente, per computare il sottogioco su cui effettuare la ricorsione. Per fare ciò viene adoperata la funzione *spg_game_restrict* [5] la quale, dato in input un gioco e un insieme di stati:

- rimuove dal set dei vertici di tutto il gioco, l'insieme di stati in input ([5] riga 1),
- allo stesso modo sottrae stati anche ai set che contengono i vertici controllati da ciascun giocatore ([5] righe 2-4),
- elimina i vertici in input dalla codifica della funzione di priorità ([5] righe 5-6),
- e infine se nel gioco così ridotto, non sono più presenti stati con minima e massima priorità associata, le priorità limite vengono aggiornate con

un nuovo valore minimo e massimo in base agli stati ancora presenti nel sottogioco ([5] righe 7-8 e 11-13).

5 *spg_game_restrict*(PG, *state_set*)

1: $vset_minus(PG.v, state_set)$
2: **for** pl `in` $PLAYERS$ **do**
3: $\quad vset_minus(PG.v_player[pl], state_set)$
4: **end for**
5: **for** pr `in` $PG.min_priority...PG.max_priority$ **do**
6: $\quad vset_minus(PG.v_priority[pr], state_set)$
7: $\quad$ **if** $pr == PG.min_priority\ AND\ vset_is_empty(PG.v_priority[pr])$ **then**
8: $\quad\quad PG.min_priority \leftarrow PG.min_priority + 1$
9: $\quad$ **end if**
10: **end for**
11: **repeat**
12: $\quad PG.max_priority \leftarrow PG.max_priority - 1$
13: **until** $(vset_is_empty(PG.v_priority[PG.max_priority]))AND(PG.max_priority \geq PG.min_priority)$

Una volta calcolato il sottogioco, vengono effettuate le chiamate ricorsive che avranno il compito di calcolare le *winning region* dei sottogiochi per ciascun giocatore ($result.win[player]$). Per un implementazione concreta di [2] relativa ad LTSmin si rimanda all'Appendice A [3].

5.1.1 Ottimizzazioni

L'algoritmo presentato in precedenza è suscettibile a varie ottimizzazioni, sia dal punto di vista algoritmico che implementativo. Di seguito sono presentate le migliorie applicate all'impemenazione LTSmin di Zielonka in questo elaborato di tesi:

Deadlocks dato un PG contenente deadlocks, si può osservare che tali nodi sono per definizione sempre vincenti per l'avversario del giocatore che li controlla [5.1]; in altre parole è possibile determinare subito a quali *winning region* appartengono senza dover sfruttare un solver di PG per doverlo scoprire. Inoltre dato un gioco PG senza *deadlocks*, se da esso si ottiene il sottogioco $PG' = PG \backslash X$ dove X è l'attrattore di un sottoinsieme degli stati di PG, si ha la garanzia che PG'

non contiene a sua volta deadlocks in quanto σ-trappola [2.2.1]. Grazie a queste considerazioni è possibile rimuovere dall'algoritmo, la parte di computazione relativa alle *deadlocks*. Esse infatti possono essere assegnate preventivamente, insieme ai loro attrattori, alle *winning region* corrette. Fatto ciò si può risolvere Zielonka, sul sottogioco $PG' = PG \backslash (A \cup B)$ con $A = Attractor(PG, deadlocks[\sigma], \bar{\sigma})$ e $B = Attractor(PG, deadlocks[\bar{\sigma}], \sigma)$.

Di seguito il codice della funzione preposta a questo scopo:

6 *remove_deadlocks*(*PG*, *result*)

```
no_deadlocks ← vset_create()
for pl in {player_0, player_1} do
    deadlocks[pl] ← vset_create()
    deadlocks[pl] ← PG.v_player[pl]
end for
for group in groups do
    vset_prev(no_deadlocks, PG.v, PG.e[group], PG.v)
    vset_minus(deadlocks[pl], no_deadlocks)
end for
if vset_is_empty(deadlocks[player_0]) then
    result.win[player_1] ← Attractor(PG, deadlocks[player_0], player_1)
end if
if vset_is_empty(deadlocks[player_1]) then
    result.win[player_0] ← Attractor(PG, deadlocks[player_1], player_0)
end if
spg_game_restrict(PG, result.win[player_0])
spg_game_restrict(PG, result.win[player_1])
```

Riduzione delle chiamate ricorsive anche dal punto di vista più strettamente algoritmico è possibile implementare una significativa miglioria rispetto a quanto proposto in [2]. Banalmente è ideale per un algoritmo di questo tipo, effettuare meno chiamate ricorsive possibili, così da risolvere un minor numero di sottoproblemi e velocizzare l'esecuzione.

A questo proposito, ritornando a [2], si vede che quando vengono calcolate le *winning regions* in un sottogioco (figura: 5.1), Zielonka verifca due casi possibili ([2] rigo 9):

1 nel sottogioco il giocatore avversario risulta sempre perdente ($WR'_{\bar{\sigma}} = \emptyset$). In questo caso si può annettere il primo attrattore ($Attr(A, \sigma)$) calcolato

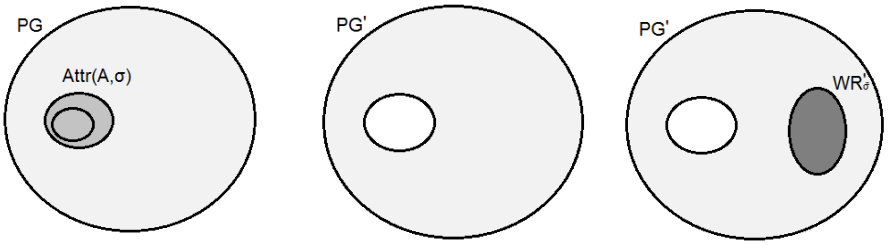


Figura 5.1: Calcolo di una *winning region in un sottogioco* PG'

dall'algortimo alla soluzione WR_σ del giocatore corrente in quanto tale insieme è dominato da una priorità con "parità" concorde al giocatore e la soluzione del sottogioco $WR'_{\bar{\sigma}}$ non può intervenire in nessun modo su A per "attrarre via" nodi all'avversario.

2 nel sottogioco il giocatore avversario risulta non sempre perdente ($WR'_{\bar{\sigma}} \neq \emptyset$) in questo caso avviene una seconda chiamata ricorsiva, sostanzialmente alla ricerca di un sottogioco vinto da un solo giocatore. In realtà però le motivazioni della condizione 1 che portano a saltare la chiamata ricorsiva, sono facilmente verificabili anche in un altro sotto caso della condizione 2. Infatti se ($WR'_{\bar{\sigma}} \neq \emptyset$) può succedere che (figura: 5.2):

- $WR'_{\bar{\sigma}}$ attragga degli elementi dal gioco originale PG ($WR'_{\bar{\sigma}} \subset Attr(PG, WR'_{\bar{\sigma}}, \bar{\sigma})$). In questo caso, non si è in grado di escludere che dei nodi di $Attr(A, \sigma)$ su PG possano essere attratti e la seconda chiamata ricorsiva diventa inevitabile,
- $WR'_{\bar{\sigma}}$ non attragga degli elementi dal gioco originale PG ($WR'_{\bar{\sigma}} = Attr(PG, WR'_{\bar{\sigma}}, \bar{\sigma})$). In questo caso, quindi a maggior ragione non attrarrà nessun nodo da A che pure fa parte del gioco originale PG. Per questo motivo è possibile evitare la chiamata ricorsiva comportandosi esattamente come nella condizione 1.

Di seguito viene riportato lo pseudocodice dell'algortimo modificato rispetto a [2], seguendo i principi sopraelencati:

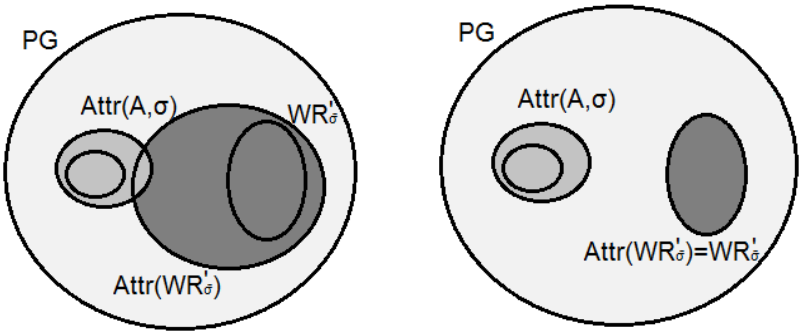


Figura 5.2: Possibili attrattori in un PG di una *winning region* calcolata da Zielonka in un sottogioco

7 Zielonka($PG = \langle A = \langle V_0, V_1, R\rangle, P, W_c\rangle$)

1: **if** $V = \emptyset$ **then**
2: **return** $(\emptyset, \emptyset)$
3: **else**
4: $max_pr \leftarrow max(P(V))$
5: $\sigma \leftarrow player_of_parity(max_pr)$
6: $A \leftarrow P^{-1}(max_pr)$
7: $A \leftarrow Attr(PG, A, \sigma)$
8: $(WR'_0, WR'_1) \leftarrow Zielonka(PG \backslash A)$
9: $Y \leftarrow Attr(PG, WR'_{\bar{\sigma}}, \bar{\sigma})$
10: **if** $WR'_{\bar{\sigma}} = Y$ **then**
11: $(WR_\sigma, WR_{\bar{\sigma}}) \leftarrow (WR'_\sigma \cup A, \emptyset)$
12: **else**
13: $(WR'_0, WR'_1) \leftarrow Zielonka(PG \backslash Y)$
14: $(WR_\sigma, WR_{\bar{\sigma}}) \leftarrow (WR'_\sigma, WR'_{\bar{\sigma}} \cup Y)$
15: **end if**
16: **end if**
17: **return** (WR_0, WR_1)

Operare su un sottogioco minimale un altro punto dove si è potuto intervenire, è nella fase di calcolo del sottogioco definita formalmente in [2.2.1] e implementata tramite *spg_game_restrict* [5]. Un PG definito sul tipo *parity_game* [3.1.4] di LTSmin infatti è un elemento piuttosto complesso e che quindi occupa una relativamente grande quantità di memoria. Tenere sullo stack durante i passaggi ricorsivi tanti PG di grandi dimensioni, però, non è una condizione auspicabile. Si è valutato inoltre che le intersezioni effettuate da *spg_game_restrict* su tutti i set di *parity_game*, per ottenere un sottogioco partendo da un PG, non necessariamente debbano essere effettuate tutte insieme; molte infatti potrebbero

essere non sempre necessarie.
Sulla base di queste considerazioni si è deciso di sviluppare un ulteriore implementazione di Zielonka che ricalcoli ricorsivamente, non un intero sottogioco PG', ma soltanto un set di stati, contenente tutti i vertici del sottogioco che chiamaremo *vsubgame* (l'equivalente di $PG'.v$).
Tale soluzione ovviamente necessita di alcune leggere ristrutturazioni a livello implementativo:

- per il calcolo della migliore priorità (Appendice A [3] righe 16-19), è necessario intersecare ogni set contenente gli stati ad una specifica priorità, con *vsubgame*,
- l'attrattore necessita di modificare la sua firma da $Attactor(PG, Y, \sigma)$ a $Attactor(PG, vsubgame, Y, \sigma)$. Ogni volta infatti che un operazione dell'attrattore coinvolge i set $PG.v$ e $PG.v_player$ va modificata per utilizzare $PG.v \cap vsubgame$ e $PG.v_player \cap vsubgame$.

5.2 Priority Promotion

Per quanto riguarda PP la nuova implementazione integrata in CuDD rispecchia anch'essa molto da vicino quanto illustrato in [3]. Di seguito verrà mostrato nel dettaglio, il funzionamento di tutte le componenti dell'algoritmo soprattutto per quanto riguarda la parte relativa al *Searcher* [4]

Verifica delle σ-Uscite Parte fondamentale dell'algoritmo di Priority Promotion è la necessità di poter verificare se un quasi-dominio [2.2.1] è aperto o chiuso rispetto ad un determinato gioco PG. Per fare ciò è stato necessario fornire un calcolo delle σ-uscite [2.2.1] per LTSmin. Di seguito è illustrata l'mplementazione finale:

- l'algoritmo prende in ingresso tre parametri:

  1 un Parity Game PG,

2 una set di stati Y rappresentante la regione di cui si deve verificare le vie di fuga,

3 il giocatore *player* che "vuole evadere" da Y.

- l'algoritmo, verifica in prima istanza se gli stati controllati dal giocatore in input (che ha interesse a scappare) hanno un successore esterno al set in ingresso e quindi una "via di fuga" (Appendice A [4] righe 12-18); nel momento in cui nè viene trovata anche solo una, si è dimostrato che esiste un uscita da Y e l'algoritmo termina,

- nel caso in cui il giocatore che ha interesse ad "evadere" non avesse vie di fuga, è ancora possibile che vi siano delle uscite se l'avversario è costretto ad uscire. Ciò può accadere se egli controlla stati che hanno archi uscenti, ma nessuno di questi gli permette di restare in Y.
  Per verificare ciò:

  – vengono in primo luogo raccolti gli stati di Y controllati dal giocatore che ha interesse a non "scappare", i quali abbiamo almeno un successore in Y stesso e che quindi possono evitare di uscire (Appendice A [4] righe 19-23),

  – infine se il set degli stati di Y controllati dal giocatore che non vuole fuggire è esattamente uguale al set degli stati di Y controllati dal giocatore che non vuole fuggire e ha un arco che glielo permetta, allora Y si può definitivamente considerare chiusa rispetto a PG (Appendice A [4] righe 24-29).

Best Exit Priority nel caso in cui una regione sia aperta in un gioco PG, ma chiusa nel sottogioco PG' correntemente esaminato dall'algortimo, va effettuata una promozione della regione corrente, con una già precedentemente calcolata nel dominio degli spazi. A tale proposito, risulta fondamentale, il calcolo della *best exit priority* definita come: la priorità più bassa, tra quelle associate ai nodi che consentono la "fuga" dalla regione [2.2.3]. Di seguito viene riportata l'implemntazione di una funzione preposta a ciò:

- l'algoritmo prende in ingresso tre parametri:

  1 un Parity Game PG di cui, effettivamente, serviranno solo le infomazioni relative alle transizioni per valutare i collegamenti tra regioni,

  2 un array *regions* di ciascuna regione computata nello spazio dei domini. Da tenere presente è che priorità associata a ciascuna regione, la mappa nello spazio dei domini,

  3 la priorita *region_priority* che funge da id nell'array *regions* della regione di cui si vuole ottenere la bep (*best exit priority*).

- inizialmente la funzione, raccoglie in un unico set, tutte le regioni con priorità associata migliore di quella corrente. Le priorità associate a tali regioni infatti, sono tutte bep candidate ([8] righe 3-5),

- successivamente vengono calcolati tutti i nodi d'uscita dalla regione corrente ([8] righe 6-12),

- viene trovato il nodo d'uscita che appartiene alla regione con priorità associata più bassa e l'algoritmo termina restituendo la priorità ([8] righe 13-18).

8 $best_exit_priority(PG, regions[], region_priority))$

```
1:  all_regions ← vset_create()
2:  escape_states ← vset_create()
3:  for pr in PG.min_priority...(region_priority − 1) do
4:      vset_union(all_regions, regions[pr])
5:  end for
6:  partial ← vset_create()
7:  for group in groups do
8:      vset_next(partial, regions[region_priority], PG.e[group])
9:      vset_intersect(partial, all_regions)
10:     vset_union(escape_states, partial)
11:     vset_clear(partial)
12: end for
13: for pr in (region_priority − 1)...PG.min_priority do
14:     vset_intersect(regions[pr], escape_states)
15:     if vset_is_empty(regions[pr]) == false then
16:         return pr
17:     end if
18: end for
```

Aggiornamento della Funzione di Priorità Ogni volta che una promozione viene effettuata, avviene un reset di tutte le regioni presenti nello spazio dei domini, con priorità associata peggiore della *best exit priority*. Tutti gli stati che quindi erano mappati con tali priorità nelle regioni, devono essere riassociati correttamente nel gioco, alla loro priorità originale[2.2.3]. Per rispondere a tale necessità è stata implementata la funzione $update_priority_function$:

- essa prende in ingresso tre parametri:
  - il gioco originale PG dato in input a priority promotion,
  - il sottogioco PG' esaminato in una specifia iterazione del *Searcher*,
  - l'array *regions* delle sole regioni che devono essere eliminate dallo spazio dei domini poichè è avvenuta una promozione in PP.
- in primo luogo vengono posti in un unico insieme, tutti gli stati che dovranno essere rimappati nella relazione di transizione del sottogioco PG'([9] righe 3-5),

- in secondo luogo viene verificato ciascuno stato a quale prioprità fosse originariamente associato, intersecando l'insieme degli stati che devono essere rimappati, con la relazione di transizione originaria $PG.v_priority$. Una volta fatto ciò, gli stati vengono riassegnati al loro mapping originario nel sottogioco $PG^{'}$([9] righe 6-9),

- infine se aggiungendo i nuovi stati in $PG^{'}$, alcuni di essi sono mappati su priorità al di fuori del range attuale del sottogioco ($PG^{'}.min_priority$ e $PG^{'}.max_priority$), tale range viene aggiornato con i nuovi valori limite.

9 $update_priority_function(PG.P, PG^{'}.P, regions[])$

```
 1: all_regions ← vset_create()
 2: states_to_remap ← vset_create()
 3: for region in regions do
 4:     vset_union(all_regions, region)
 5: end for
 6: for pr in PG.max_priority...PG.min_priority do
 7:     states_to_remap ← PG.v_priority[pr]
 8:     vset_intersect(states_to_remap, all_regions)
 9:     vset_union(PG'.v_priority[pr], states_to_remap)
10:     if vset_is_empty(PG'.v_priority[pr]) == false then
11:         if PG'.min_priority > pr then
12:             PG'.min_priority ← pr
13:         end if
14:         if PG'.max_priority < pr then
15:             PG'.max_priority ← pr
16:         end if
17:     end if
18: end for
```

5.2.1 Ottimizzazioni

Di seguito è riportata una variante dell'algoritmo presentato in [3] e implementato seguendo i dettami della sezione precedente:

Calcolo dei sottogiochi Come visibile nel *Searcher* ([4] 13-16), dato un sottogioco $PG^{'}$ preso in esame ad una data iterazione, e posto che in tale iterazione si innesca una promozione da una priorita α ad una priorità *bep* migliore,

bisogna calcolare un nuovo insieme $PG^{'}.V$ su cui operare alla successiva iterazione.

Si può notare che ad ogni ciclo, l'insieme $PG^{'}.V$ contiene esattamente l'unione degli stati non annessi a nessuna regione e gli stati presenti nella regione correntemente presa in esame.

Conseguenza di ciò è che ogni volta che si esegue una promozione l'insieme $PG^{'}.V$ è identico al $PG^{'}.V$ calcolato la prima volta che la regione con priorità di destinazione *bep* è stata creata e quindi sostanzialmente si sta ricalcolando due volte lo stesso insieme.

Per sfuttare questa situazione si è deciso di implementare una variante di PP che preservi uno stack degli stati dei sottogiochi così da ottenere $PG^{'}.V$ dopo una promozione, in tempo costante.

La struttura del Searcher dell'algoritmo così organizzato è la seguente:

10 Searcher($PG = \langle A = \langle V_0, V_1, R\rangle, P, W_c\rangle$,$PG^{'}$, max_pr)

1: $Reg[max_pr] \leftarrow PG.V[max_pr]$
2: $Reg[max_pr] \leftarrow Attr(PG^{'}, Reg[max_pr], \sigma)$
3: $subgame[max_pr] \leftarrow PG^{'}.V$
4: **while** $true$ **do**
5: $\quad \sigma \leftarrow player_of_parity(max_pr)$
6: $\quad$ **if** $Esc_{\bar{\sigma}}(PG, Reg[max_pr]) = \emptyset$ **then**
7: $\quad\quad$ **return** $(Attr(PG, Reg[max_pr], \sigma), \emptyset)$
8: $\quad$ **else**
9: $\quad\quad$ **if** $Esc_{\bar{\sigma}}(PG^{'}, Reg[max_pr]) = \emptyset$ **then**
10: $\quad\quad\quad bep \leftarrow best_exit_priority(Reg, Reg[max_pr]))$
11: $\quad\quad\quad Reg[bep] \leftarrow merge(Reg[bep], Reg[max_pr])$
12: $\quad\quad\quad reset(Reg, bep)$
13: $\quad\quad\quad update_priority_function(PG.P, PG^{'}.P, Reg)$
14: $\quad\quad\quad PG^{'}.V \leftarrow subgame[bep]$
15: $\quad\quad\quad max_pr \leftarrow bep$
16: $\quad\quad$ **else**
17: $\quad\quad\quad PG^{'} \leftarrow PG^{'}.V \backslash Reg[max_pr]$
18: $\quad\quad\quad max_pr \leftarrow PG^{'}.max_pr$
19: $\quad\quad\quad subgame[max_pr] \leftarrow PG^{'}.V$
20: $\quad\quad$ **end if**
21: $\quad$ **end if**
22: **end while**

5.3 L'Attrattore

Osservando l'attrattore implementato in LTSmin $attractor(PG, attractor, player)$ ([Allegato A 2]) e utilizzato per entrambi i solver, è evidente che valutare quali nodi controllati da *player* sono attratti (step 1 Allegato A [2] righe 6-10), è molto meno oneroso rispetto alla valutazione dei nodi controllati da $1 - player$ (step 2 Allegato A [2] righe 12-33). In quest'ultimo caso infatti, bisogna verificare che tutti gli archi di questi vertici siano indirizzati verso l'attrattore.

Idealmente, per velocizzare l'esecuzione, è auspicabile ridurre il numero di volte in cui viene eseguito questo secondo passaggio. Per questo motivo si è deciso di produrre un ulteriore attrattore che modifichi la sequenza d'acquisizione dei nodi, con lo scopo di eseguire meno volte il secondo step e recuperare ad ogni iterazione un numero maggiore dei nodi controllati da $1-player$ destinati ad essere attratti. Se infatti (Allegato A [2]) attrae in modo alternato nodi di *player* e di $1-player$ fino a raggiungere una convergenza sui nodi presenti nell'attrattore, questa ulteriore metodologia punta a ottenere prima un recupero massimale di tutti i nodi di *player* e solo in un secondo momento a recuperare quelli controllati da $1-player$. L'implementazione di tale metodo è la seguente:

11 $attractor(PG, attractor, player)$

```
state_set ← vset_create()
step1_attractor ← vset_create()
step2_attractor ← vset_create()
repeat
    state_set ← attractor
    repeat
        step1_attractor ← attractor
        attractor ← step1(PG, attractor, player)
    until vset_is_equal(step1_attractor ← attractor)
    repeat
        step2_attractor ← attractor
        attractor ← step2(PG, attractor, player)
    until vset_is_equal(step2_attractor ← attractor)
until vset_is_equal(attractor, state_set)
return attractor
```

12 $step1(PG, attractor, player)$

```
partial_pr ← vset_create()
pred ← vset_create()
for group in groups do
    vset_prev(partial_pr, attractor, PG.e[group], PG.v_player[player])
    vset_union(pred, partial_pr)
end for
vset_union(attractor, pred)
return attractor
```

13 $step2(PG, attractor, player)$

```
partial_pr ← vset_create()
pred ← vset_create()
for group in groups do
    vset_prev(partial_pr, attractor, PG.e[group], PG.v_player[1 − player])
    vset_union(pred, partial_pr)
end for
succ ← vset_create()
partial_succ ← vset_create()
for group in groups do
    vset_next(partial_succ, pred, PG.e[group])
    vset_intersect(partial_succ, PG.v)
    vset_union(succ, partial_succ)
end for
vset_minus(succ, attractor)
partial_pr ← vset_clear()
not_taken_pred ← vset_create()
for group in groups do
    vset_prev(partial_pr, succ, PG.e[group], PG.v_player[1 − player])
    vset_union(not_taken_pred, partial_pr)
end for
vset_minus(pred, not_taken_pred)
vset_union(attractor, pred)
return attractor
```

Capitolo 6

Generatori Simbolici di Parity Games

Operazioni identiche, eseguite dai *solver* sia su PG rappresentati simbolicamente, sia enumerativamente, possono pesare in modo totalmente diverso sui tempi di risoluzione. Per questo motivo, una corretta valutazione empirica, risulta assolutamente fondamentale, per verificare le performance in merito alla bontà degli algoritmi implementati e dell'impatto che hanno su di essi tutte le varianti descritte in precedenza.

Una problematica, a riguardo, è sicuramente la necessità di eseguire i risolutori su giochi che siano sufficientemente complessi, così da mettere effettivamente alla prova gli algoritmi ed esplorare le loro prestazioni in casi sfavorevoli. Inoltre è importante che i PG utilizzati come *benckmark* per i test, siano derivati o quanto meno "assomiglino", a problemi reali. Quasi mai, infatti, PG ottenuti da *transition systems* modellati su sistemi veri, presentano una struttura interna completamente randomica; molto più spesso contengono schemi ricorrenti e comportamenti peculiari difficilmente riproducibili in modo casuale, ma che non di meno è importante affrontare.

Allo stesso tempo, non sono presenti in letteratura, *dataset* di PG simbolici che coprano esaustivamente tutte le casistiche sopra illustrate. Per questo motivo si è provveduto all'implementazione di varie metodologie di generazione di PG simbolici che abbiano le caratteristiche desiderate.

6.1 Convertitore di giochi espliciti in simbolici

Se da una parte non esistono generatori simbolici di PG, questo non è vero per quanto riguarda i PG espliciti [Keiren, 2015], dove nel corso degli anni, vari tool sono stati proposti dalla comunità scientifica. Per questo motivo si è deciso, in prima istanza, di sfruttare quanto già presente in letteratura, per ottenere giochi simbolici con una conversione diretta da generatori espliciti.

6.1.1 Il tool PGSolver

PGsolver [Oliver Friedmann, 2014] è un tool sviluppato presso la facoltà d'informatica di Francoforte, il quale raccoglie al suo interno tutti i principali risolutori per PG, proposti in letteratuta fino ad oggi. Inoltre è corredato di alcuni generatori di Parity Game espliciti, utili per la formazione di Benchmark di testing. Tali generatori permettono di ottenere PG di varie tipologie, al fine di testare le performance dei risolutori in diverse situazioni. In particolare è possibile generare le seguenti categorie di gioco prese in esame:

- *randomgame*: il grafo del gioco ha la struttura di un'unica componente connessa generata in modo completamente casuale,
- *clusteredrandomgame*: il grafo del gioco è organizzato in diversi *cluster* di nodi che poi sono interconnessi tra loro,
- *steadygame*: manipola le priorità e l'appartenenza ai giocatori di ciascun nodo del grafo, in modo da generare giochi che cercano di eludere, le principali ottimizzazioni e semplificazioni fatte dai *solver*,
- *cliquegame*: genera un grafo in cui ciascun nodo non ha cicli su se stesso, ma presenta una struttura di tipo *clique* e quindi con molti cicli, configurazione questa, che talvolta mette in difficoltà alcuni *solver*.

I generatori di PGSolver una volta invocati restituiscono PG espliciti con la seguente notazione:

- un *header* contenente il massimo id assegnabile a un vertice,

- una sequenza di righe con la presente struttura:

  1 un valore intero id_0 corrispondente all'id di un vertice del grafo,

  2 priorità assegnata al vertice id_0,

  3 un valore compreso tra zero e uno che sta ad indicare l'appartenenza, ad un dei due giocatori, del vertice id_0 nel PG,

  4 una sequenza $id_1, id_2 ... id_n$ che rappresenta i vertici tali per cui nel PG esistono le transizioni $(id_0, id_1), (id_0, id_2), ...(id_0, id_n)$.

Si è provveduto quindi ad elaborare un *parser* che scorresse il gioco così strutturato per tradurlo nella sua forma simbolica.
Di seguito è illustrata una spiegazione di come funziona tale convertitore con tanto d'implementazione:

- innanzitutto l'algoritmo prende in input due argomenti:
  - uno *stream* da cui recuperare il gioco esplicito,
  - un file in cui verrà salvato il gioco simbolico.
- all'avvio, vengono inzializzati una serie di parametri:
  - Innanzitutto viene recuperato dall'*header* dello *stream*, l'informazione sul massimo id presente nel gioco, il quale ci permette di avere anche un idea esatta del numero di stati di cui sarà formato il PG ([14] righe 1-2). In base a ciò viene impostato il valore *vars_number* che sarà corrispondente al numbero di variabili di cui saranno formati i BDD che codificheranno gli stati simbolicamente,
  - grazie alle informazioni dell'header viene allocato anche un array di una struttura *state* per salvare provvisoriamente gli stati in modo esplicito ([14] righe 3), avente la seguente struttura:

```
1  typedef struct state {
2      int     id; /*id dello stato corrente*/
3      int     player; /*giocatore che controlla lo stato corrente*/
```

```
4        int     priority ; /*priorita dello stato corrente*/
5        int    *transitions ; /*id degli stati d'arrivo delle transizioni
         che hanno come punto di partenza lo stato corrente*/
6    }state;
```

  – vengono inoltre inizializzati altri parametri relativi al gioco che verrà creato ([14] righe 4-6). Da notare che poichè la codifica esplicita di PGSolver non prevede la divisione delle transizioni in gruppi, di default viene impostato un unico gruppo.

- l'algoritmo procede con l'esaminare una riga alla volta, lo *stream* in input [15], recuperando tutte le informazioni relative a ciascuno stato e alle transizioni che lo coinvolgono come "vertice di partenza",

- una volta conclusa la fase di *parsing* si prosegue con la codifica simbolica ([14] righe 10-28). Inizialmente infatti, viene creato un PG simbolico vuoto ([14] righe 8-9),

- successivamente viene passato in rassegna l'array degli stati popolato dal *parser*. Ciascun id di uno stato, viene convertito in una codifica binaria tramite la funzione *encode_state* (con la quale verrà mappato in seguito su un BDD) e aggiunto agli insiemi simbolici del PG vuoto creato in precedenza([14] righe 17-22),

- vengono poi recuperati, per ciascuno stato, gli id dei vertici d'arrivo per le sue transizioni. Ciascuno di essi è poi convertito in binario per eseguire la conversione in transizioni simboliche ([14] righe 25-28),

- infine il gioco simbolico così codificato, viene salvato sul file dato in input.

14 $converter(Stream, file)$

```
states_number ← get_header(Stream) + 1
vars_number ← ⌈log_2(states_number)⌉
states[states_number]
min_priority ← ∞
max_priority ← −1
groups ← 1
states ← parser(Stream, states_number)
domain ← create_domain(vars_number, backend_id)
PG ← spg_game_create(domain, vars_number, groups, min_priority, max_priority)
for state in states do
    if state.priority < min_priority then
        min_priority ← state.priority
    end if
    if state.priority > max_priority then
        max_priority ← state.priority
    end if
    encoded_state ← encode_state(state.id, vars_number)
    vset_add(PG.v, encoded_state)
    if state.player == player_0 then
        vset_add(PG.v_player[player_0], encoded_state)
    else
        vset_add(PG.v_player[player_0], encoded_state)
    end if
    vset_add(PG.v_priority[state.priority], encoded_state)
    for transition in states.transitions do
        encoded_trans ← encode_state(transition, vars_number)
        vrel_add(PG.e[0], encoded_state, encoded_trans)
    end for
end for
spg_save(PG, file)
```

15 $parser(Stream, states_number)$

```
states[states_number]
parsed_states ← 0
while Stream ≠ ∅ do
    row ← get_row(Stream)
    states[parsed_states] ← get_state_id(row)
    states[parsed_states].id ← get_state_id(row)
    states[parsed_states].player ← get_state_player(row)
    states[parsed_states].priority ← get_state_priority(row)
    states[parsed_states].transitions ← get_state_transitions(row)
    parsed_states ← parsed_states + 1
end while
return states
```

6.2 Generatore di giochi simbolici random

Ottenere giochi simbolici partendo da una rappresentazione esplicita presenta una criticità particolarmente penalizzante: dover codificare simbolicamente gli stati e le transizioni una a una costringe alla creazione di giochi di piccole dimensioni. Dover scorrere infatti, tutti gli stati e le transizioni iterativamente porta a tempi di creazione enormi per giochi nell'ordine di milioni di stati. Inoltre aggiungendo, all'interno dei BDD del gioco, una clausola per ogni stato, si arriva molto spesso alla generazione di Decision Diagram con una struttura molto complessa ed intricata. Dati n stati codificati con un BDD B, nel caso peggiore possono essere necessari b nodi per ottenere B, con b linerare rispetto ad n. Utilizzare una codifica simbolica, in questo modo perde completamente di significato, in quando viene a mancare la dote di sinteticità dei dati che essa porta.

Per risolvere tali problemi è auspicabile quindi rappresentare molteplici stati con singole clausole senza scorrerli uno ad uno, al fine di mantenere il numero delle iterazioni del generatori ragionevoli e la crescita della dimensione dei BDD, logaritmica rispetto al numero di stati rappresentati. Inoltre può essere interessante misurare l'impatto che può avere la scomposizione della relazione di transizione in gruppi, ma nel caso di generatori espliciti di giochi, tale configurazione non è contemplata.

Per affrontare tali necessità si è deciso di implementare dei generatori di PG che non si basassero (almeno non completamente) su giochi espliciti.

6.2.1 Codifica di clausole

LTSmin fornisce un wrapper solo per l'aggiunta ad un set di un singolo stato [3.2.1], non è quindi prevista la possibiltà di inserire sottoinsiemi in set già esistenti, utilizzando una singola clausola. Per questo motivo è stato integrato nel sistema il wrapper:

```
1    void vset_add_subset(vset_t set,const int* e)
```

rispetto a *vset_add* che prende ingresso un array di interi positivi da codificare; add subset accetta, nelle caselle dell'array anche il valore -1. Date le variabili $x_n, ..., x_m$ usate per codificare l'i-esima casella dell'array tale che $e[i] = -1$, per tali variabili verrà considerata "valida" qualsiasi codifica binaria (figura 6.1).

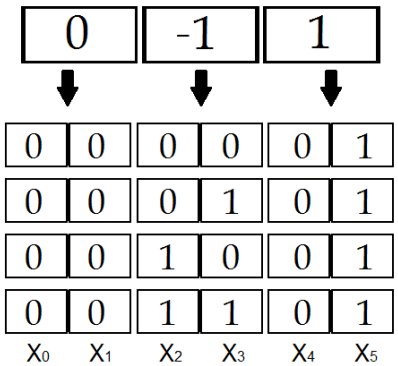


Figura 6.1: Esempio di semantica di una clausola

In questo modo il numero di stati codificati da un singolo array, aumenta esponenzialmente per ogni gruppo di variabili il cui valore "non è vincolato". Inoltre grazie alle regole di riduzione proprie dei BDD [2.3] i Decision Diagram assumono una forma estremamente sintetica; nel caso esposto in figura 6.1 ad esempio le variabili X_2 e X_3 vengono sottintese nel BDD che codifica la sequenza

"$0, -1, 1$" poichè il sottoalbero di ciascuna delle loro combinazioni (formato dai nodi delle variabili X_4 e X_5) è identico (figura 6.2).

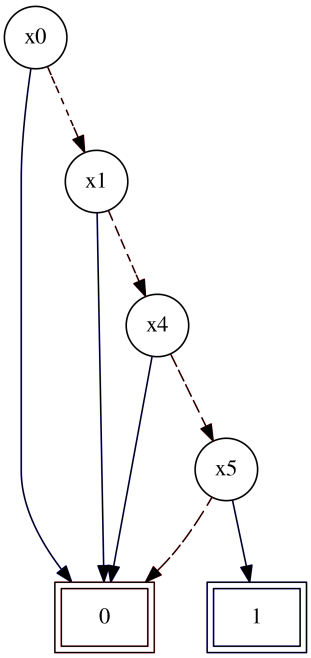


Figura 6.2: BDD che codifica l'esempio in figura 6.1

Trattandosi *vset_add_subset* di un wrapper, per poter essere adoperato, ne va generata un'implementazione per i backend che s'intende utilizzare. Per quanto concerne l'implementazione CuDD, come per l'aggiunta di un singolo stato [4.2.3] va utilizzata la funzione:

```
1 DdNode *Cudd_bddComputeCube(DdManager *dd, DdNode **vars, int
      *encode, int  n)
```

Eliminando infatti dal array *vars* delle variabili su cui un BDD è definito, quelle da "non vincolare", è possibile ottenere un Decision Diagram come mostra l'esempio precedente.

6.2.2 Generazione e suddivisione di insiemi di stati

Il primo passo per ottenere un PG simbolico è generare degli stati e suddividerli nei vari insiemi in modo randomico, ma coerente con quella che è la struttura di un gioco.

Stati dell'intero gioco: una prima osservazione da fare è che dato un insieme di variabili $vars$ su cui sono definiti i BDD B_i degli insiemi di stati di un parity game PG, si può notare come il numero di codifiche booleane generabili con $vars$ (e quindi di stati in un B_i) è equivalente al massimo a $2^{|vars|}$. Inoltre se in un B_i nessuna delle sue variabili $vars$ è vincolata, esso codifica come vere tutte le possibili combinazioni di valori per ogni $x \in vars$, in modo estremamente sintetico. Un BDD di questo tipo infatti ha la forma mostrata in figura 6.3.

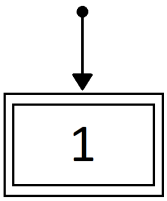


Figura 6.3: BDD che per qualsiasi valore delle variabili su cui è definito, restituisce valore *true*

Per creare gli stati del Parity Game $PG.v$ si è scelto di utilizzare proprio questa codifica, essa infatti perde leggermente di generalità in quanto il numero di stati dei PG creati saranno sempre una potenza di due, ma ciò non costituisce un vincolo particolarmente grave per generare benchmark validi, in compenso però, fornisce una codifica estremamente sintetica per $PG.v$ ([16] riga 2). In termini pratici per fare ciò basterà utilizzare *vset_add_subset* senza vincolare alcuna variabile (per l'implementazione concreta si rimanda all' Appendice A [1]).

Stati di ciascun giocatore: anche nel caso dei set $PG.v_player[player_0]$ e $PG.v_player[player_1]$ si è privilegiata la semplicità del BDD piuttostochè la completa randomicità del gioco. Affinchè un gioco sia effettivamente ragionevole è semplicemente importante che ciascun giocatore controlli un numero congruo di stati. Per assegnare i vertici a ciascuno dei due giocatori si è ancora una volta utilizzato le variabili $vars$. Scegliendo infatti casualmente una variabile $x_i = rand(vars)$ ([16] righe 3) e usandone il valore per discriminare l'appartenenza ad un giocatore piuttosto che all'altro ([16] righe 3-7) si ottengono intrinsecamente una serie di importanti vantaggi:

- innanzitutto tale procedimento è molto rapido. Con la generazione di sole due clausole si ottengono immaediatamente due insiemi la cui unione è l'insieme di tutti gli stati del gioco,
- si ha automaticamente la garanzia che i due insiemi differendo per il valore di una variabile siano disgiunti,
- ancora una volta sono ottenuti BDD estremamente sintetici visto che vincolano il valore di una sola variabile.

La perdita di generalità sta nel fatto che entrambi i giocatori controllano così lo stesso numero di stati, ma poichè la distribuzione in termini di priorità, come si vedrà tra poco, non è invece omogenea come invece è stato per gli insiemi illustrati fin ora, ciò non rappresenta un'assunzione troppo grave.
Di seguito è illustrato lo pseudocodice della generazione di PG simbolici relativamente alla creazione degli insiemi degli stati del gioco e dei vertici controllati dai giocatori:

16 $states_generation(PG, vars_number)$

1: $i \leftarrow Rand(0...n)$
2: $PG.V \leftarrow true$
3: **if** $var == player_variable$ **then**
4: $\quad PG.v_player[player_0] \leftarrow \bar{x}_i$
5: **else**
6: $\quad PG.v_player[player_0] \leftarrow x_i$
7: **end if**

La funzione di priorità: fissato un set Pr di n priorità, far corrispondere sottoinsiemi di stati a ciscuna di esse è un' operazione molto delicata. Bisogna infatti evitare una distribuzione troppo omogenea dei vertici per non dare ai giochi prodotti una fisionomia poco generale. Allo stesso tempo non vincolare opportunamente la distribuzione degli stati, aumenta il rischio che la funzione di priorità mappi una percentuale rilevante degli elementi di Pr su un insieme di vertici vuoto creando un gioco che in realtà ha un numero consistente di priorità in meno, rispetto a quanto fissato.
Per effettuare una suddivisione degli stati che non fosse troppo ricorrente, ma

al tempo stesso non lasciasse priorità mappate su un insieme vuoto di stati si è deciso di agire nel seguente modo:

- innanzitutto ogni volta, prima di assegnare stati ad una determinata priorità, si determina qual'è il valore medio di stati atteso per quella determinata priorità in base al rapporto $states_per_priority = \frac{|remaining_states|}{|uncompleted_priorities|}$ ([17] righe 6-7),

- fatto ciò si stabilisce una tolleranza orientativa entro cui, il numero effettivo di nodi assegnati a una priorità si può discostare dal valore medio; ad esempio: $threshold = \frac{|states_per_priority|}{3}$ ([17] riga 8),

- in base ai valori ottenuti ai passi precedenti si stabilisce quante variabili non vincolare per ottenere un sottinsieme con una clausola che rimanga nei margini stabiliti $free_vars = \lfloor log_2(states_per_priority + threshold) \rfloor$([17] riga 11),

- detto ciò si genera in modo random una clausola che abbia al massimo $free_var$ variabili non vincolate (funzione $clause(variabili, numero_di_variabili_libere)$ [17] riga 11-12),

- la clausola così generata non ha alcuna garanzia di codificare stati che siano disgiunti dagli altri set popolati in precedenza in questo modo. Per questo motivo, affinchè la priorità assegnata ad uno stato sia univoca, si provvedere a sottrarre dalla clausola generata, gli stati già assegnati in altri insiemi ([17] riga 12),

- il procedimento così descritto è eseguito per ciascuna priorità, finchè essa non è stata assegnata ad almeno $states_per_priority - threshold$ stati,

- infine, alla priorità più bassa sono assegnati tutti gli stati rimanenti ([17] riga 21).

Di seguito l'implementazione della funzione di priorità così descritta:

17 $priority_generation(PG, vars)$

1: $current_pr \leftarrow PG.min_priority$
2: $states_with_no_priority \leftarrow PG.v$
3: $subset \leftarrow \emptyset$
4: $uncompleted_priorites \leftarrow PG.max_priority - PG.min_priority$
5: **while** $current_pr \neq PG.max_priority$ **do**
6: $\quad remaining_states \leftarrow |states_with_no_priority|$
7: $\quad states_per_priority = \dfrac{|remaining_states|}{|uncompleted_priorities|}$
8: $\quad threshold \leftarrow \dfrac{states_per_priority}{3}$
9: $\quad added_states \leftarrow 0$
10: $\quad$ **while** $added_states < states_per_priority - threshold$ **do**
11: $\quad\quad free_vars_number \leftarrow \lfloor log_2(states_per_priority + threshold) \rfloor$
12: $\quad\quad subset \leftarrow states_with_no_priority \cap clause(vars, free_vars_number)$
13: $\quad\quad added_states \leftarrow added_states + |subset|$
14: $\quad\quad PG.v_priority[current_pr] \leftarrow PG.v_priority[current_pr] \cup subset$
15: $\quad\quad states_with_no_priority \backslash states_with_no_priority \backslash subset$
16: $\quad\quad subset \leftarrow \emptyset$
17: $\quad$ **end while**
18: $\quad uncompleted_priorites \leftarrow uncompleted_priorities - 1$
19: $\quad current_pr \leftarrow current_pr - 1$
20: **end while**
21: $PG.v_priority[PG.max_priority] \leftarrow states_with_no_priority$

6.2.3 Generazione e suddivisione delle transizioni:

Per quanto riguarda la relazione di transizione, risulta particolarmente interessante studiare l'impatto che può avere la sua suddivisione in gruppi, rispetto l'utilizzo in blocco unico.

Allo stesso tempo però, nello generare randomicamente gruppi di transizioni, bisogna tenere in conto alcuni importanti aspetti:

- innanzitutto, per quanto la generazione debba essere il più casuale possibile, bisogna però tenere comunque sotto controllo il grado degli archi in uscita e in entrata per i nodi. Una quantità di transizioni troppo elevata infatti, può rendere oltremodo semplice la fase di attrazione della risoluzione di un PG, rendendo il gioco banale da risolvere indipendentemente dalla sua dimensione. L'ideale da questo punto di vista, sarebbe specificare in input un grado minimo e massimo da rispettare per ciascun nodo,

- se generati in modo completamente random, possono verificarsi situazioni non desiderate tra un gruppo e l'altro. Ad esempio può accadere che vi sia una relazione di inclusione $Gr_1 \subseteq Gr_2$ oppure un elevato sbilanciamento del numero di transizioni codificate tra i gruppi.

Per rispondere a tutte le esigenze di randomicità e consistenza si è proceduto nella seguente maniera:

- innanzitutto si è cercato di distribuire in modo omogenero il numero di transizioni tra un gruppo e l'altro. Tale procedimento si è attuato dividendo le variabili di codifica dei BDD, in modo randomico tra i vari insiemi([18] righe 10-24). La generazione dei gruppi effettuata in questo modo è stata così vincolata:

  1 affinchè ciascun gruppo "fosse a conoscenza" del giocatore di cui codifica le transizioni, si è deciso di definire ciascun gruppo con la variabile che discrimina il giocatore ([18] riga 7),

  2 per ciascun gruppo creato ci si è accertati che per ogni coppia Gr_1, Gr_2 definita sulle variabili $vars_1, vars_2$ non si verificasse mai la condizione $vars_1 \subseteq vars_2$. Non è plausibile infatti che due gruppi siano chiamati a regolare lo stesso insieme di transizioni,

  3 per non rendere i gruppi totalmente indipendenti tra loro, si è inoltre imposto che i set di variabili $vars_1, vars_2, ...vars_n$ su cui sono definiti i gruppi fossero in parte sovrapposti prevedendo per ciascuna coppia $(vars_i, vars_j)$ un numero massimo di variabili comuni ([18] righe 2-3).

Tenere il controllo del grado dei vertici invece è risultato essere un'operazione particolarmente ostica. Si ricorda infatti che non definire un gruppo di transizioni su delle variabili, equivale a dare per esse, una semantica ben precisa di invarianza tra stato di partenza e stato d'arrivo [3.1.2]. A causa di ciò definire due transizioni a e b su insiemi di variabili diverse, non garantisce affatto che a e b non siano esattamente la stessa transizione. Un'ulteriore problematica inoltre è che per tenere sotto controllo il grado in uscita ed in entrata dei vertici senza però perdere

di casualità, prima di aggiungere una nuova transizione, bisognerebbe effettuare una verifica del grado dei vertici scelti. Ciò comporterebbe però, in primo luogo un controllo su tutti i gruppi, in secondo luogo di dover ricercare nodi finchè non se ne ottengono due con un grado valido. Una verifica di questo tipo risulterebbe oltremodo onerosa in termini di tempi per il generatore di giochi.
Poichè i PG prodotti esplicitamente con PGSolver [6.1.1] permettono di tenere sotto controllo il grado dei vertici, si è deciso di sfruttare il tool, per generare le transizoni come segue:

- data una serie di gruppi $Gr_1, Gr_2, ...Gr_n$, ciascuno di essi definito su un insieme di variabili $vars_1, vars_2, ...vars_n$, per ogni gruppo Gr_i si è deciso di generare un gioco esplicito di PGsolver con esattamente $2^{\frac{|vars_i|}{2}}$ stati ([19] righe 3-4).

  In questo modo è possibile associare l'id di un vertice del gioco esplicito così generato, a ciascuna codifica di delle variabili di $vars_i$ non primate ([19] riga 8). A questo punto, sfruttando le transizioni generate dal gioco esplicito e il mapping tra id e variabili non primate è possibile riprodurre simbolicamente le transizioni all'interno del gruppo ([19] righe 8-13).

Poichè in base alla relazione di transizione, si definisce gran parte della topologia e della difficoltà del gioco, si è deciso di supportare la generazione di transizioni basata sui quattro tipi diversi di giochi espliciti supportati da PGSolver: *randomgame*, *clusteredrandomgame*, *steadygame*, *cliquegame* [6.1.1].

Di seguito lo pseudocodice dell'implementazione delle transizioni diviso per semplicità in due parti:

- una prima parte di definizione, relativa alla distribuzione delle variabili dei BDD nei gruppi;

- una seconda parte di generazione per mezzo di giochi espliciti generati con PGSolver.

18 $trans_definition(vars, player_variable, vars_number, groups_number)$

```
1: groups ← ∅
2: max_vars_per_group ← ⌈(vars_number × 2) / groups_number⌉
3: min_vars_per_group ← ⌈vars_number / groups_number⌉
4: for i in 0...groups_number − 1 do
5:     group ← new()
6:     group.length ← rand(min_vars_per_group...max_vars_per_group)
7:     group.vars ← group.vars ∪ player_variable
8:     groups ← groups ∪ group
9: end for
10: vars_copy ← vars
11: while groups ≠ ∅ do
12:     for group in groups do
13:         if vars == ∅ then
14:             vars ← vars_copy
15:         end if
16:         if group.length > |group.vars| then
17:             groups.vars ← group.vars ∪ pick_one(vars, group.vars)[1]
18:         end if
19:         if group.length == |group.vars| then
20:             result ← result ∪ group)
21:             groups ← groups\group)
22:         end if
23:     end for
24: end while
25: return result
```

[1] *pick_one* prende in ingresso due set di variabili e resituisce una variabile dal primo set (rimuovendola) che non sia presente nel secondo.

19 $trans_generator(PG, groups, explicit_game, explicit_game_params)$

```
1: for group in groups do
2:     create_transition_relation(PG.R, group)
3:     exp_game_size ← 2^(groups.length)
4:     Stream ← exec(PGSolver, explicit_game, exp_game_size, explicit_game_params)
5:     while Stream ≠ ∅ do
6:         row ← get_row(Stream)
7:         state ← get_state_id(row)
8:         first_state ← encode_state(state, groups.vars)
9:         transitions ← get_state_transitions(row)
10:        for transition in transitions do
11:            second_state ← encode_state(transition, groups.vars)
12:            PG.R ← PG.R ∪ (first_state, second_state)
13:        end for
14:    end while
15: end for
```

6.2.4 Modalità alternativa di generazione

La tecnica per generare la funzione di priorità descritta in precedenza è ottima dal punto di vista della randomicità. Nella praticà però presenta una difficoltà non di poco conto: man mano che l'algoritmo procede, il numero di stati senza priorità, banalmente, si riduce. Ciò comporta una difficoltà sempre maggiore nel trovare in modo casuale, clausole che codifichino stati non assegnati. Durante l'esecuzione, le clausole generate avranno una probabilità sempre maggiore, di apportare una quantità minima di stati o di essere un sottoinsieme di clausole già generate in precedenza. Poichè quindi raggiungere la soglia minima di stati per ciascuna priorità, diventa sempre più complicato ad ogni iterazione, tale modalità risulta molto lenta in fase di generazione dei giochi.

A causa delle motivazioni sopra riportate si è deciso di implementare una generazione alternativa della funzione di priorità, allo scopo di accelerare la creazione dei PG, anche a costo di perdere un pò di "casualità" nella produzione delle clausole.

Per implementare la modalità "veloce" ancora una volta si agisce direttamente sulle variabili di codifica dei BDD in particolare:

- sia $n = |Pr|$ sono necessarie almeno $m = \lceil log_2(n) \rceil$ variabili per assegna-

re almeno una codifica booleana (clausola) a ciascun insieme di priorità (esattamente m se n è una potenza di due)([20] righe 1-2).

- una volta stabilito m, le variabili sono recuperate dai gruppi nel seguente modo:
  - viene presa una viariabile da ogni gruppo (ovviamente e3vitando duplicati),
  - se $m > group_number$ si riscorrono i gruppi prendendo un'ulteriore variabile,
  - il passo precedente viene ripetuto finchè le variabili prescelte non sono esattamente m.

  Poichè le variabili sono assegnate ai gruppi in modo casuale, recuperarle da essi in questo modo, non costituisce un vincolo troppo rigido per la funzione di priorità ([20] righe 5-14),

- vengono ricavate le 2^m clausole ottenibili delle m variabili e ciascuna di esse è poi è assegnata iterativamente a un set di priorità diverso ([20] riga 19),

- infine, nel caso in cui n non sia potenza di due e quindi restino delle combinazioni delle m priorità non assegnate; poichè non può accadere che degli stati non abbiano una priorità assegnata, si procede ad aggiungere le clausole rimanenti agli insiemi di priorità già generati rieseguendo il passo precedente ([20] righe 21-23).

Di seguito l'implementazione di questa ulteriore tecnica per la creazione della funzione di priorità:

20 $fast_priority_generation(PG, groups, vars)$

```
priorities_number ← PG.max_priority − PG.min_priority
m ← ⌈log_2(priorities_number)⌉
taken_vars ← 0
k ← ∅
while taken_vars < m do
    for group in groups do
        current ← group.vars
        if taken_vars ≥ m then
            Break
        end if
        k ← k ∪ pick_one(current, k)
        taken_vars ← taken_vars + 1
    end for
end while
clause_added ← 0
priority ← PG.min_priority
clause_number ← 2^m
while clause_added < clause_number do
    PG.v_priority[priority] ← PG.v_priority[priority] ∪ ((x_{k_0} = c_0) ∧ (x_{k_1} =
    c_1) ∧ ... ∧ (x_{k_{m−1}} = c_{m−1})) /*nota a piè pagina[2]*/
    priority ← priority + 1
    if priority > PG.max_priority then
        priority ← PG.min_priority
    end if
    clause_added ← clause_added + 1
end while
```

[2] $(x_{k_0} = c_0, x_{k_{m-1}} = c_{m-1})$ Con tale notazione si vuole intendere una clausola in cui a ciascuna variabile in k, selezionata per codificare i set di priorità, è assegnato il valore c_i corrispondente all'i-esimo bit meno significativo del contatore *clause_added*. Tutte le altre variabili nvece sono lasciate non vincolate.

Capitolo 7

Sperimentazioni

Una valutazione corretta della bontà degli algoritmi e delle loro varianti proposte, richiede molta attenzione. Fondamentale è infatti, produrre PG che da una parte abbiano caratteristiche variabili, al fine di notare vantaggi o svantaggi rispetto a particolari tipi di giochi, dall'altra siano complessi e mettano seriamente alla prova gli algoritmi di risoluzione.
Un altro aspetto molto importante è la *feature selection* dei dati ottenuti dai risolutori. Analizzare infatti i parametri più significativi è fondamentale per non classificare in modo erroneo i *solver*. Nella prima parte di questo capitolo ci si concentrerà principalmente sugli aspetti sopra citati, nella seconda invece, saranno esamiate nel dettaglio le prestazioni e le motivazioni per cui, determinati algoritmi risultano più efficaci.

7.1 Generazione di test e raccolta dati

Tutti i test effettuati di seguito sono stati creati utilizzando, l'algoritmo di generazione speigato in [6.2.4]. Tale generatore infatti, offre da una parte, una discreta velocità di creazione, dall'altra una buona casualità dei giochi prodotti. Questa tecnica generativa inoltre è stata preferita a quella illustrata in [6.1] in quanto permette di ottenere PG con un numero di stati e di transizioni, molto più elevato.

I tipi di giochi e loro configurazioni: Per effettuare i test di questo elaborato sono stati utilizzati fondamentalmente, due versioni di gioco basate sulle seguenti tipologie di PGSolver:

- *randomgames:* giochi generati senza una precisa struttura interna, di cui è possibile definire in ingresso:
  - la quantità di vertici,
  - il grado in entrata e in uscita dei vertici,
  - il numero di priorità,
  - numero di gruppi in cui è scomposta la relazione di transizione.
- *clusteredrandomgames:* giochi generati in modo pseudo-random che preservano una certa struttura interna, in cui i vertici sono organizzati per *cluster* interconnessi tra loro.

Le caratteristiche dei PG di test sono state impostate come segue:

- la cardinalità dell'insieme dei vertici del gioco $PG.V$ è stata fissata a $2^{|vars|}$ con $|vars| = 20$ e $vars$ insieme delle variabili non primate su cui è definito il gioco. Si è scelto tale valore in quanto, circa un milione di vertici permettono di avere un gioco di una taglia considerevole, che tuttavia è risolvibile in tempi non troppo elevati;
- il grado in ingresso e in uscita dei vertici è stato fissato a due. Ciò perchè PG con vertici di grado uno rappresentano una casistica troppo specifica, ma allo stesso tempo giochi con vertici di grado elevato, sono in linea generale, troppo facilmente risolvibili;
- il numero di priorità dei PG è stato variato tra duecento, quattrocento e mille priorità per gioco;
- il numero di gruppi dei PG è stato variato tra quattro, sei e otto per gioco.

Per quanto riguarda invece il numero di test effettuati, sono stati generati e risolti complessivamente circa mille e ottocento giochi equamente distribuiti sia tra *clustered* e *random*, sia tra le variazioni di priorità e gruppi spiegate in precedenza.

I giochi così generati sono stati testati sui cinque algoritmi riportati di seguito (di seguito è fornito anche un ID per ciascun algoritmo, per riferimenti futuri):

`ZL0`: l'algoritmo di Zielonka base, disponibile in LTSmin;

`ZL1`: l'algoritmo del punto precedente, con la variante che cerca di ridurre il numero di seconde chiamate ricorsive [5.1.1];

`ZL2`: l'algoritmo del punto precedente che in fase di ricorsione, non passa sottogiochi tra i parametri, ma semplicemente il set di stati di cui è formato il gioco [5.1.1];

`PP0`: la versione base fornita per Priority Promotion;

`PP1`: l'algoritmo al punto precedente con la gestione dei sottogiochi in uno stack [5.2.1].

Di seguito una tabella riassuntiva che illustra tutte le possibili combinazioni testabili con ciascun algoritmo:

Tabella 7.1: Possibili configurazioni di test.

Tipologia di giochi	**# Priorità**	**# Gruppi**
randomgame	200	4
randomgame	200	6
randomgame	200	8
randomgame	400	4
randomgame	400	6
randomgame	400	8
randomgame	1000	4
randomgame	1000	6
randomgame	1000	8
clusteredrandomgame	200	4
clusteredrandomgame	200	6
clusteredrandomgame	200	8
clusteredrandomgame	400	4
clusteredrandomgame	400	6
clusteredrandomgame	400	8
clusteredrandomgame	1000	4
clusteredrandomgame	1000	6
clusteredrandomgame	1000	8

I dati: Allo scopo di ottenere una piena comprensione delle motivazioni per cui, un algoritmo ottenga risultati migliori rispetto ad un altro, si è cercato di estrarre dai *solver* quante più informazioni utili possibili. In particolare per ciascun algoritmo sono stati ricavati:

- il tempo effettivo di risoluzione,
- il tempo complessivo speso dagli attrattori,
- la taglia media dei BDD dei sottogiochi risolti dagli algoritmi,
- il numero di attrattori eseguiti,
- il numero di volte che è stata effettuata un operazione di calcolo dei predecessori di un insieme di stati, nell'attrattore.

Inoltre per quanto riguarda i *solver* basati sull'algoritmo di Zielonka sono stati aggiunti i dati specifici:

- numero di volte che l'algoritmo esegue la sua prima chiamata ricorsiva,
- numero di volte che l'algoritmo esegue la sua seconda chiamata ricorsiva.

Infine per le versioni di *Priority Promotion* gli ulteriori dati specifici estratti sono:

- numero di query (attrattori) eseguite nel complesso dall'algoritmo, per trovare i domini,
- numero di promozioni eseguite dall'algoritmo,
- numero di domini diversi trovati dall'algortimo,
- numero massimo di promozioni necessarie ad ottenere un dominio,
- numero massimo di query necessarie ad ottenere un dominio.

7.2 Analisi delle prestazioni

Una iniziale premessa da fare riguarda le specifiche del dispositivo utilizzato per effettuare i test di performance: tutti gli esperimenti sono stati eseguiti su una macchina con processore AMD A8-6410 APU 2.00 GHz e 8 GB di memoria totale, con limite fissato a 6 GB.
Al fine di ottenere una prima idea di massima sulle prestazioni dei vari algoritmi sono stati eseguiti e riportati i risultati ottenuti da ciascun algoritmo su tutto il dataset.

7.2.1 Le prestazioni di Zielonka

Per quanto riguarda le tre varianti dell'algoritmo di Zielonka, sono stati riportate di seguito:

- numero di primi e secondi passi ricorsivi effettuati dall'algoritmo,
- numero di attrattori computati,
- numero di operazioni di calcolo del predecessore effettuate,
- taglia media dei sottogiochi risolti durante l'esecuzione,
- numero di sottogiochi computati,
- il tempo di risoluzione di ciascun pariti game,
- media e varianza dei tempi di risoluzione.

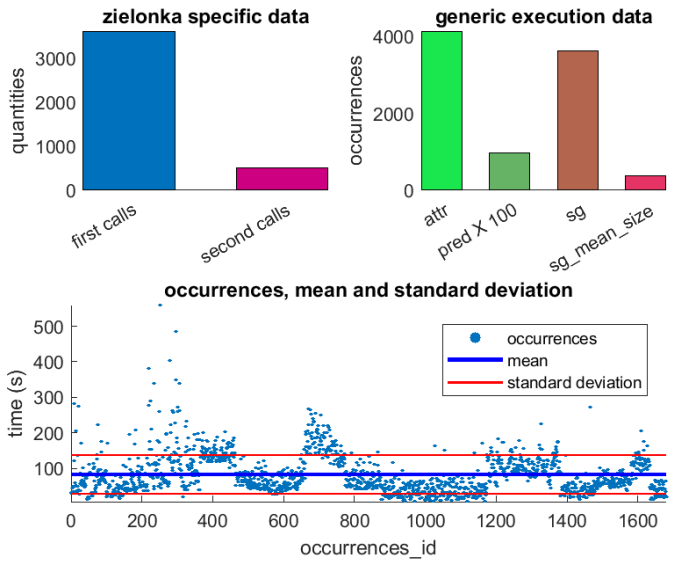


Figura 7.1: Dati relativi all'esecuzione di ZL0 su tutti i giochi.

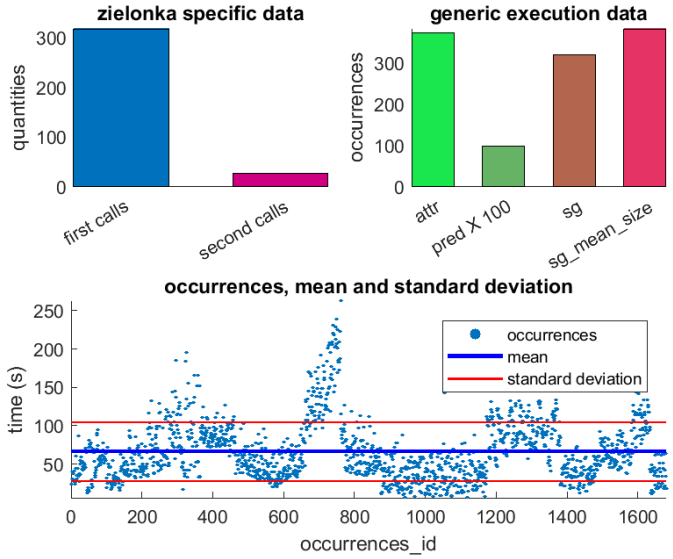


Figura 7.2: Dati relativi all'esecuzione di ZL1 su tutti i giochi.

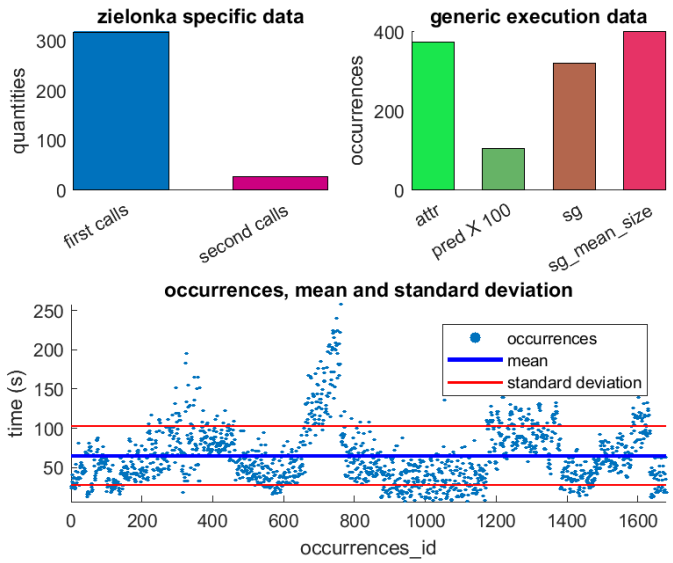


Figura 7.3: Dati relativi all'esecuzione di ZL2 su tutti i giochi.

Si può subito notare, come la variante relativa alla riduzione delle chiamate ricorsive sia, in linea generale, molto efficace. Il numero di chiamate effettuate dal primo algoritmo è di un ordine di grandezza superiore rispetto alle altre due versioni. Diretta conseguenza di ciò è che anche il numero di attrattori calcolati è molto maggiore.

Si può apprezzare inoltre, che i tre *plot* dei tempi di ciascun gioco, hanno una forma molto simile, ciò presuppone che i problemi più "difficili" per l'algoritmo base, restino tali anche per le varianti. Infine è da notare che anche la taglia media dei sottogiochi risolti dal primo algoritmo, risulta essere molto più elevata in contronto alle altre versioni.

Quanto detto fin ora, si traduce in un netto svantaggio in termini di performance rispetto alle due varianti migliorate. L'evidenza di ciò è visibile nella differenza del tempo medio di risoluzione e della varianza, riportata tra i dati in figura 7.1 e quelli in figura 7.2. Non è invece evidente se il passaggio nelle chiamate ricorsive, di un insieme di stati piuttosto che del gioco completo porti effettivamente vantaggi apprezzabili. Per approfondire tale aspetto, di seguito, sono stati messi a confronto i tempi di risoluzione per ciascun PG:

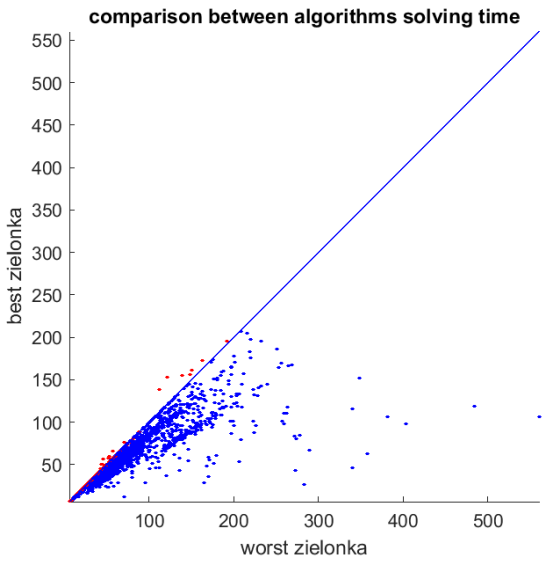


Figura 7.4: A confronto i tempi di risoluzione (in secondi) per ciascun PG di `ZL0` (*worst zielonka*) con `ZL2` (*best zielonka*).

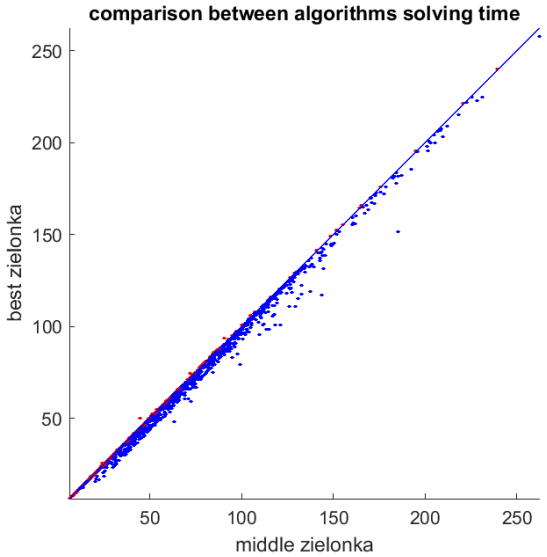


Figura 7.5: Confronto (in secondi) tra i tempi di risoluzione di `ZL1` (*middle zielonka*) e `ZL2` (*best zielonka*) su ciascun PG.

Come evidente gia da prima, nel primo caso riportato in figura 7.4 la differenza e schiacciante. Risulta invece, particolarmente interessante, notare in figura 7.5 che benchè, non fosse evidente nei dati precedenti, l'utilizzo di un insieme di stati, al posto di un intero sottogioco nelle chiamate ricorsive, porta effettivamente dei vantaggi, non elevati, ma molto consistenti, in quanto pochissimi PG sono stati risolti prima dall'altro modello.

Dopo aver osservato come si comportano le varainti di Zielonka in generale, di seguito, sono illustrati gli stessi dati mostrati in precedenza, divisi però, per tipologia di gioco (*randomgame* e *clusteredrandomgame*):

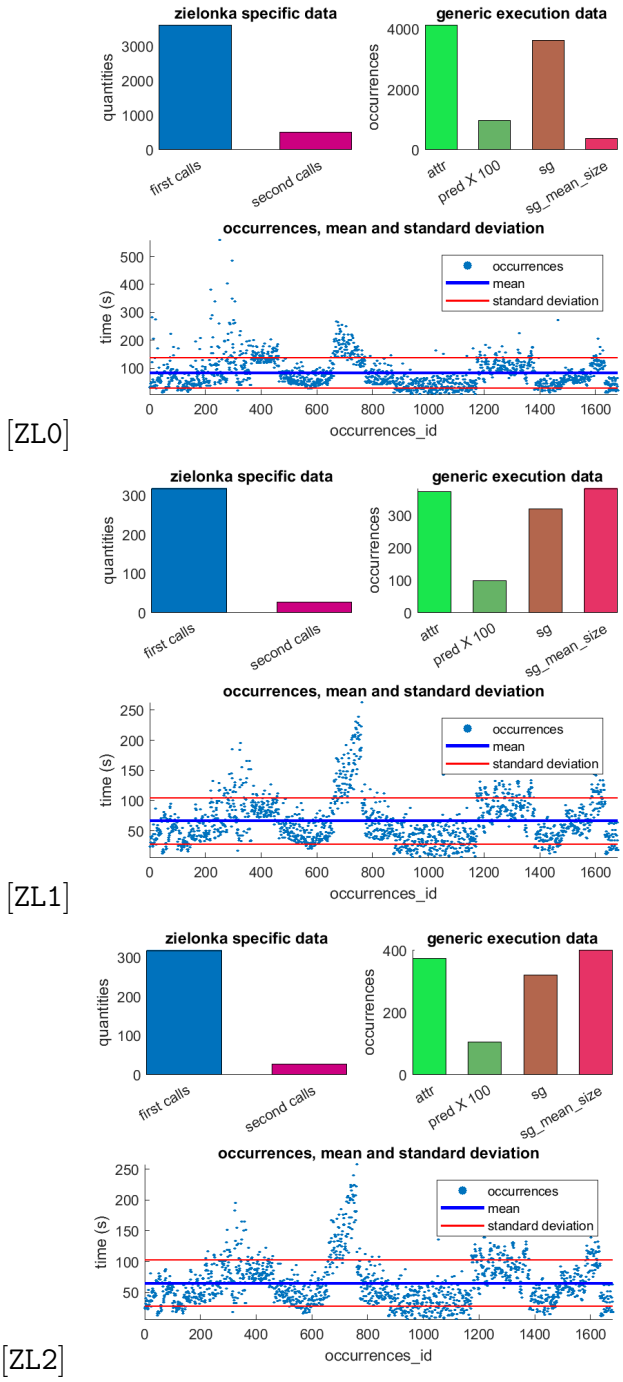


Figura 7.6: Dati relativi alle performance delle varianti di Zielonka su giochi di tipo *randomgame*.

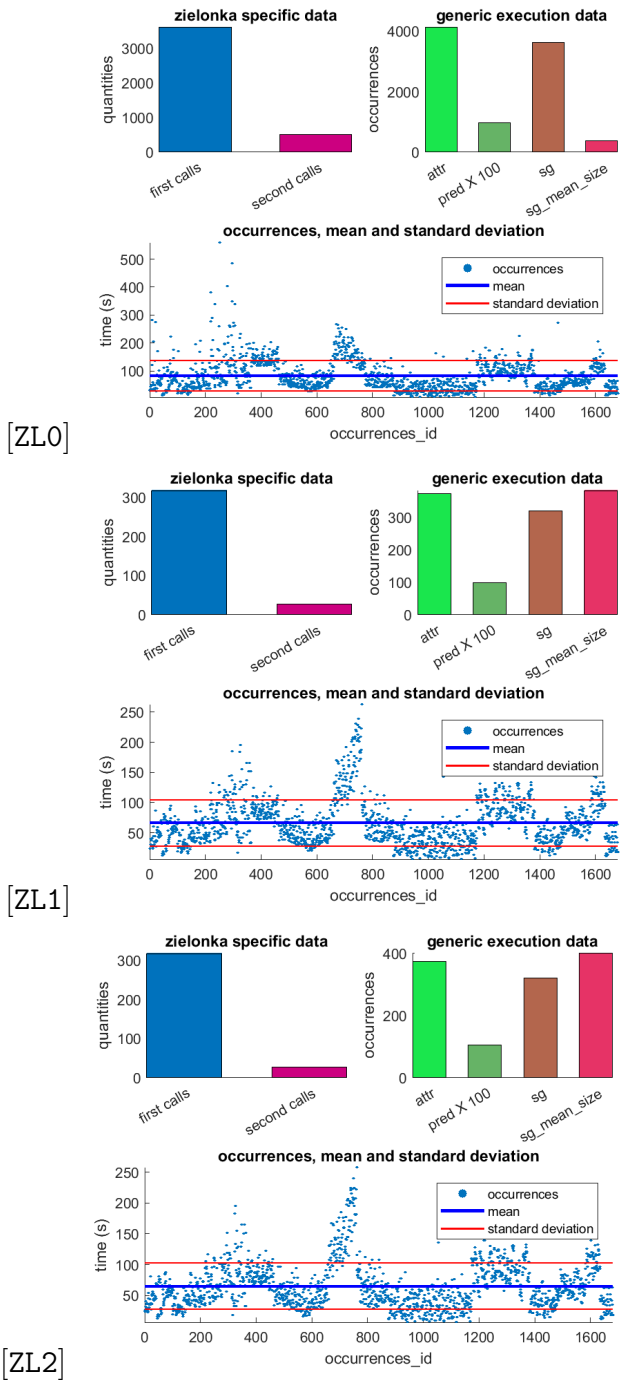


Figura 7.7: Dati relativi alle performance delle varianti di Zielonka su giochi di tipo *clusteredrandomgame*

come si può notare, non sembrano emergere differenze di rilievo da tali grafici generali. Per questo motivo si è deciso di approfondire ulteriormente, andando a verificare come si comportano i *solver* su giochi con specifiche caratteristiche. Di seguito sono ripotati tempo medio di risoluzione e varianza per ciascun'im-

plementazione di Zielonka. I valori così rappresentati discriminano i tempi di risoluzione in primo luogo per tipologia di gioco, in secondo luogo per numero di gruppi e numero di priorità:

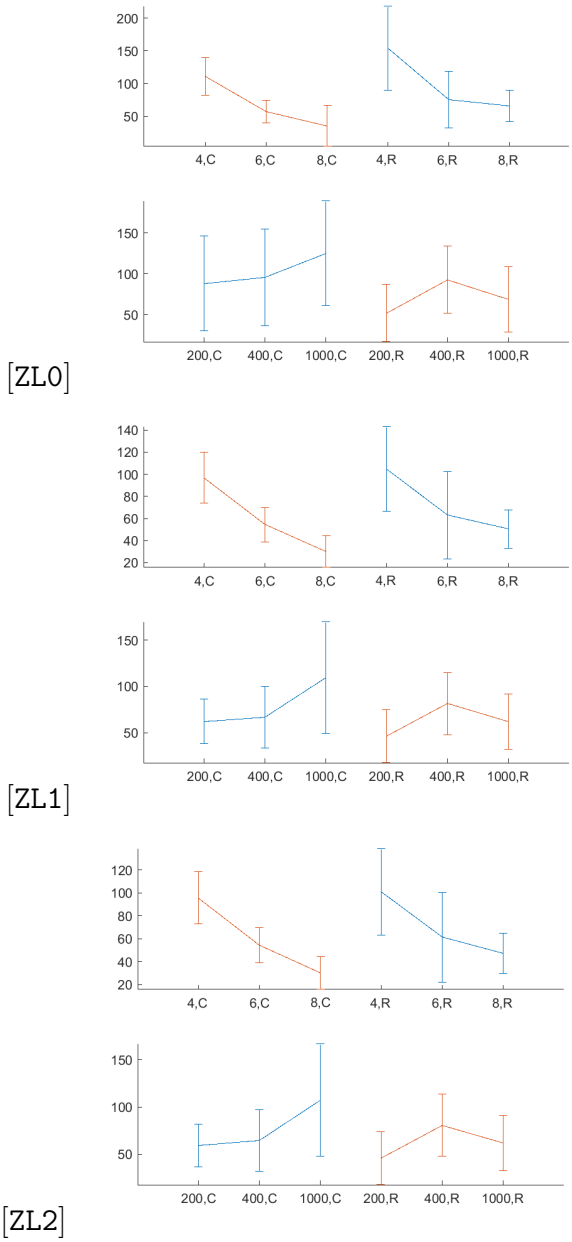


Figura 7.8: Dati relativi alle performance delle varianti di Zielonka filtrate per gioco (C=*clusteredrandomgame*, R=*randomgame*) in secondo luogo per gruppo (4,6,8) o per priorità (200,400,1000)

Dai valori di figura 7.8 emergono alcuni interessanti aspetti. in primo luogo, la complessità dei giochi sembra essere in modo consistente, inversamente proporzionale al numero di gruppi, e direttamente proprozionale al numero di priorità.

Ciò risulta vero in particolare per i giochi di tipo *clusteredrandomgame*, per i quali la prestazione di Zielonka degrada sensibilmente su giochi da quattro gruppi o mille priorità.

7.2.2 Le prestazioni di Priority Promotion

Per quanto riguarda Priority Promotion, al fine di effettuare una valutazione esaustiva, sono state riportate di seguito le seguenti informazioni supplementarii:

- numero di query totali effettuate dall'algortimo,
- numero di promozioni totali effettuate dall'algoritmo,
- numero di domini trovati, utilizzati per comporre le *winning region* dei giocatori,
- numero massimo di query effettuate per ottenere un dominio,
- numero massimo di promozioni effettuate per ottenere un dominio.

Di seguito, com'è stato fatto per le varianti di Zielonka, sono riportate le informazioni generali delle due versioni di Priority Promotion:

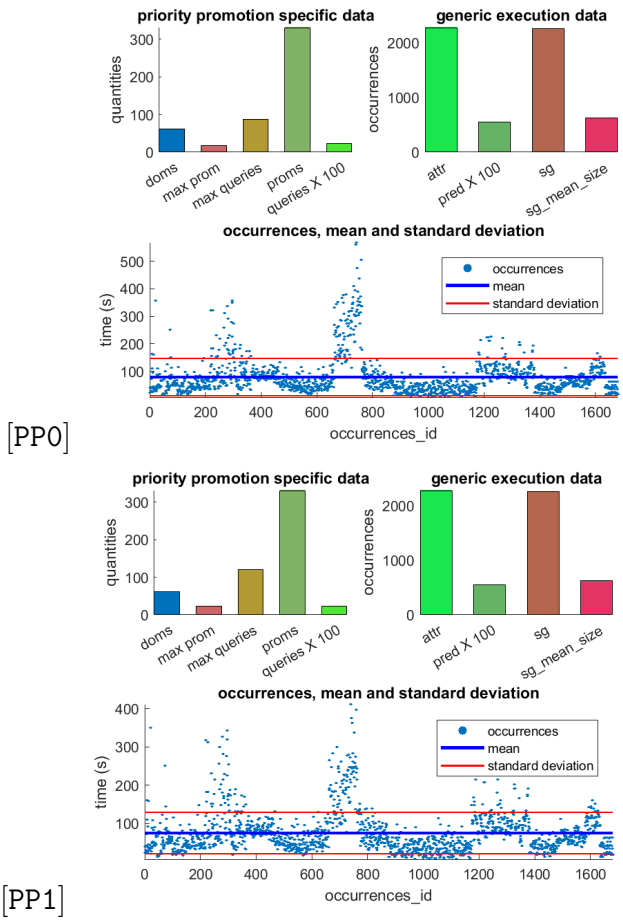


Figura 7.9: Dati relativi alle performance su tutto il *dataset*, della versione base di Priority Promotion `PP0` e della sua variante che utilizza uno stack per evitare il ricalcolo dei sottogiochi ad ogni promozione `PP1`

Una prima osservazione da fare è che i dati osservati per entrambe le versioni rispetto ai dati specifici dell'algoritmo sono identici, la variante proposta infatti, non modifica il procedimento dell'algoritmo, ma semplicemente tenta di velocizzarne l'esecuzione a prezzo di un utilizzo maggiore della memoria. Risulta poi evidente, il gran numero di attrattori effettuati e la taglia media decisamente elevata dei sottogiochi manipolati, in rapporto alle versioni migliorate di Zielonka. Dal grafico proposto non si evincono particolari infromazioni relativamente all'efficacia della variante apportata, per questo motivo di seguito è effettuato un confronto diretto dei tempi:

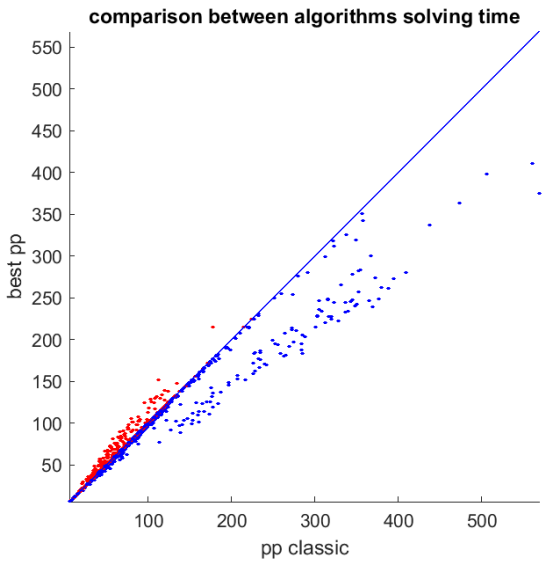


Figura 7.10: Confronto (in secondi) tra la `PP0` (*pp classic*) e `PP1` (*best pp*) su tutto il dataset

Il grafico in figura 7.10 mostra un interessante, risvolto. Per giochi "facili", tenere in memoria uno stack ausiliario, non è di nessun aiuto e anzi diventa controproducente. Man mano che però i giochi aumentano di complessità, e quindi effettuano più promozioni, la variante adottata diventa sempre più efficace, risultando vincente nella quasi totalità dei casi.
Come per gli algoritmi di Zielonka, di seguito è investigato più a fondo, il comportamento di PP rispetto ai diversi tipi di gioco:

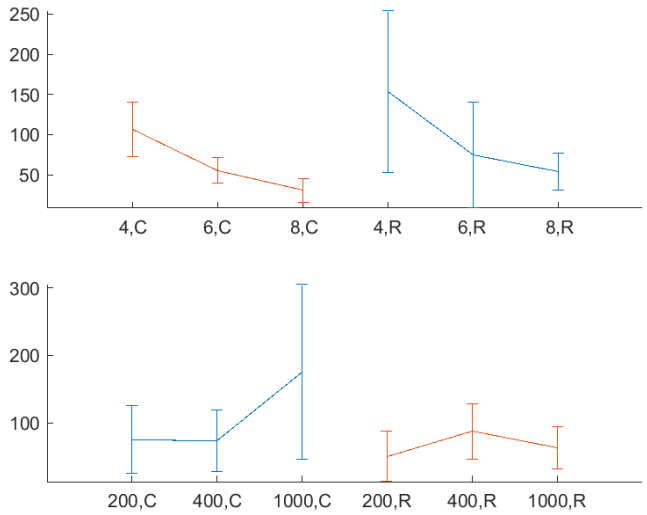
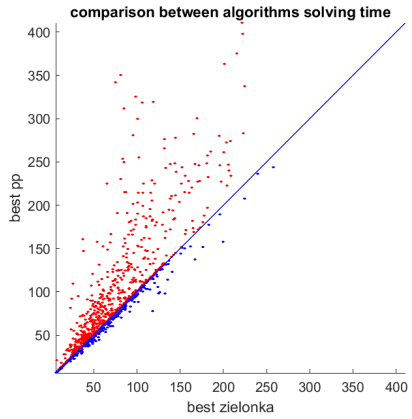


Figura 7.11: Prestazioni di `PP1` rispetto alle diverse tipologie di giochi esaminate

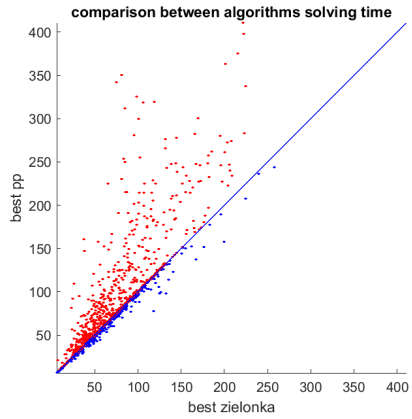
Da quanto esposto nelle figure 7.11 e 7.8 risulta evidente che i due algoritmi, PP e Zielonka, trovano difficoltà nel risolvere le stesse tipologie di giochi. Anche PP infatti impiega più tempo per giochi con più priorità e meno gruppi di transizioni.

7.2.3 Prestazioni a confronto

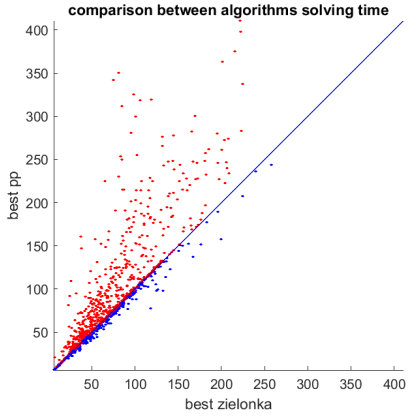
Dopo aver esaminato singolarmente le prestazioni di ciascuno dei due algoritmi con le relative varianti, di seguito è effettuato un confronto diretto tra le due versioni risultate più efficaci:



[a]



[b]



[c]

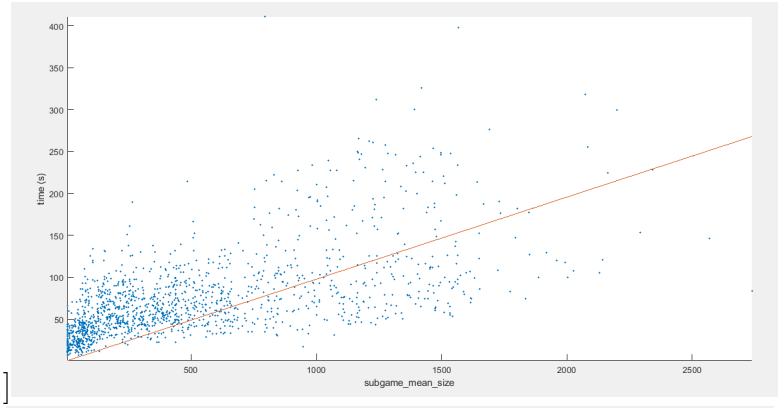
Figura 7.12: Confronto (in secondi) delle prestazioni tra la miglior versione di Zielonka e la migliore di Priority Promotion (`ZL2` vs.`PP1`). I confronti sono effettuati su tutto il dataset (a), sui giochi di tipo *clusteredrandomgame* (b) e sui giochi di tipo *randomgame*.

Inoltre si è cercato di trovare quali fossero le caratteristiche dei giochi in grado di influenzare la performance di ciascun algoritmo al di la delle proprietà basilari dei giochi, quali priorità e numero di transizioni, esaminate in precedenza. Per questo motivo per `ZL2` e `PP2` si è esaminato l'indice di correlazione di Spearman

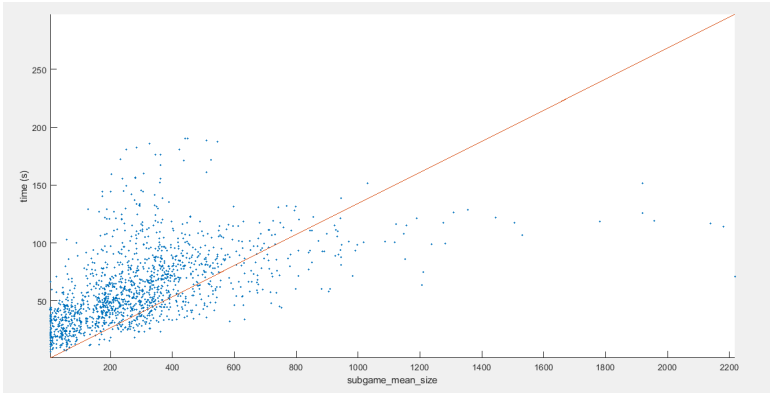
[Zar, 2008] esistente tra i tempi di risoluzione e i seguenti valori:

- numero di attrattori eseguiti dall'algoritmo,
- numero di operazioni di calcolo dei predecessori eseguiti dall'algoritmo,
- taglia media dei sottogiochi risolti dall'algoritmo,
- numero di sottogiochi risolti dall'algoritmo.

Per entrambi gli algoritmi, l'unico valore tra questi ad aver riportato un indice di correlazione con i tempi di risoluzione superiore al 60% è risultato essere la taglia media dei sottogiochi risolti. Per `ZL2` la correlazione di Spearman tra tempo di risoluzione e grandezza media dei sottogiochi, misurata su tutto il dataset risulta essere del $74,1\%$, mentre per `PP1` del $71,7\%$.



[PP1]



[ZL2]

Figura 7.13: Corrispondenza su tutto il dataset, tra tempi di risoluzione (in secondi) e taglia media dei sottogiochi (in numero di stati) corredata di regressione lineare.

In conclusione, benchè l'algoritmo classico di Zielonka offra delle prestazioni non eccezionali, tramite le migliorie proposte, giunge ad ottenere ottimi risultati.

Il confronto sembra andare piuttosto nettamente a sfavore di Priority Promotion il quale, in tutti e tre i casi di studio, ottiene prestazioni peggiori su gran parte dei giochi, in particolare con l'aumentare dei tempi di risoluzione.
Bisogna però dire che Zielonka non è migliore in assoluto, in quanto, anche per giochi complessi, talvolta PP ottinene buoni risultati. Questa condizione suggerisce la possibiltà che PP non sia in necessariamente sempre meno performante, ma che, nonostante tutte le verifiche fatte in questo elaborato, la versione simbolica di PP possa essere favorita su giochi con strutture particolari, diverse da quelle esaminate.

7.2.4 Esempi di prestazioni su modelli reali

A riprova di quanto detto al termine della sezione precedente, sono riportati di seguito, alcuni test, effettuati utilizzando l'hardware descritto all'inizio di [7.2], su altri tipi di PG proposti in [Kant and van de Pol, 2014], in cui non emerge con chiarezza un *solver* migliore. Benchè essi non rappresentino un campione statistico in alcun modo rilevante, offrono comunque un'idea di come i *solver* si comportano, quando chiamati ad agire su PG modellati da problemi reali.
I PG considerati sono infatti espressione dei seguenti *transition system* e delle seguenti proprietà:

- $four5 \times 4$, $four6 \times 5$, $four7 \times 6$:
  - *transition systems*: *board* del gioco forza quattro di grandezza rispettivamente (5×4), (6×5), (7×6)
  - proprietà: il giocatore giallo ha una strategia vincente.
- *nodeadlocks*:
  - *transition system*: il rilevatore Compact Muon Solenoid (CMS), parte del Large Hadron Collider (LHC), che altro non è che l'acceleratore di particelle, attualmente utilizzato al CERN,
  - proprietà: il sistema, non può giungere in una configurazione di stallo.

Di seguito la tabella delle caratteristiche dei giochi e delle prestazioni dei *solver*[1]:

Tabella 7.2: Caratteristiche dei giochi e prestazioni dei *solver*.

Gioco	# Stati	# Variabili	# Priorità	# Gruppi	ZL0 (s)	ZL1 (s)	ZL2 (s)	PP0 (s)	PP1 (s)
$four7 \times 6$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
$four6 \times 5$	$\sim 4,8 \cdot 10^{11}$	32	4	53	n/a	n/a	n/a	n/a	n/a
$four5 \times 4$	$\sim 1,4 \cdot 10^{8}$	22	4	29	287.98	227.15	228.27	217.17	220.97
$nodeadlocks$	$\sim 1,4 \cdot 10^{7}$	89	3	181	70.85	70.85	73.30	92.44	90.86

Come si può notare, alcuni dati risultano non disponibili, ciò è dovuto al superamento dei limiti di memoria imposti (illustrati all'inizio di [7.2]). In particolare:

- $four7 \times 6$ eccede i limiti di memoria già durante la fase d'instanziazione del gioco (descritta in [3.3]) per tanto non è possibile fornire nemmeno i dati sulla struttura del PG,
- $four6 \times 5$ invece viene correttamente generato, ma ancora una volta vengono superati i limiti di memoria, questa volta dai *solver*, nessuno di essi riesce infatti a terminare l'esecuzione rimanendo sotto i valori imposti.

Tabella 7.3: Informazioni aggiuntive delle versioni di Zielonka.

Gioco	# *Solver*	# 1° chiamate ricorsive	# 2° chiamate ricorsive	# Attrattori
$four5 \times 4$	ZL0	3	1	4
$four5 \times 4$	ZL1	2	0	2
$four5 \times 4$	ZL2	2	0	2
$nodeadlocks$	ZL0	2	0	2
$nodeadlocks$	ZL1	2	0	2
$nodeadlocks$	ZL2	2	0	2

Tabella 7.4: Informazioni aggiuntive delle versioni di Priority Promotion.

Gioco	# *Solver*	# Domini	# Max prom. per dom.	# Max query per dom.	# Prom.	# Query	# Attr.
$four5 \times 4$	PP0	2	0	1	0	2	4
$four5 \times 4$	PP1	2	0	1	0	2	4
$nodeadlocks$	PP0	2	0	1	0	2	4
$nodeadlocks$	PP1	2	0	1	0	2	4

Dalle tabelle [7.3] e [7.4] emerge chiaramente che i giochi proposti non sono algoritmicamente molto probanti, in quanto le versioni di Priority Promotion non effettuano promozioni e le versioni ZL1 e ZL2 di Zielonka riducono il numero di

[1] il backend utilizzato per effettuare tali test è stato lddmc e non CuDD come per gli altri giochi, ciò al fine di occupare meno spazio in memoria in quanto lddmc usa una variante di BDD più compatta rispetto a CuDD [3.2.2]

chiamate ricorsive su $four5 \times 4$ di molto poco rispetto a ZL0 e non ne riducono affatto il numero per $nodeadlocks$. Ad ogni modo si può notare come in un caso sia Priority Promotion ad avere i migliori risultati, mentre nell'altro Zielonka.

Appendice A

Implementazioni

Di seguito sono riportate le implementazioni realtive agli algoritmi descritti nei capitoli [5] e [6], al fine di rendere più chiaro quanto illustrato in precedenza.

1 $states_generation(PG, vars_number)$

```
state_number ← 2^vars_number
init_BDD_vars(vars_number)
player_var ← Rand(vars)
vars_encode[vars_number]
vars_player_0_encode[vars_number]
vars_player_1_encode[vars_number]
i ← 0
for var in vars do
    vars_encode[i] ← −1
    if var == player_variable then
        vars_player_0_encode[i] ← 0
        vars_player_1_encode[i] ← 1
    end if
    i ← i + 1
end for
vset_add_subset(PG.V, vars_encode)
vset_add_subset(PG.v_player[player_0], vars_player_0_encode)
vset_add_subset(PG.v_player[player_1], vars_player_1_encode)
```

2 *attractor*(*PG*, *attractor*, *player*)

```
state_set ← vset_create()
repeat
    state_set ← attractor
    partial_pr ← vset_create()
    pred ← vset_create()
    for group in groups do
        vset_prev(partial_pr, attractor, PG.e[group], PG.v_player[player])
        vset_union(pred, partial_pr)
    end for
    vset_union(attractor, pred)
    pred ← vset_clear()
    for group in groups do
        vset_prev(partial_pr, attractor, PG.e[group], PG.v_player[1−player])
        vset_union(pred, partial_pr)
    end for
    succ ← vset_create()
    partial_succ ← vset_create()
    for group in groups do
        vset_next(partial_succ, pred, PG.e[group])
        vset_intersect(partial_succ, PG.v)
        vset_union(succ, partial_succ)
    end for
    vset_minus(succ, attractor)
    partial_pr ← vset_clear()
    not_taken_pred ← vset_create()
    for group in groups do
        vset_prev(partial_pr, succ, PG.e[group], PG.v_player[1 − player])
        vset_union(not_taken_pred, partial_pr)
    end for
    vset_minus(pred, not_taken_pred)
    vset_union(attractor, pred)
    pred ← vset_clear()
    succ ← vset_clear()
until vset_is_equal(attractor, state_set)
return attractor
```

3 *spg_solve_recursive*(PG)

```
result ← create_empty_result()
if vset_is_empty(PG.v) then
    return result
end if
for pl in {player_0, player_1} do
    deadlock[pl] ← PG.v_player[pl]
    for group in groups do
        no_deadlock_states ← vset_create()
        vset_prev(no_deadlock_states, PG.v, PG.e[group], PG.v)
        vset_minus(deadlock[pl], PG.e[group], no_deadlock_states)
        if !vset_is_empty(deadlock[pl]) then
            have_deadlocks[pl] = true
        end if
    end for
end for
best_priority ← PG.min_priority
repeat
    best_priority ← best_priority + 1
until vset_is_empty(PG.v_priority[best_priority])
if PG.min_priority > 0 AND have_deadlock[player_1] == true then
    best_priority ← 0
else if PG.min_priority > 1 AND have_deadlock[player_0] == true then
    best_priority ← 1
end if
player ← mod(best_priority, 2)
attracted ← PG.v_priority[best_priority]
if best_priority < 2 then
    attracted ← vset_union(attracted, deadlock[1 − player])
end if
attracted ← attactor(PG, attracted, player)
PG' ← PG
spg_game_restrict(PG', attracted)
result' ← spg_solve_recursive(PG')
if vset_is_empty(result'.win[1 − player]) then
    vset_union(result'.win[player], attracted)
    result.win[player] ← result'.win[player]
    result.win[1 − player] ← ∅
else
    attracted' ← result'.win[1 − player]
    attracted' ← attactor(PG, attracted', player)
    PG'' ← PG
    spg_game_restrict(PG'', attracted')
    result' ← spg_solve_recursive(PG'')
    result.win[player] ← result'.win[player])
    vset_union(result'.win[1 − player], attracted')
    result.win[1 − player] ← result'.win[1 − player]
end if
return result
```

4 Esc($PG, Y, player$))

```
Y_complement ← vset_create()
no_want_escape_states ← vset_create()
want_escape_states ← vset_create()
escape_routes ← vset_create()
can_avoid_escape ← vset_create()
Y_complement ← PG.v
vset_minus(Y_complement, Y)
no_want_escape_states ← PG.v_player[1 − player]
vset_intersect(no_want_escape_states, Y)
want_escape_states ← PG.v_player[player]
vset_intersect(want_escape_states, Y)
for group in groups do
    vset_next(escape_routes, want_escape_state, PG.e[group])
    vset_intersect(escape_routes, Y_complement)
    if vset_is_empty(escape_routes) == false then
        return true
    end if
end for
partial ← vset_create()
for group in groups do
    vset_prev(partial, Y, PG.e[group], no_want_escape_states)
    vset_union(can_avoid_escape, partial)
end for
vset_minus(no_want_escape_states, can_avoid_escape)
if vset_is_empty(no_want_escape_states) then
    return false
else
    return true
end if
```

Appendice B

Manuale Utente

Di seguito è illustrato nel dettaglio, come installare e utilizzare correttamente, il tool sviluppato per questo elaborato di tesi.

B.1 Installazione

Il primo passo da effettuare per poter utilizzare il software è installare le dipendenze di seguito elencate:

- popt (versione $\geq$ 1.7)[1],
- zlib[2],
- GNU make[3],
- Flex (versione $\geq$ 2.5.35)[4],
- Apache Ant (versione $\geq$ 1.7.1)[5],
- libtool (versione $\geq$ 2.2.6)[6],

[1]popt: download disponibile a http://rpm5.org/files/popt/
[2]zlib: download disponibile a http://www.zlib.net/
[3]make: download disponibile a http://www.gnu.org/software/make/
[4]Flex: download disponibile a http://flex.sourceforge.net/
[5]Apache Ant: download disponibile a http://ant.apache.org/
[6]libtool: download disponibile a http://www.gnu.org/software/libtool/

- mcrl2: l'installazione deve avvenire dal sorgente della versione 201409.1[7]. Si può installare mcrl2 anche senza le sue componenti grafiche, in quanto non utilizzate nel tool, inoltre è necessario specificare esattamente al tool il percorso d'installazione delle librerie:

```
cmake . -DCMAKE_INSTALL_PREFIX=/[percorso_locale]
    -DBUILD_SHARED_LIBS=ON
    -DMCRL2_ENABLE_GUI_TOOLS=FALSE
make
```

- PGSolver: necessario solo nel caso in cui si voglia usare anche i generatori simbolici di PG e non solo i risolutori(l'installazione deve essere globale)[8].

Dopo aver installato le dipendenze si può procedere con la configurazione del sistema vero e proprio[9] nel seguente modo:

- effettuare il passo di configurazione come segue:

```
./configure LDFLAGS=-L[percorso_librerie_mcrl2]
    CPPFLAGS=-I[percorso_sorgenti_mcrl2]
```

- installare CuDD:

```
cd cudd
./configure --enable-dddmp --enable-obj --enable-shared
    --enable-static
make
cd ..
```

- fare la *build* del tool:

```
make
```

[7]mcrl2: download disponibile a https://www.mcrl2.org/web/user_manual/historic_releases.html#historic-releases
[8]PGSolver: download disponibile a https://github.com/tcsprojects/pgsolver
[9]LTSmin: download disponibile a https://bitbucket.org/mfaella/ltsmin-sympa/src/master/

oppure

```
make CPPFLAGS="-std=c++11"
```

nel caso in cui c++11 non sia lo standard utilizzato dal compilatore,

- infine installare le funzionalità che estendono LTSmin:

```
make -f Sympa.mk [ZIELONKA_OPT_VERSION=1]
    [ATTRACTOR_OPT_VERSION=1]
```

Di default sono utilizzate le versioni dell'algoritmo standard di Zielonka e dell'attrattore di LTSmin. Le opzioni sopra citate compilano le varianti dell'algoritmo di Zielonka con meno chiamate ricorsive e di attrattore descritte rispettivamente in [5.1.1] e [5.3].

B.2 Utilizzo

B.2.1 I Risolutori

L'istruzione per utilizzare i risolutori è la seguente:

```
./sympaexe --vset=[cudd|sylvan|lddmc|tree|fdd|ldd|ldd64]
    [percorso_del_parity_game] [ZL0|ZL1|PP0|PP1]
```

- L'opzione *vset* prende in ingresso uno dei backend di LTSmin (oltre a CuDD integrato con questo elaborato) con il quale verrà effettuata la codifica del PG,
- l'ultimo parametro da passare, fa riferimento alla tecnica di risoluzione:
  - ZL0 eseguirà la versione dell'algoritmo di Zielonka compilata (standard o con l'ottimizzazione [5.1.1] in base al parametro ZIELONKA_OPT_VERSION precedentemente visto),

- ZL1 eseguirà la versione di Zielonka compilata, con l'aggiunta della variante [5.1.1] che opera su un sottogioco minimale,
- PP0 eseguirà la versione base di Priority Promotion, implementata nel tool,
- PP1 eseguirà la versione di Priority Promotion con la variante descritta in [5.2.1].

B.2.2 I Generatori

Nel caso in cui si voglia convertire un PG da un gioco esplicito generato con PGSolver, la metodologia è la seguente:

- Generare un gioco con PGSolver, ad esempio:

```
./randomgame [stati] [massima_priority] [grado_minimo_uscente_dei_nodi]
    [grado_massimo_uscente_dei_nodi] > [ file_gioco_esplicito ]
```

- poichè PGSolver genera e risolve giochi di tipo, *max parity* mentre LTSmin codifica e risolve giochi di tipo *min parity*, può essere utile, in caso si vogliano confrontare i risultati tra risoluzione esplicita e simbolica, invertire la parità del gioco esplicito come segue:

```
./transformer -mm [gioco_esplicito] > [ gioco_esplicito_invertito ]
```

- infine, per ottenere la versione simbolica del gioco:

```
./converter --vset=[cudd|sylvan|lddmc|...] [percorso_gioco_PGSolver]
    [nome_del_gioco_simbolico]
```

Se invece si vuole ottenere un gioco generato direttamente in modo simbolico l'esecuzione deve avvenire come segue:

```
./random_generator [ numero_di_variabili_per_codificare_gli_stati ]
  [numero_di_priority] [numero_di_gruppi]
  [randomgame|clusteredrandomgame|cliquegame|steadygame]
  "[parametri_aggiuntivi]"
  [QUICK_GENERATION|BETTER_GENERATION] --vset=[cudd|sylvan]
  [nome_del_gioco_simbolico]
```

- il terzo parametro si riferisce alla metodologia di mappatura seguita per le transizioni [6.2.3], nell'istruzione sono riportate le tipologie di gioco attualmente supportate,

- per quanto riguarda i parametri aggiuntivi, essi si riferiscono alla metodologia descritta al punto precedente. Nello specifico, deve essere passata in input una stringa contenente i parametri richiesti da PGSolver per la tipologia di gioco prescelta, nell'ordine corretto e senza i dati [*numero_di_priority*] e [*numero_di_nodi*] in quanto uno è già specificato come secondo parametro in ingresso, mentre l'altro è direttamente calcolabile dal numero di variabili necessarie per codificare gli stati,

- il dato [$QUICK_GENERATION|BETTER_GENERATION$] fa riferimento alle possibili tipologie di generazione del gioco:

  – generazione lenta [6.2],

  – generazione più veloce, ma meno casuale [6.2.4].

- da notare che in questo caso - -*vset* non prende in ingresso un backend qualsiasi, ma solo cudd e sylvan. Poichè infatti per implementare il generatore simbolico è stato necessario estendere le funzionalità offerte dai wrapper di LTSmin, l'implementazione concreta non è stata effettuata per tutti i backend.

Elenco delle figure

1.1 Parte di un albero degli stati di Forza Quattro. Immagine ottenuta da `https://medium.com/@gillesvandewiele/creating-the-perfect-connect-four-ai-bot-c165115557b0` 5

2.1 Esempio di Parity Game 10

2.2 Esempio di codifica di un'espressione booleana tramite BDD . . . 20

3.1 Esempio di codifica di uno stato codificato con una sequenza di interi in espressione booleana 27

3.2 File coinvolti nell'implementazione dei backend 28

3.3 Mappa delle trasformazioni possibili per ottenere un Parity Game simbolico [Kant and van de Pol, 2014] 38

4.1 Esempio di parity game 44

4.2 BDD con concatenazione delle variabili X_0 e X_1 46

4.3 BDD con concatenazione delle variabili X_0' e X_1' 46

4.4 BDD con concatenazione delle variabili X_0 e X_0' su cui è definita la transizione ((1,0),(0,0)) presente in figura 4.1 47

4.5 BDD con concatenazione delle variabili X_1 e X_1' su cui è invariante la transizione ((1,0),(0,0)) presente in figura 4.1 48

4.6 *universe*, *prime_universe* e *complete_universe* per un gioco modellato sull'esempio in figura 4.1 49

4.7 Un esempio di corretta referenziazione dei BDD [Somenzi, 2016] . 51

4.8 OR di due generiche funzioni booleane f e g con relativa codifica in BDD 54

4.9 XNOR 56

5.1 Calcolo di una *winning region in un sottogioco* PG' 63
5.2 Possibili attrattori in un PG di una *winning region* calcolata da Zielonka in un sottogioco 64

6.1 Esempio di semantica di una clausola 79
6.2 BDD che codifica l'esempio in figura 6.1 80
6.3 BDD che per qualsiasi valore delle variabili su cui è definito, restituisce valore *true* 81

7.1 Dati relativi all'esecuzione di `ZL0` su tutti i giochi. 96
7.2 Dati relativi all'esecuzione di `ZL1` su tutti i giochi. 96
7.3 Dati relativi all'esecuzione di `ZL2` su tutti i giochi. 97
7.4 A confronto i tempi di risoluzione (in secondi) per ciascun PG di `ZL0` (*worst zielonka*) con `ZL2` (*best zielonka*). 98
7.5 Confronto (in secondi) tra i tempi di risoluzione di `ZL1` (*middle zielonka*) e `ZL2` (*best zielonka*) su ciascun PG. 98
7.6 Dati relativi alle performance delle varianti di Zielonka su giochi di tipo *randomgame*. 99
7.7 Dati relativi alle performance delle varianti di Zielonka su giochi di tipo *clusteredrandomgame* 100
7.8 Dati relativi alle performance delle varianti di Zielonka filtrate per gioco (C=*clusteredrandomgame*, R=*randomgame*) in secondo luogo per gruppo (4,6,8) o per priorità (200,400,1000) 101
7.9 Dati relativi alle performance su tutto il *dataset*, della versione base di Priority Promotion `PP0` e della sua variante che utilizza uno stack per evitare il ricalcolo dei sottogiochi ad ogni promozione `PP1` 103
7.10 Confronto (in secondi) tra la `PP0` (*pp classic*) e `PP1` (*best pp*) su tutto il dataset 104
7.11 Prestazioni di `PP1` rispetto alle diverse tipologie di giochi esaminate 104

7.12 Confronto (in secondi) delle prestazioni tra la miglior versione di Zielonka e la migliore di Priority Promotion (`ZL2` vs.`PP1`). I confronti sono effettuati su tutto il dataset (a), sui giochi di tipo *clusteredrandomgame* (b) e sui giochi di tipo *randomgame*. 106
7.13 Corrispondenza su tutto il dataset, tra tempi di risoluzione (in secondi) e taglia media dei sottogiochi (in numero di stati) corredata di regressione lineare. 107

Bibliografia

[Benerecetti et al., 2016] Benerecetti, M., Dell'Erba, D., and Mogavero, F. (2016). Solving parity games via priority promotion. *Theor. Comput. Sci.*, pages 270–290.

[Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691.

[Bryant, 1992] Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318.

[de jong et al., 2000] de jong, H., Olivier, P., Stichting, C., Centrum, M., Klint, P., and Olivier, P. (2000). Efficient annotated terms. *Software: Practice and Experience*, 30.

[Emerson and Jutla, 1991] Emerson, E. A. and Jutla, C. S. (1991). Tree automata, mu-calculus and determinacy. In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 368–377, Washington, DC, USA. IEEE Computer Society.

[Emerson et al., 2001] Emerson, E. A., Jutla, C. S., and Sistla, A. P. (2001). On model checking for the μ-calculus and its fragments. *Theor. Comput. Sci.*, 258(1-2):491–522.

[Friedmann, 2011] Friedmann, O. (2011). Recursive algorithm for parity games requires exponential time. *RAIRO - Theoretical Informatics and Applications*, 45(4):449–457.

[Grädel et al., 2002] Grädel, E., Thomas, W., and Wilke, T., editors (2002). *Automata Logics, and Infinite Games: A Guide to Current Research.* Springer-Verlag New York, Inc., New York, NY, USA.

[Jurdziński, 1998] Jurdziński, M. (1998). Deciding the winner in parity games is in up co-up. *Inf. Process. Lett.*, 68(3):119–124.

[Kant and van de Pol, 2014] Kant, G. and van de Pol, J. (2014). Generating and solving symbolic parity games. In *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2014, Grenoble, France, 5th April 2014.*, pages 2–14.

[Keiren, 2015] Keiren, J. (2015). Benchmarks for parity games. In *FSEN.*

[Kupferman and Vardi, 1998] Kupferman, O. and Vardi, M. Y. (1998). Weak alternating automata and tree automata emptiness. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 224–233, New York, NY, USA. ACM.

[Nielsen, 2006] Nielsen, J.-L. (2006). Buddy-a binary decision diagram package. *Technical report, Tech Report Technical University of Denmark.*

[Obdržálek, 2006] Obdržálek, J. (2006). *Algorithmic Analysis of Parity Games.* PhD thesis, University of Edinburgh. Submitted: January 31, 2006. Examined: May 29, 2006.

[Oliver Friedmann, 2014] Oliver Friedmann, M. L. (2014). The pgsolver collection of parity game solvers.

[Somenzi, 2016] Somenzi, F. (2016). Cudd: Cu decision diagram package release 3.0.0. *University of Colorado at Boulder.*

[van Dijk and van de Pol, 2017] van Dijk, T. and van de Pol, J. (2017). Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19(6):675–696.

[Zar, 2008] Zar, J. H. (2008). *Spearman Rank Correlation Coefficient*, pages 502–505. Springer New York, New York, NY.

[Zielonka, 1998] Zielonka, W. (1998). Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200:135–183.