UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Corso di Laurea Triennale in Informatica

Dipartimento di Ignegneria Elettrica e delle Tecnologie dell'Informazione

Tesi di Laurea Triennale

TECNICHE DI GENERAZIONE DI BENCHMARK PER LA VALUTAZIONE EMPIRICA DELL' EFFICIENZA DEGLI ALGORITMI DI RISOLUZIONE DI GIOCHI SU GRAFI

TECHNIQUES OF BENCHMARK GENERATION FOR THE EMPIRICAL EVALUATION OF SOLUTION ALGORITHMS FOR GAMES ON GRAPHS

Relatore:
Ch.mo. Prof.
Fabio Mogavero

Candidato:
Giuseppe Amato
N86001984

ANNO ACCADEMICO 2021/2022

Indice

1 Introduzione **5**

1.1 Cenni di Teoria dei Giochi . . . 6

1.1.1 Cos'è . . . 6

1.1.2 Dove viene applicata . . . 6

1.2 Definizioni . . . 8

1.2.1 Il concetto di Grafo . . . 8

1.2.2 Il concetto di Arena . . . 8

1.2.2.1 Arena a due giocatori . . . 9

1.2.3 Il concetto di Gioco . . . 9

1.2.3.1 Gioco a turni . . . 9

1.2.3.2 Gioco a somma zero . . . 9

1.2.3.3 Plays e strategie . . . 9

1.2.3.4 Winning condition e tipi di giochi . . . 10

1.2.3.5 Riducibilità tra giochi su grafi . . . 11

2 GamesOnGraphTool **13**

2.1 Lo scopo del tool . . . 14

2.2 Guida all'installazione . . . 14

2.2.1 Ottenimento delle parti rilevanti . . . 14

2.2.2 Compilazione del GamesOnGraphTool . . . 15

2.3 Esecuzione del GamesOnGraphTool . . . 15

2.3.1 Comandi di creazione . . . 16

2.3.1.1 Creazione di grafi . . . 16

2.3.1.2 Creazione di arene . . . 16

2.3.1.3 Creazione di giochi . . . 17

2.3.2 Comandi di estrazione . . . 17

2.3.2.1 Estrazione del grafo da un'arena . . . 17

2.3.2.2 Estrazione dell'arena da un gioco . . . 18

2.4 Specifica dei formati . . . 18

2.4.1 Specifica dei grafi 19
2.4.2 Specifica delle arene 21
2.4.3 Specifica dei giochi 23

3 Guida per lo sviluppatore **25**
3.1 Progettazione 26
3.1.1 Struttura del codice sorgente 26
3.1.2 Design delle classi di libreria 26
3.1.2.1 La classe Graph 27
3.1.2.2 Le classi VertexExtendableGraph e NamedVertexExtendableGraph 28
3.1.2.3 Le classi VertexMetaLabeledGraph e ModifiableVertexMetaLabeledGraph 29
3.1.2.4 La classe VertexMetaLabeledExtendableGraph 29
3.1.2.5 Le classi EdgeExtendableGraph e NamedEdgeExtendableGraph 30
3.1.2.6 Le classi EdgeMetaLabeledGraph e ModifiableEdgeMetaLabeledGraph 31
3.1.2.7 La classe EdgeMetaLabeledExtendableGraph 31
3.1.2.8 La classe VertexReducibleGraph 32
3.1.2.9 La classe EdgeReducibleGraph 33
3.1.2.10 Le classi InDirectedGraph e OutDirectedGraph 33
3.1.2.11 Le classi VertexLabeledGraph ModifiableVLVertexLabeledGraph VertexMetaLabeledAndLabeledExtendableGraph VertexExtendableVertexLabeledGraph NamedVertexExtendableVertexLabeledGraph 34
3.1.2.12 Le classi EdgeLabeledGraph ModifiableELEdgeLabeledGraph EdgeMetaLabeledAndLabeledExtendableGraph EdgeExtendableEdgeLabeledGraph EdgeExtendableNamedEdgeLabeledGraph 36
3.1.2.13 La classe Arena 38
3.1.2.14 La classe VertexExtendableArena 38
3.1.2.15 La classe VertexMetaLabeledExtendableArena 38
3.1.2.16 La classe VertexMetaLabeledAndLabeledExtendableArena . . . 39
3.1.2.17 La classe OwnerModifiableArena 39
3.1.2.18 La classe MutableDirectedGraph 40
3.1.2.19 La classe MutableFullLabeledDirectedGraph 40

3.1.2.20 La classe MutableNodeLabeledDirectedGraph 40
3.1.2.21 La classe MutableEdgeLabeledDirectedGraph 40
3.1.2.22 La classe MutableDirectedArena 41
3.1.2.23 La classe MutablePositionLabeledDirectedArena 41
3.1.2.24 La classe MutableMoveLabeledDirectedArena 41
3.1.2.25 La classe MutablePositionTargetLabeledDirectedGame 41
3.1.2.26 La classe MutableMoveTargetLabeledDirectedGame 42
3.1.2.27 La classe MutablePositionPriorityLabeledDirectedGame 42
3.1.2.28 La classe MutableMovePriorityLabeledDirectedGame 42
3.1.2.29 La classe MutablePositionPayoffLabeledDirectedGame 43
3.1.2.30 La classe MutableMovePayoffLabeledDirectedGame 43
3.1.2.31 Le classi PositionTargetLabeledReachabilityGame PositionTargetLabeledSafetyGame PositionTargetLabeledBuchiGame PositionTargetLabeledCoBuchiGame 43
3.1.2.32 Le classi MoveTargetLabeledReachabilityGame MoveTargetLabeledSafetyGame MoveTargetLabeledBuchiGame MoveTargetLabeledCoBuchiGame 44
3.1.2.33 La classe PositionPriorityLabeledParityGame 44
3.1.2.34 La classe MovePriorityLabeledParityGame 44
3.1.2.35 La classe PositionLabeledMeanPayoffGame 44
3.1.2.36 La classe MoveLabeledMeanPayoffGame 44
3.1.2.37 Le classi DiscountedPayoffGame e DiscountModifiableDiscountedPayoffGame 45
3.1.2.38 La classe PositionLabeledDiscountedPayoffGame 45
3.1.2.39 La classe MoveLabeledDiscountedPayoffGame 45
3.2 Implementazione 46
3.2.1 Il CLI Lyra 46
3.2.1.1 Analisi dei comandi 46
3.2.1.2 Integrare nuovi comandi 46
3.2.2 Le classi concrete 47
3.2.2.1 I grafi 47
3.2.2.2 Le arene 48
3.2.2.3 I giochi 48
3.2.3 Funzione di generazione random 49
3.2.4 I parser 52

3.2.4.1 Parser per Parity Game 52
3.2.5 I printer 61
3.2.5.1 Printer per Parity Game 61

4 Valutazioni **65**
4.1 Testing 66
4.2 Analisi statistica 67
4.3 Conclusioni 86

Bibliografia **87**

Capitolo 1

Introduzione

Sommario

1.1 Cenni di Teoria dei Giochi **6**

1.1.1 Cos'è 6

1.1.2 Dove viene applicata 6

1.2 Definizioni **8**

1.2.1 Il concetto di Grafo 8

1.2.2 Il concetto di Arena 8

1.2.3 Il concetto di Gioco 9

1.1 Cenni di Teoria dei Giochi

1.1.1 Cos'è

I concetti della teoria dei giochi ci forniscono uno strumento per formulare, strutturare, analizzare e comprendere differenti scenari strategici.
Un gioco, secondo questa teoria, è dato da un numero di giocatori che interagiscono secondo determinate regole. Generalmente, studia situazioni di conflitto, le decisioni degli agenti e la loro interazione. Quest'ultima ha un impatto su ognuno dei giocatori che si possono, quindi, considerare interdipendenti.
Per essere più precisi: un gioco è descritto da un insieme di giocatori e la loro possibilità di giocare secondo le regole, ognuno utilizzando una propria strategia.
La teoria dei giochi ha la sua origine storica nel 1928. John Von Neumann si rese conto della possibilità di applicare i suoi approcci, i suoi studi all'analisi di problemi economici.
Nel suo libro *Theory of Games and Economic Behavior*, scritto insieme a Oskar Morgenstern nel 1944, aveva già utilizzato la sua teoria matematica in campo economico.
La pubblicazione del suo libro è, in genere, vista come il punto iniziale della moderna Teoria dei Giochi.

1.1.2 Dove viene applicata

La Teoria dei Giochi è una branca della Matematica la quale fornisce un linguaggio comune per descrivere i giochi. Quando più parti agiscono al fine di ottenere una stessa risorsa, e dunque c'è una sorta di concorrenza da analizzare, la Teoria dei Giochi può essere usata sia per spiegare comportamenti esistenti che per migliorare le strategie.
La prima è applicata da scienze che analizzano situazioni a lungo termine, come la Biologia o la Sociologia. La seconda è uno strumento di vitale importanza in scienze come l'Economia. Le compagnie utilizzano la Teoria dei Giochi per migliorare la loro situazione strategica all'interno del mercato.
Nonostante le importanti e profonde intuizioni ottenute dall'utilizzo della Teoria dei Giochi nel campo dell'Economia, Von Neumann era maggiormente interessato all'applicazione dei suoi metodi alla politica e all'arte bellica e infatti modellò l'interazione tra gli Stati Uniti e l'URSS nella Guerra Fredda rappresentandoli come due giocatori in un gioco a somma zero.

La vasta applicabilità della Teoria dei Giochi è dovuta al fatto che è uno strumento matematico context-free che può essere usato in ogni situazione di decision-making interattiva. Una lista parziale dei campi dove la teoria è applicata, insieme a esempi di alcuni problemi che vengono studiati all' interno di ciascun campo include:

- **Economia Teorica**: Un mercato in cui fornitori vendono oggetti ad acquirenti è un esempio di gioco. Ogni venditore imposta il prezzo degli oggetti che desidera vendere, e ogni acquirente decide da quale fornitore acquisterà e in che quantità. Nei modelli di mercato, la teoria dei giochi prova a predirre i prezzi che verranno piazzati insieme alla domanda per ogni oggetto, e di studiare le relazioni tra prezzi e domanda.

- **Reti**: il mondo moderno è pieno di reti; la rete telefonica e la rete internet ne sono due esempi. Ogni utente desidera ottenere il miglior servizio possibile (per esempio, per inviare e ricevere la quantità massima di informazione attraverso Internet nel più breve lasso di tempo, o per effettuare chiamate di massima qualità usando il telefono cellulare) al minor costo possibile. Un utente deve scegliere un fornitore di servizi Internet o un gestore di telefonia mobile, dove questi fornitori sono anche giocatori nel gioco, dato che impostano il prezzo del servizio che offrono. La Teoria dei Giochi cerca di predirre il comportamento di tutti i partecipanti in questi mercati. Questo gioco è più complicato dalla prospettiva dei fornitori di servizi poichè possono cooperare l'un l'altro (ad esempio usando l'uno le infrastrutture dell'altro in modo da ridurre i costi) e la teoria dei giochi può essere usato per predire quale coalizione cooperativa sarà formata e suggerisce modi di determinare una "giusta" divisione dei profitti di tale cooperazione tra i partecipanti.

- **Scienze Politiche**: i partiti politici che formano una coalizione governativa dopo le elezioni parlamentari giocano un gioco in cui il risultato è la formazione della coalizione che include alcuni dei partiti. Questa coalizione divide i ministri tra i membri della coalizione. La Teoria dei Giochi permette di sviluppare indici che misurano il potere di ogni partito politico. Tali indici possono predirre o spiegare le divisioni dei ministri del governo dati i risultati delle elezioni. Un altro ramo della teoria dei giochi suggerisce vari metodi di votazione e studia le loro proprietà.

- **Applicazioni Militari**: Una classica applicazione militare della Teoria dei Giochi modella un missile all'inseguimento di un aereo da caccia. Qual è la migliore strategia di inseguimento per il missile? Qual è la migliore strategia che il pilota dell'aereo può usare per evitare di essere colpito dal missile? Ha contribuito al campo della difesa con l'intuizione che lo studio di tali situazioni richiede pensiero strategico.

- **Biologia**: possiamo dire che anche piante e animali stanno partecipando a giochi. L'evoluzione "determina" le strategie che i fiori usano per attrarre insetti per l'impollinazione e "determina" le strategie che gli insetti usano per scegliere quali fiori visiteranno. L'introduzione della teoria dei giochi alla biologia in generale e alla biologia evolutiva in particolare spiega, talvolta sorprendentemente bene, vari fenomeni biologici.

1.2 Definizioni

In questa sezione vengono introdotte definizioni e concetti che servono alla comprensione del documento. Verrà definito il concetto di grafo, di arena, di gioco e di quest'ultimo si parlerà di giocate (o play), strategie e obiettivi (o anche condizioni di vittoria di un gioco, a tal proposito verrà specificato per ogni tipo di gioco quand'è che i giocatori partecipanti vincono).
L'argomento risulta essere molto vasto e il documento non si propone di ricoprire tutti gli aspetti relativi ai giochi su grafi, in particolar modo questa sezione vuole offrire precisazioni su argomenti che sono presi in considerazione per lo sviluppo del *GamesOnGraphTool* che questo manufatto si prefigge di studiare e discutere.

1.2.1 Il concetto di Grafo

Un *grafo* è una struttura dati astratta che può essere usata per rappresentare relazioni complesse, non lineari tra oggetti. Un grafo è costituito da *vertici (o nodi)* e *archi*. I grafi possono essere *non orientati* dove gli archi collegano due nodi simmetricamente e *orientati* in cui gli archi collegano due nodi asimmetricamente.
Formalmente, un grafo è una coppia $G = (V, E)$ in cui:

- V è l'insieme di vertici (o nodi).
- $E \subseteq \{(x, y) : (x, y) \in V^2\}$ è un insieme di *coppie ordinate* nel caso di grafi orientati ed un insieme di *coppie non ordinate* nel caso di grafi non orientati.

Due nodi che sono connessi da un arco verranno chiamati *adiacenti*.
Il *grado* di un nodo è il numero di nodi a cui è connesso.
Un *loop* è un arco che connette un nodo a se stesso.
Un *percorso* è una sequenza di nodi che sono connessi da archi.
Un *grafo etichettato* è un grafo equipaggiato con una funzione di etichettatura che associa ogni vertice ad un' etichetta che può essere numerica o alfanumerica, una funzione che associa ogni arco ad un' etichetta oppure con entrambe.
Quando le etichette fanno parte di un insieme ordinato (come quello dei numeri reali) il grafo prende il nome di *grafo pesato*.

1.2.2 Il concetto di Arena

Un' *arena* $A = \langle (S, E), g \rangle$ consiste di un *insieme finito* S di stati, un insieme E di archi ed una funzione $g : S \to \mathbb{N}_0$ che permette di associare ogni stato ad un giocatore.
La coppia (S, E) altro non è che un grafo orientato.
Formalmente si fa riferimento ai suddetti stati (o vertici, o nodi) come *posizioni* e agli archi come *mosse*.

Dunque, l'arena la si può intendere come un'estensione del grafo orientato in cui ad ogni nodo del grafo viene aggiunta una informazione in più che specifica a quale giocatore appartiene quel dato nodo.

1.2.2.1 Arena a due giocatori

Un' *arena a due giocatori* è un'arena in cui la funzione che associa ogni stato (o posizione) ad un giocatore è fatta in questo modo: $g : S \to \{1, 2\}$ e dunque S è un'insieme finito di stati che può essere partizionato in stati del giocatore-1 e stati del giocatore-2 ($S = S_1 \cup S_2$ e $S_1 \cap S_2 = \emptyset$).

1.2.3 Il concetto di Gioco

Un *gioco su grafo* è un'estensione dell'arena, vi è infatti, anche una *condizione di vittoria* e ulteriore informazione sulle posizioni, sulle mosse o su entrambi in modo dipendente dal tipo di gioco di cui si tratta.
Per convenienza assumeremo che l'insieme $E \subseteq S \times S$ di archi è tale che per ogni $s \in S$ esiste (almeno uno) $t \in S$ tale che $(s, t) \in E$. In altre parole, ogni stato ha almeno un arco uscente.

1.2.3.1 Gioco a turni

Un gioco su G che inizia da uno stato $s_0 \in S$ è giocato a turni come segue.
Se il gioco è in uno stato relativo al giocatore-1, allora il giocatore 1 sceglie l'arco uscente che determina lo stato successivo; altrimenti, se il gioco si trova in uno stato relativo al giocatore-2 sarà lui a scegliere lo stato successivo. In questo modo, il gioco risulta in una *play* da s_0.

1.2.3.2 Gioco a somma zero

Nella teoria dei giochi, un gioco a somma zero descrive una situazione in cui il guadagno o la perdita di un partecipante è perfettamente bilanciato da una perdita o un guadagno di un altro partecipante in una somma uguale e opposta. Se alla somma totale dei guadagni dei partecipanti si sottrae la somma totale delle perdite, si ottiene zero.
Sono giochi in cui vi è un vincitore ed uno sconfitto.

1.2.3.3 Plays e strategie

Una *play* è un percorso infinito $\rho = s_0 s_1 \ldots \in S^\omega$ tale che $(s_i, s_{i+1}) \in E$ per ogni $i \geq 0$.
Una *play iniziale* è il prefisso di una play. Il prefisso di lunghezza n di ρ è denotato con $\rho(n)$.
Una *strategia* per il giocatore-1 la possiamo vedere informalmente come una serie di istruzioni che descrivono come estendere il prefisso di una play. Formalmente, una strategia σ per il giocatore-1 è una funzione $\sigma : S^* S_1 \to S$ tale che $(s, \sigma(w \cdot s)) \in E$ per ogni $w \in S^*$ e $s \in S_1$.
Una play $\rho = s_o, s_1, \ldots$ segue la strategia σ se $s_{\ell+1} = \sigma(s_0, \ldots, s_\ell)$ per tutti ℓ con $s_\ell \in S_1$.
La nozione di strategia per il giocatore-2 è definita analogamente.

1.2.3.4 Winning condition e tipi di giochi

Una *winning condition* su G è un insieme $W \subseteq S^\omega$ di play. Il giocatore-1 vince una play ρ se $\rho \in W$, altrimenti vince il giocatore-2.

- **Reachability Game**: è un gioco la cui winning condition è specificata da un insieme $F \subseteq S$ di posizioni *target*:

  $$\{s_0, s_1, \ldots : s_\ell \in F \text{ per qualche } \ell \geq 0\}$$

  In modo informale possiamo dire che il giocatore-1 vince il gioco se attraversa almeno una posizione (o stato) appartenente all'insieme F, altrimenti vince il giocatore-2.

- **Safety Game**: è un gioco la cui winning condition è praticamente il duale di quella del Reachability Game ed è specificata da un insieme $F \subseteq S$ di posizioni *target*:

  $$\{s_0, s_1, \ldots : s_\ell \in F \text{ per tutti } \ell \geq 0\}$$

  Il giocatore-1 vince il gioco se tutte le posizioni che attraversa appartengono ad F altrimenti vince il giocatore-2.

Da notare che per le condizioni dei Reachability Game e Safety Game solo un numero finito di mosse è necessario per stabilire se la condizione è soddisfatta o violata rispettivamente.
Per definire condizioni infinitarie, ci riferiamo all'insieme degli elementi che si presentano infinitamente spesso in una sequenza:

$$Inf(s_0, s_1, s_2, \ldots) = \{s : s_\ell = s \text{ per infiniti } \ell \geq 0\}$$

- **Büchi Game**: è un gioco la cui winning condition è specificata da un insieme $F \subseteq S$ di posizioni *ricorrenti*:

  $$\{\rho : F \cap Inf(\rho) \neq \emptyset\}$$

  Il giocatore-1 vince il gioco se tra le posizioni che attraversa ve n'è almeno una appartenente ad F che ricorre infinite volte.

- **CoBüchi Game**: è un gioco la cui winning condition è il duale di quella del Büchi Game ed è specificata da un insieme $F \subseteq S$ di posizioni *ricorrenti*:

  $$\{\rho : Inf(\rho) \subseteq F\}$$

  Il giocatore-1 vince il gioco se tutte le posizioni che attraversa infinite volte appartengono ad F, altrimenti vince il giocatore-2.

- **Parity Game**: è un gioco la cui winning condition è specificata da una funzione di *priorità* $\Omega : S \to \mathbb{N}_0$:

$$\{s_0, s_1, s_2, \ldots : \min Inf(\Omega(s_0), \Omega(s_1), \Omega(s_2), \ldots) \text{ è pari}\}$$

  Il giocatore-1 vince il gioco se il minimo tra le priorità associate alle posizioni che incontra infinitamente spesso risulta essere pari, altrimenti vince il giocatore-2.

- **Mean-Payoff Game**: è un gioco la cui winning condition è specificata da una funzione di *payoff* $p : S \to \mathbb{R}$:

$$\{s_0, s_1, s_2, \ldots : \text{massimizza } \liminf_{n\to\infty} \frac{\sum_{i=0}^{i=n-1} p(s_i)}{n}\}$$

  Il giocatore-1 anche chiamato *max* vince se i payoff associati alle posizioni che incontra durante la play massimizzano $\liminf_{n\to\infty} \frac{\sum_{i=0}^{i=n-1} p(s_i)}{n}$, altrimenti vince il giocatore-2 anche chiamato *min*.

- **Discounted-Payoff Game**: nel caso dei Discounted-Payoff Game, la descrizione del gioco su grafo include oltre alla funzione di *payoff* anche un numero reale $0 < \lambda < 1$ chiamato *discount factor* e la condizione di vittoria diventa:

$$\{s_0, s_1, s_2, \ldots : \text{massimizza } \sum_{i=0}^{\infty} \lambda^i s_i\}$$

  Il giocatore-1 anche in questo caso chiamato *max* vince se i payoff associati alle posizioni che incontra massimizzano $\sum_{i=0}^{\infty} \lambda^i s_i$, altrimenti vince il giocatore-2 chiamato *min*.

1.2.3.5 Riducibilità tra giochi su grafi

Ora discutiamo la riducibilità tra classi di gioco.

- **Da Parity a Mean-Payoff Game**: la riduzione dei Parity Game a Mean-Payoff Game è stata presentata in [1]. La riduzione è definita sullo stesso grafo che sottende il gioco e la *funzione di riduzione* è definita come segue: per uno stato con priorità i, il payoff è $(-1)^i \cdot n^i$, dove n è il numero degli stati. A questo punto avremo che la winning condition del Parity Game sarà equivalente a quella del Mean-Payoff Game che diventa:

$$\{s_0, s_1, s_2, \ldots : \liminf_{n\to\infty} \frac{\sum_{i=0}^{i=n-1} p(s_i)}{n} \geq 0\}$$

  La domanda se un Mean-Payoff Game può essere ridotto ad un Parity Game è ancora aperta.

- **Da Mean-Payoff a Discounted-Payoff Game**: la riduzione dei Mean-Payoff Game a Discounted-Payoff Game è stata presentata in [2]. La riduzione è definita sullo stesso grafo che sottende il gioco, con la stessa *funzione di payoff*, e il *discount factor* λ definito

come $1 - \frac{1}{4 \cdot n^3 \cdot W}$ dove n è il numero degli stati e W è il massimo valore assoluto dei pesi. La domanda se un Discounted-Payoff Game può essere ridotto ad un Mean-Payoff Game è ancora aperta.

Capitolo 2

GamesOnGraphTool

Sommario

2.1 Lo scopo del tool **14**

2.2 Guida all'installazione **14**

2.2.1 Ottenimento delle parti rilevanti 14

2.2.2 Compilazione del GamesOnGraphTool 15

2.3 Esecuzione del GamesOnGraphTool **15**

2.3.1 Comandi di creazione 16

2.3.2 Comandi di estrazione 17

2.4 Specifica dei formati **18**

2.4.1 Specifica dei grafi 19

2.4.2 Specifica delle arene 21

2.4.3 Specifica dei giochi 23

2.1 Lo scopo del tool

Premesso che in questo documento vengono presentati solo una minima parte dei giochi su grafi esistenti in letteratura, e solo quelli che sono in un qualche modo riconosciuti dal tool (anche se non tutti utilizzabili e sfruttabili), la varietà degli algoritmi che operano su di essi ha fornito una buona comprensione della teoria dei giochi anche se per alcuni la complessità computazionale non è stata ancora determinata precisamente.
Tuttavia questa conoscenza teorica non risulta essere eguagliata dalla piccola quantità della sperimentazione e dell'indagine negli aspetti pratici della risoluzione dei giochi.
Lo scopo di questo tool è di fornire un aiuto alla sperimentazione e all'indagine in campo pratico: dovrebbe permettere la comparazione tra differenti algoritmi non solo per quanto riguarda la complessità asintotica in termini di tempo ma in base alle loro performance effettive su varie classi di giochi.

Il *GamesOnGraphTool*, quindi, può essere considerato l'inizio di un software molto più ampio di valutazione sperimentale della bontà e dell'efficienza degli algoritmi di risoluzione di giochi su grafi.
E' un tool di utilità che allo stadio attuale offre due comandi principali: un comando di generazione di grafi, arene e giochi ed un comando di estrazione di arena da gioco e di grafo da arena.

2.2 Guida all'installazione

L'installazione del tool richiede pochi e semplici passi che verranno descritti di seguito.

2.2.1 Ottenimento delle parti rilevanti

Attualmente il codice sorgente del tool non è pubblicato sulla piattaforma Github, è prevista una sua pubblicazione futura in modo tale che chiunque possa collaborare alla sua espansione essendo ancora in versione embrionale e avendo, a mio parere, potenzialità per poter diventare un ottimo tool di riferimento nel campo del testing di algoritmi di risoluzione di giochi su grafi.

Il codice sorgente del software, quindi, è ottenibile per ora solamente tramite richiesta diretta ed è pensato per essere compilato ed eseguito in ambiente UNIX.

Una volta ottenuto sarà necessario, per prepararsi alla compilazione, munirsi dei due moduli principali utilizzati all'interno di esso: la *libreria LEDA* e il *CLI LYRA*.

- La *libreria LEDA* è scaricabile attraverso il sito ufficiale.

  Una volta scaricata la libreria, bisognerà far partire il comando *make* al suo interno, ciò permetterà di eseguirne una sorta di installazione e la renderà pronta per essere usata.

Terminato ciò bisognerà esportare la variabile d'ambiente LD_LIBRARY_PATH avente come valore il percorso in cui è stata salvata la libreria, in questo modo:

```
$ export LD_LIBRARY_PATH=path/to/leda
```

- L'ultima versione del *CLI LYRA* è ottenibile in questo modo:

```
$ wget https://github.com/bfgroup/Lyra/archive/refs/tags/1.6.tar.gz
```

2.2.2 Compilazione del GamesOnGraphTool

Una volta terminati i passi della sezione precedente si è pronti per la compilazione vera e propria del tool. Tutti i riferimenti agli altri file del codice sorgente sono presenti all'interno di un solo file: il *main.cpp*, sarà sufficiente eseguire il compilatore dando in input solo il suddetto file, la compilazione dovrà avvenire nel seguente modo:

```
$ g++ -o gogt main.cpp \
-I path/to/leda/incl \
-L path/to/leda \
-l leda \
-l X11 \
-l m \
-I path/to/lyra/include
```

avendo cura di sostituire *path/to/leda/incl* con il percorso in cui si trova la directory *incl* della libreria leda, *path/to/lead* con il percorso in cui si trova la directory della libreria LEDA e *path/to/lyra/include* con il percorso in cui si trova la directory *include* del CLI Lyra.

2.3 Esecuzione del GamesOnGraphTool

Una compilazione avvenuta con successo, e senza errori di sorta, produrrà in output un file eseguibile di nome *gogt*, a questo punto il tool sarà utilizzabile.
Prima di procedere è consigliabile spostarsi nella cartella ove è situato l'eseguibile del tool e invocare il terminale da lì, in questo modo non sarà necessario dover specificare il percorso dell'eseguibile ogni volta che si vorrà utilizzarlo.
Per ottenere una lista di comandi ed una descrizione parametri utilizzabili è possibile eseguire il tool in questo modo:

```
$ ./gogt --help
```

Il tool prevede due categorie di comandi: un comando per la creazione di grafi, arene, giochi ed un comando per l'estrazione di arena da gioco e di grafo da arena.

2.3.1 Comandi di creazione

2.3.1.1 Creazione di grafi

L'invocazione del comando per la *creazione di grafi* deve essere fatta nel seguente modo ed è importante l'ordine con cui i parametri vengono specificati:

```
$ ./gogt create <numNodes> <minDeg> <maxDeg> [<destFilename>]
```

- *<numNodes>* indica il numero di nodi che dovrà contenere il grafo che si intende creare.
- *<minDeg>* indica il minimo numero di archi uscenti da un nodo del grafo.
- *<maxDeg>* indica il massimo numero di archi uscenti da un nodo del grafo.
- *<destFilename>* è un parametro opzionale che se indicato permette di salvare il grafo creato in un file al percorso specificato e col nome specificato, altrimenti il grafo verrà dato in output in *stdout*.

2.3.1.2 Creazione di arene

Il comando di creazione di un'arena prevede che venga creata a partire da un grafo, il comando è il seguente:

```
$ ./gogt graph2arena
```

I parametri per questo tipo di comando sono tutti opzionali e non è importante l'ordine con cui vengono specificati.

- *--arenaType = {positionLabel | movelabel | fullLabel | unlabeled}*: a seconda del valore di questo attributo è possibile etichettare l'arena sulle posizioni, sulle mosse, sia sulle posizioni che sulle mosse o lasciarlo non etichettato.
- *--minLabel <val>*: permette di specificare il valore minimo del range di interi associabili come etichetta. Se non specificato assumerà come valore di default -100.
- *--maxLabel <val>*: permette di speficare il valore massimo del range di interi associabili come etichetta. Se non specificato assumerà come valore di default 100.
- *--srcFile <path>*: permette di specificare il nome del file contenente le specifiche del grafo utilizzato come base per la creazione dell'arena. Se non specificato viene utilizzato lo *stdin*.
- *--destFile <path>*: permette di specificare il nome del file in cui verrà salvata l'arena generata. Se non specificato viene mandata in output sullo *stdout*.

2.3.1.3 Creazione di giochi

Il comando di creazione di un gioco prevede che venga creata a partire da un'arena non etichettata, il comando è il seguente:

```
$ ./gogt arena2game
```

I parametri per questo tipo di comando sono tutti opzionali e non è importante l'ordine con cui vengono specificati.

- *--gameType = {reachability | safety | buchi | coBuchi | pg | parity | meanPayoff | discountedPayoff}*: permette di specificare il tipo di gioco che si intende creare. Gli unici valori per cui, in questo stadio, il tool ha un comportamento definito e testato sono *parity* e *pg*: entrambi permettono di definire parity game, il primo seguirà le specifiche definite in [1], il secondo quelle che verranno definite nelle prossime sezioni del documento.
- *--labelType = {positionLabel | moveLabel}*: permette di specificare il tipo di etichettatura del gioco.
- *--minLabel <val>*: permette di specificare il valore minimo del range di interi associabili come etichetta. Se non specificato assumerà come valore di default 0.
- *--maxLabel <val>*: permette di specificare il valore massimo del range di interi associabili come etichetta. Se non specificato assumerà come valore di default 100.
- *--srcFile <path>*: permette di specificare il nome del file contenente le specifiche dell'arena utilizzata come base per la creazione del gioco. Se non specificato viene utilizzato lo *stdin*.
- *--destFile <path>*: permette di specificare il nome del file in cui verrà salvata il gioco generato. Se non specificato viene mandato in output sullo *stdout*.

2.3.2 Comandi di estrazione

2.3.2.1 Estrazione del grafo da un'arena

L'*estrazione del grafo da un'arena* è una operazione che prevede di eliminare dall'arena l'informazione di giocatore proprietario di una posizione ed ottenere, così, il grafo sottostante.

Il comando per effettuare questa operazione è il seguente:

```
$ ./gogt arena2graph
```

I parametri per questo tipo di comando sono tutti opzionali e non è importante l'ordine con cui vengono specificati.

- *–srcFile <path>*: permette di specificare il nome del file contenente le specifiche dell'arena utilizzata come base per l'estrazione del grafo. Se non specificato viene utilizzato lo *stdin*.
- *–destFile <path>*: permette di specificare il nome del file in cui verrà salvato il grafo estratto. Se non specificato viene mandato in output sullo *stdout*.

2.3.2.2 Estrazione dell'arena da un gioco

L'*estrazione dell'arena da un gioco* è una operazione che prevede di eliminare dal gioco l'informazione in più che permette di definire una *Winning Condition*, specificata sulle posizioni, sulle mosse o su entrambi in modo dipendente dal tipo di gioco di cui si tratta.

Il comando per effettuare questa operazione è il seguente:

```
$ ./gogt game2arena
```

I parametri per questo tipo di comando sono tutti opzionali e non è importante l'ordine con cui vengono specificati.

- *–srcFile <path>*: permette di specificare il nome del file contenente le specifiche del gioco utilizzato come base per l'estrazione del'arena. Se non specificato viene utilizzato lo *stdin*.
- *–destFile <path>*: permette di specificare il nome del file in cui verrà salvata l'arena estratta. Se non specificato viene mandato in output sullo *stdout*.

2.4 Specifica dei formati

GamesOnGraphTool si aspetta che i vari grafi, arene, giochi che devono essere generati o estratti vengano forniti secondo delle regole ben precise. Il formato consiste in una *header line* e in una lista di specifiche di nodi. La header line serve a fornire il tipo di struttura che si sta specificando, il numero di nodi che verrà utilizzato sia per questioni di controllo che per possibili ottimizzazioni ed il tipo di etichettatura. La specifica dei nodi, invece, conterrà informazioni diverse a seconda del tipo di struttura che si sta definendo: un identificativo del nodo nel file corrente, un meta-nome per il nodo, il proprietario del nodo (se si tratta di un'arena o di un gioco), una lista di adiacenti, etichettatura e meta-nomi per gli archi. L'importanza della *meta-etichettatura* sta nel fatto che tali strutture sono soggette ad alterazioni (ad esempio estrazione e generazione che è proprio ciò che il tool stesso permette di fare), in seguito a queste non è detto che l'identificativo di un nodo della struttura pre alterazione sia lo stesso di quello della struttura post alterazione in quanto esso è solamente un modo per identificarlo all'interno del file stesso, dunque se volessimo confrontare le due strutture nei due file differenti la meta-etichettatura potrebbe essere usata come identificativo in questo senso. Questo è solo uno dei suoi possibili utilizzi.

I vari formati possono essere facilmente descritti in EBNF.

2.4.1 Specifica dei grafi

$$
\begin{array}{rcl}
\langle Init \rangle & ::= & DirectedGraph \quad [\langle NodeNumber \rangle] \quad \langle Structure \rangle \\
\langle Structure \rangle & ::= & [Unlabeled \mid U] : \quad \langle UnlabeledSpecification \rangle^{+} \\
& \mid & (NodeLabel \mid N) : \quad \langle NodeLabelSpecification \rangle^{+} \\
& \mid & (EdgeLabel \mid E) : \quad \langle EdgeLabelSpecification \rangle^{+} \\
& \mid & (FullLabel \mid F) : \quad \langle FullLabelSpecification \rangle^{+}
\end{array}
$$

$$
\begin{array}{rcllll}
\langle UnlabeledSpecification \rangle & ::= & \langle Node \rangle & & [\langle UnlabeledAdjacentList \rangle] & ; \\
\langle NodeLabelSpecification \rangle & ::= & \langle Node \rangle & \langle NodeLabel \rangle & [\langle UnlabeledAdjacentList \rangle] & ; \\
\langle EdgeLabelSpecification \rangle & ::= & \langle Node \rangle & & [\langle LabeledAdjacentList \rangle] & ; \\
\langle FullLabelSpecification \rangle & ::= & \langle Node \rangle & \langle NodeLabel \rangle & [\langle LabeledAdjacentList \rangle] & ;
\end{array}
$$

$$
\langle Node \rangle \quad ::= \quad \langle NodeIdentifier \rangle \quad [\langle NodeMetaName \rangle]
$$

$$
\begin{array}{rcllll}
\langle UnlabeledAdjacentList \rangle & ::= & \langle NodeIdentifier \rangle & & & [\langle EdgeMetaName \rangle] \\
& & (, \ \langle NodeIdentifier \rangle & & & [\langle EdgeMetaName \rangle])^{*} \\
\langle LabeledAdjacentList \rangle & ::= & \langle NodeIdentifier \rangle & & \langle EdgeLabel \rangle & [\langle EdgeMetaName \rangle] \\
& & (, \ \langle NodeIdentifier \rangle & & \langle EdgeLabel \rangle & [\langle EdgeMetaName \rangle])^{*}
\end{array}
$$

$$
\begin{array}{rcl}
\langle NodeNumber \rangle, \ \langle NodeIdentifier \rangle & ::= & \langle natural\ number \rangle \\
\langle NodeLabel \rangle, \ \langle EdgeLabel \rangle & ::= & \langle numeric\ label \rangle \\
\langle NodeMetaName \rangle, \ \langle EdgeMetaName \rangle & ::= & "\langle string \rangle"
\end{array}
$$

$$
\begin{array}{rcl}
\langle numeric\ label \rangle & ::= & \langle integer\ number \rangle \\
& \mid & \langle fractional\ number \rangle \\
& \mid & \langle floating - point \rangle \\
& \mid & '(' \ \langle numeric\ label \rangle (, \ \langle numeric\ label \rangle^{+} \ ')'
\end{array}
$$

$$
\begin{array}{c}
\langle natural\ number \rangle \ \langle integer\ number \rangle \ \langle fractional\ number \rangle \ \langle floating - point \rangle \ \langle string \rangle \\
::= built - in\ data\ types
\end{array}
$$

Example 2.4.1 *Di seguito un esempio di grafo avente 5 nodi etichettato sia su nodi che su archi e con meta-etichettatura sia su nodi che su archi, definibile con la grammatica specificata.*

```
DirectedGraph 5 FullLabel:
0 "Popcorn" 10 3 20 "Napoli" , 4 5 "Milano";
1 "Patatine" 30 2 50 "Roma", 3 -20 "Firenze", 0 100 "Torino";
2 "Pizza" 65 0 -10 "Venezia";
3 "Kebab" 15 4 25 "Palermo";
4 "Crepe" 58 1 19 "Verona", 2 99 "Bologna";
```

2.4.2 Specifica delle arene

$$
\begin{array}{rcl}
\langle Init \rangle & ::= & Arena \quad [\langle PositionNumber \rangle] \quad \langle Structure \rangle \\
\langle Structure \rangle & ::= & [Unlabeled \mid U] : \quad \langle UnlabeledSpecification \rangle^{+} \\
& \mid & (PositionLabel \mid P) : \quad \langle PositionLabelSpecification \rangle^{+} \\
& \mid & (MoveLabel \mid M) : \quad \langle MoveLabelSpecification \rangle^{+} \\
& \mid & (FullLabel \mid F) : \quad \langle FullLabelSpecification \rangle^{+}
\end{array}
$$

$$
\begin{array}{rcllll}
\langle UnlabeledSpecification \rangle & ::= & \langle Position \rangle & & & [\langle UnlabeledMoves \rangle] \ ; \\
\langle PositionLabelSpecification \rangle & ::= & \langle Position \rangle & \langle PositionLabel \rangle & & [\langle UnlabeledMoves \rangle] \ ; \\
\langle MoveLabelSpecification \rangle & ::= & \langle Position \rangle & & & [\langle LabeledMoves \rangle] \ ; \\
\langle FullLabelSpecification \rangle & ::= & \langle Position \rangle & \langle PositionLabel \rangle & & [\langle LabeledMoves \rangle] \ ;
\end{array}
$$

$$
\langle Position \rangle \quad ::= \quad \langle PositionIdentifier \rangle \quad [\langle PositionMetaName \rangle] \quad \langle Owner \rangle
$$

$$
\begin{array}{rcllll}
\langle UnlabeledMoves \rangle & ::= & \langle PositionIdentifier \rangle & & & [\langle MoveMetaName \rangle] \\
& & (, \ \langle PositionIdentifier \rangle & & & [\langle MoveMetaName \rangle])^{*} \\
\langle LabeledMoves \rangle & ::= & \langle PositionIdentifier \rangle & & \langle MoveLabel \rangle & [\langle MoveMetaName \rangle] \\
& & (, \ \langle PositionIdentifier \rangle & & \langle MoveLabel \rangle & [\langle MoveMetaName \rangle])^{*}
\end{array}
$$

$$
\begin{array}{rcl}
\langle PositionNumber \rangle, \ \langle PositionIdentifier \rangle & ::= & \langle natural\ number \rangle \\
\langle Owner \rangle & ::= & 0 \mid 1 \\
\langle PositionLabel \rangle, \ \langle MoveLabel \rangle & ::= & \langle numeric\ label \rangle \\
\langle PositionMetaName \rangle, \ \langle MoveMetaName \rangle & ::= & "\langle string \rangle"
\end{array}
$$

$$
\begin{array}{rcl}
\langle numeric\ label \rangle & ::= & \langle integer\ number \rangle \\
& \mid & \langle fractional\ number \rangle \\
& \mid & \langle floating - point \rangle \\
& \mid & '(' \ \langle numeric\ label \rangle (, \ \langle numeric\ label \rangle^{+} \ ')'
\end{array}
$$

$$
\begin{array}{c}
\langle natural\ number \rangle \ \langle integer\ number \rangle \ \langle fractional\ number \rangle \ \langle floating - point \rangle \ \langle string \rangle \\
::= built - in\ data\ types
\end{array}
$$

Example 2.4.2 *Di seguito un esempio di arena avente 5 posizioni etichettata e meta-etichettata sia sulle posizioni che sulle mosse, definibile con la grammatica specificata.*

```
Arena 5 FullLabel:
0 "Popcorn" 0 10 3 20 "Napoli" , 4 5 "Milano";
1 "Patatine" 1 30 2 50 "Roma", 3 -20 "Firenze", 0 100 "Torino";
2 "Pizza" 0 65 0 -10 "Venezia";
3 "Kebab" 1 15 4 25 "Palermo";
4 "Crepe" 0 58 1 19 "Verona", 2 99 "Bologna";
```

2.4.3 Specifica dei giochi

$$
\begin{array}{rcllll}
\langle Init \rangle & ::= & Reachability & [\langle PositionNumber \rangle] & \langle TargetStructure \rangle \\
 & | & Safety & [\langle PositionNumber \rangle] & \langle TargetStructure \rangle \\
 & | & Buchi & [\langle PositionNumber \rangle] & \langle TargetStructure \rangle \\
 & | & CoBuchi & [\langle PositionNumber \rangle] & \langle TargetStructure \rangle \\
 & | & PG & [\langle PositionNumber \rangle] & \langle ParityStructure \rangle \\
 & | & MPG & [\langle PositionNumber \rangle] & \langle PayoffStructure \rangle \\
 & | & DPG & [\langle PositionNumber \rangle] & \langle DiscountedStructure \rangle
\end{array}
$$

$$
\begin{array}{rcllll}
\langle TargetStructure \rangle & ::= & ToP & : & \langle TargetLabeledPositionSpecification \rangle^{+} \\
 & | & ToM & : & \langle TargetLabeledMoveSpecification \rangle^{+} \\
\langle ParityStructure \rangle & ::= & PoP & : & \langle PriorityLabeledPositionSpecification \rangle^{+} \\
 & | & PoM & : & \langle PriorityLabeledMoveSpecification \rangle^{+} \\
\langle PayoffStructure \rangle & ::= & PoP & : & \langle PayoffLabeledPositionSpecification \rangle^{+} \\
 & | & PoM & : & \langle PayoffLabeledMoveSpecification \rangle^{+} \\
\langle DiscountedStructure \rangle & ::= & & & \langle Discount \rangle \quad \langle PayoffStructure \rangle
\end{array}
$$

$$
\begin{array}{rcllll}
\langle TargetLabeledPositionSpecification \rangle & ::= & \langle Position \rangle & \langle TargetLabel \rangle & [\langle UnlabeledMoves \rangle] \ ; \\
\langle TargetLabeledMoveSpecification \rangle & ::= & \langle Position \rangle & & [\langle TargetLabeledMoves \rangle] \ ; \\
\langle PriorityLabeledPositionSpecification \rangle & ::= & \langle Position \rangle & \langle PriorityLabel \rangle & [\langle UnlabeledMoves \rangle] \ ; \\
\langle PriorityLabeledMoveSpecification \rangle & ::= & \langle Position \rangle & & [\langle PriorityLabeledMoves \rangle] \ ; \\
\langle PayoffLabeledPositionSpecification \rangle & ::= & \langle Position \rangle & \langle PayoffLabel \rangle & [\langle UnlabeledMoves \rangle] \ ; \\
\langle PayoffLabeledMoveSpecification \rangle & ::= & \langle Position \rangle & & [\langle PayoffLabeledMoves \rangle] \ ;
\end{array}
$$

$$
\begin{array}{rcllll}
\langle TargetLabeledMoves \rangle & ::= & \langle PositionIdentifier \rangle & \langle TargetLabel \rangle & [\langle MoveMetaName \rangle] \\
 & & (, \langle PositionIdentifier \rangle & \langle TargetLabel \rangle & [\langle MoveMetaName \rangle])^{*} \\
\langle PriorityLabeledMoves \rangle & ::= & \langle PositionIdentifier \rangle & \langle PriorityLabel \rangle & [\langle MoveMetaName \rangle] \\
 & & (, \langle PositionIdentifier \rangle & \langle PriorityLabel \rangle & [\langle MoveMetaName \rangle])^{*} \\
\langle PayoffLabeledMoves \rangle & ::= & \langle PositionIdentifier \rangle & \langle PayoffLabel \rangle & [\langle MoveMetaName \rangle] \\
 & & (, \langle PositionIdentifier \rangle & \langle PayoffLabel \rangle & [\langle MoveMetaName \rangle])^{*}
\end{array}
$$

$$
\begin{array}{rcl}
\langle Discount \rangle, \ \langle PayoffLabel \rangle & ::= & \langle fractional\ number \rangle \\
 & | & \langle floating - point \rangle \\
\langle TargetLabel \rangle & ::= & 0 \ | \ 1 \\
\langle PriorityLabel \rangle & ::= & \langle natural\ number \rangle
\end{array}
$$

Example 2.4.3 *Di seguito un esempio di Discouted-Payoff Game avente 5 posizioni etichettata sulle posizioni, con meta-etichettatura sia sulle posizioni che sulle mosse ed avente un fattore di sconto di 0.7, definibile con la grammatica specificata.*

```
DPG 5 0.7 PoP:
0 "Popcorn" 0 1.1 3 "Napoli" , 4 "Milano";
1 "Patatine" 1 3.2 2 "Roma", 3 "Firenze", 0 "Torino";
2 "Pizza" 0 6.5 0 "Venezia";
3 "Kebab" 1 1.5 4 "Palermo";
4 "Crepe" 0 5.8 1 "Verona", 2 "Bologna";
```

Capitolo 3

Guida per lo sviluppatore

Sommario

3.1 Progettazione **26**

3.1.1 Struttura del codice sorgente 26

3.1.2 Design delle classi di libreria 26

3.2 Implementazione **46**

3.2.1 Il CLI Lyra 46

3.2.2 Le classi concrete 47

3.2.3 Funzione di generazione random 49

3.2.4 I parser 52

3.2.5 I printer 61

Lo scopo di questo capitolo è fornire una visione sufficientemente buona della struttura del GamesOnGraphTool in modo che sia possibile estenderlo con nuovi moduli o algoritmi o, addirittura, migliorare quelli esistenti. Essendo un tool ancora in fase embrionale ed avendo ottime potenzialità nel campo della risoluzione di giochi su grafi questa parte risulta essere di enorme importanza.

3.1 Progettazione

3.1.1 Struttura del codice sorgente

Il codice sorgente è diviso in directory e subdirectory per facilitare la leggibilità e la comprensione del tutto.

- *ext* contiene tutto ciò che è esterno al tool stesso.
- *ext/tools* contiene due tool di utilità: *rngame* e *oink* che è possibile trovare qui [1]. Il primo utilizzato per la creazione di giochi di parità, il secondo utilizzato per la risoluzione degli stessi.
- *lib* contiene tutti i file di libreria, comprende sia la parte astratta che la parte implementativa concreta in due moduli differenti.
- *lib/abstract* contiene tutta la struttura astratta che definisce come il tool è stato progettato.
- *lib/concrete* contiene la parte implementativa, quella che lavora al di sotto della parte astratta e che definisce il comportamento del tool.
- *prg/cli* contiene tutto ciò che è relativo al command-line interface. Ogni comando utilizzabile dal tool è definito in una classe separata.
- *script* contiene tutti gli script utilizzati per la generazione dei test.
- *test* contiene i risultati dei test di generazione ed estrazione. Contiene, inoltre, i tempi di risoluzione calcolati sui giochi generati sia con il tool *rngame* sia con il tool *GamesOnGraphTool*.

3.1.2 Design delle classi di libreria

Lo sviluppo del tool è stato eseguito seguendo uno dei principi cardini della buona programmazione: *"Programma tramite le interfacce non tramite l'implementazione"* ciò ha permesso di separare il "cosa" dal "come" che rende estremamente facile e conveniente il miglioramento del codice già esistente e l'evoluzione futura con l'introduzione di nuove funzionalità.

Un'altra regola tenuta in mente durante tutto il tempo di sviluppo del tool è stata l'*l'alta coesione* che è una misura di quanto siano fortemente relate e mirate le responsabilità di un modulo. Quando ciascuna unità è responsabile di un singolo compito possiamo dire che tale unità ha un'alta coesione. Uno dei benefici apportati da questo principio è la facilità nella comprensione dei ruoli di una classe e nella manutenzione di questa. Si è puntato molto su questo principio in quanto si è supposto che il tool potesse essere ampliato in futuro possibilmente anche non dallo stesso team di sviluppatori il che significa che la comprensibilità è una caratteristica fondamentale per questo fine. Ulteriori benefici apportati sono: il riutilizzo di una stessa classe in modo da non avere ridondanze, l'utilizzo di nomi appropriati, efficaci e comunicativi, il vantaggio di limitare e focalizzare i cambiamenti nel codice.
Ogni classe (astratta) progettata all'interno del tool rappresenta un singolo concetto ben definito ed ogni metodo è responsabile di un singolo compito ben delineato, ciò ha un motivo: chi si ritrova ad implementare il codice e a dare un comportamento alla parte astratta deve essere libero di poter realizzare solo le funzionalità che vuole senza dover per forza concretare metodi che non sono necessari o che addirittura potrebbero obbligare lo sviluppatore a non raggiungere l'efficienza desiderata.

Un *basso accoppiamento* è un altro dei principi utilizzati nella realizzazione del tool, si è cercato di avere meno collegamenti possibili tra le varie classi e di tenere solo quelli strettamente necessari facendo risultare semplice capire il codice di una classe senza dover andare a leggere i dettagli delle altre e permettendo di effettuarne modifiche senza che queste comportino conseguenze sulle altre classi.

Largo uso si è fatto della *programmazione generica* per aumentare l'efficienza del codice. La programmazione generica ha permesso di scrivere algoritmi meno specifici che lavorano con tutti i tipi di dato ed ha eliminato la necessità di riscrivere più e più volte lo stesso algoritmo ma che lavorasse con tipi di dato differenti.

3.1.2.1 La classe Graph

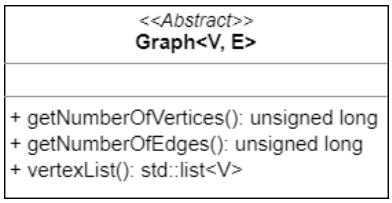

Figura 3.1: Diagramma UML della classe Graph.

Questa è la classe astratta che modella il concetto di grafo nella sua accezione più semplice. Rappresenta un grafo immutabile non etichettato, ovvero, nè estendibile nè riducibile, con metodi di sola lettura e possibilità di utilizzare un qualsiasi tipo di dato per i vertici e per gli

archi. Da questo concetto si dipartono tutti gli altri.
Nella sua attuale definizione comprende 3 semplici metodi:

- *unsigned long getNumberOfVertices()* restituisce il numero di vertici contenuti nel grafo.
- *unsigned long getNumberOfEdges()* restituisce il numero di archi contenuto nel grafo.
- *std::list<V> vertexList()* restituisce una lista dei vertici contenuti nel grafo.

3.1.2.2 Le classi VertexExtendableGraph e NamedVertexExtendableGraph

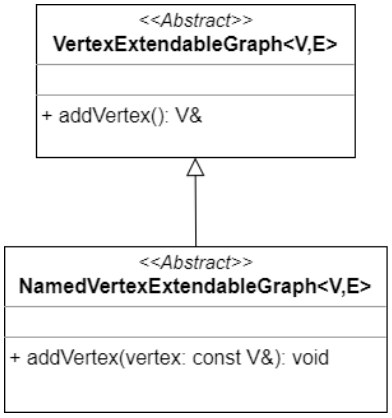

Figura 3.2: Diagramma UML delle classi VertexExtendableGraph e NamedVertexExtendable-Graph.

Entrambe le classi astratte definiscono il concetto di *grafo estendibile per i vertici*, ciò significa che un' implementazione di queste classi permetterà di costruire dei grafi in cui è possibile effettuare l'operazione di inserimento dei vertici.

La classe astratta *VertexExtendableGraph* è un'estensione della classe Graph vista nella sezione precedente e come tale eredita, quindi, tutti i suoi metodi. Il concetto che definisce è quello di grafo che permette l'inserimento di vertici dove però non ci si vuole preoccupare (o non se ne ha il bisogno) del tipo di vertice che si vuole aggiungere, lascia, quindi, la libertà di definirlo a colui che la implementa.

NamedVertexExtendableGraph è un'estensione di VertexExtendableGraph e definisce lo stesso concetto di grafo estendibile ma con la particolarità che è possibile aggiungere un qualsiasi oggetto come vertice, tale oggetto lo si potrebbe intendere come una sua denominazione, da qui deriva il nome dato alla classe.

I due metodi permetteranno l'estensione del grafo attraverso l'aggiunta di vertici.

3.1.2.3 Le classi VertexMetaLabeledGraph e ModifiableVertexMetaLabeledGraph

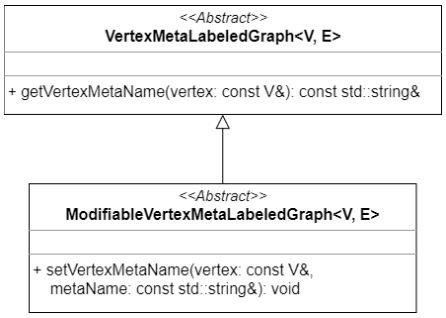

Figura 3.3: Diagramma UML delle classi VertexMetaLabeledGraph e ModifiableVertexMetaLabeledGraph.

Entrambe le classi astratte modellano il concetto di *grafo meta-etichettato sui vertici*.

La classe astratta *VertexMetaLabeledGraph* estende la classe Graph e da sola definisce tale concetto nella sua accezione più semplice: chi si ritrova a dare una concretizzazione di tale astrazione definirà un grafo non estendibile, non riducibile, non orientato, non modificabile, non etichettato i cui vertici avranno una meta-etichettatura *non modificabile* che è possibile conoscere tramite l'unico metodo che offre:

- *const std::string& getVertexMetaName(const V& vertex)* consente di conoscere la meta-etichetta associata al vertice *vertex*.

La classe astratta *ModifiableVertexMetaLabeledGraph* estende *VertexMetaLabeledGraph*, modella il suo stesso concetto ma con la particolarità di avere una meta-etichettatura che è modificabile. L'implementatore avrà quindi il compito di definire un modo per ottenere le meta-etichette associate ai vertici. Ha un solo metodo che permette questa operazione:

- *void setVertexMetaName(const V& vertex, const std::string& metaName)* consente di associare la meta-etichetta *metaName* al vertice *vertex* all'interno del grafo su cui viene effettuata la chiamata.

3.1.2.4 La classe VertexMetaLabeledExtendableGraph

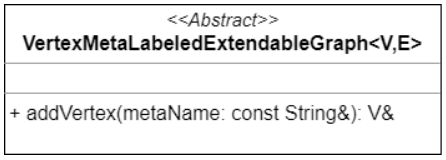

Figura 3.4: Diagramma UML della classe VertexMetaLabeledExtendableGraph.

La classe astratta *VertexMetaLabeledExtendableGraph* estende la classe Graph ereditando i suoi metodi e definisce il concetto di *grafo meta-etichettato sui vertici* estendibile per quest'ultimi. Attraverso delle implementazioni di questa classe sarà possibile creare dei grafi in cui ogni nodo avrà il proprio *metaname*.
La classe possiede un solo metodo:

- *V& addVertex(const std:string& metaName)* consente l'aggiunta di vertici con metaname associati. Restituisce il vertice appena aggiunto.

3.1.2.5 Le classi EdgeExtendableGraph e NamedEdgeExtendableGraph

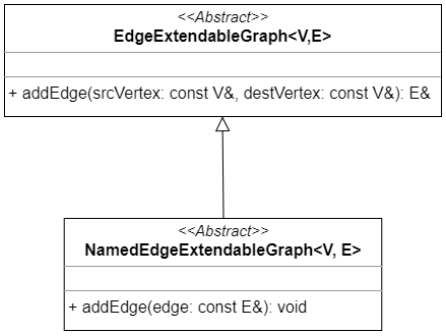

Figura 3.5: Diagramma UML delle classi EdgeExtendableGraph e NamedEdgeExtendableGraph.

Entrambe le classi astratte definiscono il concetto di *grafo estendibile per gli archi*, quindi un' implementazione di queste classi permetterà di costruire dei grafi in cui è possibile aumentare a proprio piacimento il numero di archi contenuto.

La classe astratta *EdgeExtendableGraph* è un'estensione della classe Graph. Il concetto che definisce è quello di grafo che permette l'inserimento di archi come semplice collegamento tra due vertici dove chi implementa decide in che modo concretamente rappresentare tali archi.

NamedVertexExtendableGraph è un'estensione di EdgeExtendableGraph e definisce lo stesso concetto di grafo estendibile ma con la particolarità che colui che utilizza la classe ha la responsabilità di definire l'arco.

I due metodi permetteranno l'estensione del grafo attraverso l'aggiunta di archi:

- *E& addEdge(const V& src, const V& dest)* consente l'aggiunta di un arco che collega il vertice *src* al vertice *dest*. Restituisce l'arco appena creato.

- *void addEdge(const E& edge)* consente l'aggiunta dell'arco *edge* al grafo.

3.1.2.6 Le classi EdgeMetaLabeledGraph e ModifiableEdgeMetaLabeledGraph

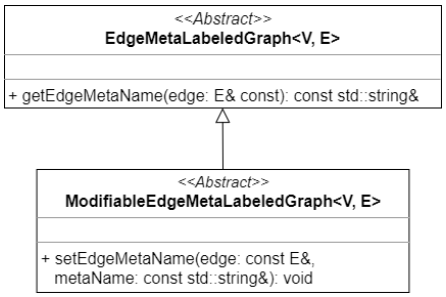

Figura 3.6: Diagramma UML delle classi EdgeMetaLabeledGraph e ModifiableEdgeMetaLabeledGraph.

Entrambe le classi astratte modellano il concetto di *grafo meta-etichettato sugli archi*.

La classe astratta *EdgeMetaLabeledGraph* estende la classe Graph e da sola definisce tale concetto nella sua accezione più semplice: chi si ritrova a dare una concretizzazione di tale astrazione definirà un grafo non estendibile, non riducibile, non orientato, non modificabile, non etichettato i cui archi avranno una meta-etichettatura *non modificabile* che è possibile conoscere tramite l'unico metodo che offre:

- *const std::string& getEdgeMetaName(const E& edge)* consente di conoscere la meta-etichetta associata all'arco *edge*.

La classe astratta *ModifiableEdgeMetaLabeledGraph* estende *EdgeMetaLabeledGraph*, modella il suo stesso concetto ma con la particolarità di avere una meta-etichettatura che è modificabile. L'implementatore avrà quindi il compito di definire un modo per ottenere le meta-etichette associate agli archi. Ha un solo metodo che permette questa operazione:

- *void setEdgeMetaName(const E& edge const std::string& metaName)* consente di associare la meta-etichetta *metaName* all'arco *edge* all'interno del grafo su cui viene effettuata la chiamata.

3.1.2.7 La classe EdgeMetaLabeledExtendableGraph

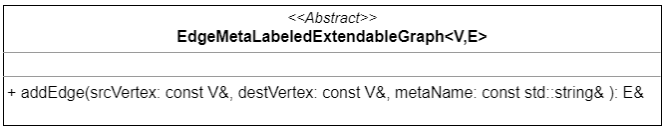

Figura 3.7: Diagramma UML della classe EdgeMetaLabeledExtendableGraph.

La classe astratta *EdgeMetaLabeledGraph* estende la classe Graph ereditando i suoi metodi e definisce il concetto di *grafo meta-etichettato sugli archi* estendibile per quest'ultimi. Attraverso

delle implementazioni di questa classe sarà possibile creare dei grafi in cui ogni arco avrà il proprio *metaname*.
La classe possiede un solo metodo:

- *E& addEdge(const V& src, const V& dest, const std::string& metaname)* consente l'aggiunta di un arco che collega il vertice *src* al vertice *dest* con metaname associato. Restituisce l'arco appena aggiunto.

3.1.2.8 La classe VertexReducibleGraph

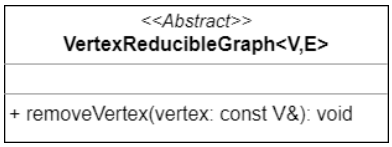

Figura 3.8: Diagramma UML della classe VertexReducibleGraph.

La classe astratta *VertexReducibleGraph* estende la classe Graph ereditando i suoi metodi e definisce il concetto di *grafo riducibile per i vertici*. Attraverso delle implementazioni concrete di questa classe saranno realizzabili dei grafi in cui è possibile diminuire il numero di vertici tramite eliminazione di questi dalla struttura.
La classe possiede un solo metodo:

- *void removeVertex(const V& vertex)* consente la cancellazione del vertice *vertex* dal grafo.

3.1.2.9 La classe EdgeReducibleGraph

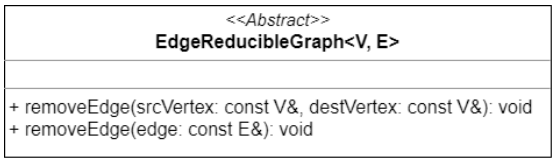

Figura 3.9: Diagramma UML della classe EdgeReducibleGraph.

La classe astratta *EdgeReducibleGraph* estende la classe Graph e definisce il concetto di *grafo riducibile per gli archi*. Attraverso delle implementazioni concrete saranno realizzabili dei grafi in cui è possibile diminuire il numero di archi eliminandoli.
Possiede due metodi di eliminazione, uno permette di specificare i due vertici dai quali si vuole rimuovere l'arco che li congiunge, l'altro permette di specificare direttamente l'arco da cancellare, lasciando così la possibilità a chi utilizzerà tale classe di scegliere il metodo migliore secondo le proprie necessità:

- *void removeEdge(const V& src, const V& dest)* permette la cancellazione dell'arco che collega il vertice *src* al vertice *dest* dal grafo.
- *void removeEdge(const V& edge)* permette la cancellazione dell'arco *edge* dal grafo.

3.1.2.10 Le classi InDirectedGraph e OutDirectedGraph

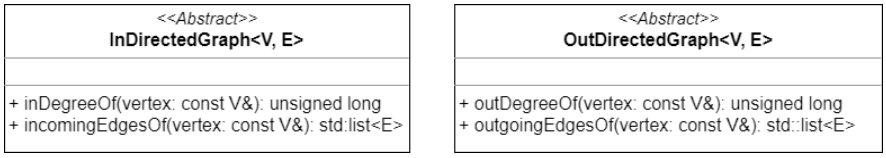

Figura 3.10: Diagramma UML delle classi InDirectedGraph e OutDirectedGraph.

Entrambe le classi astratte estendono la classe Graph e definiscono il concetto di *grafo orientato*, quindi una loro implementazione permetterà di costruire degli oggetti su cui è possibile effettuare le operazioni tipiche dei grafi orientati. Qui è stata effettuata un'ulteriore distinzione tra grafo su cui è possibile effettuare solo operazioni sugli archi entranti e grafo su cui è possibile effettuare solo operazioni sugli archi uscenti, in modo tale da lasciare estrema libertà di implementazione a colui che si occuperà di sviluppare una concretizzazione delle due.

La classe astratta *InDirectedGraph* è quella che permette di eseguire operazioni sugli archi entranti di un vertice.
Possiede i seguenti metodi:

- *unsigned long inDegreeOf(const V& vertex)* consente di conoscere il numero di archi entranti del vertice *vertex* all'interno del grafo su cui viene effettuata la chiamata.

- *std::list<E> incomingEdgesOf(const V& vertex)* consente di ottenere una lista degli archi entranti del vertice *vertex* contenuti nel grafo su cui viene effettuata la chiamata.

La classe astratta *OutDirectedGraph* è quella che permette di eseguire operazioni sugli archi uscenti di un vertice.
Possiede i seguenti metodi:

- *unsigned long outDegreeOf(const V& vertex)* consente di conoscere il numero di archi uscenti del vertice *vertex* all'interno del grafo su cui viene effettuata la chiamata.
- *std::list<E> outgoingEdgesOf(const V& vertex)* consente di ottenere una lista degli archi uscenti del vertice *vertex* contenuti nel grafo su cui viene effettuata la chiamata.

3.1.2.11 Le classi VertexLabeledGraph ModifiableVLVertexLabeledGraph VertexMetaLabeledAndLabeledExtendableGraph VertexExtendableVertexLabeledGraph NamedVertexExtendableVertexLabeledGraph

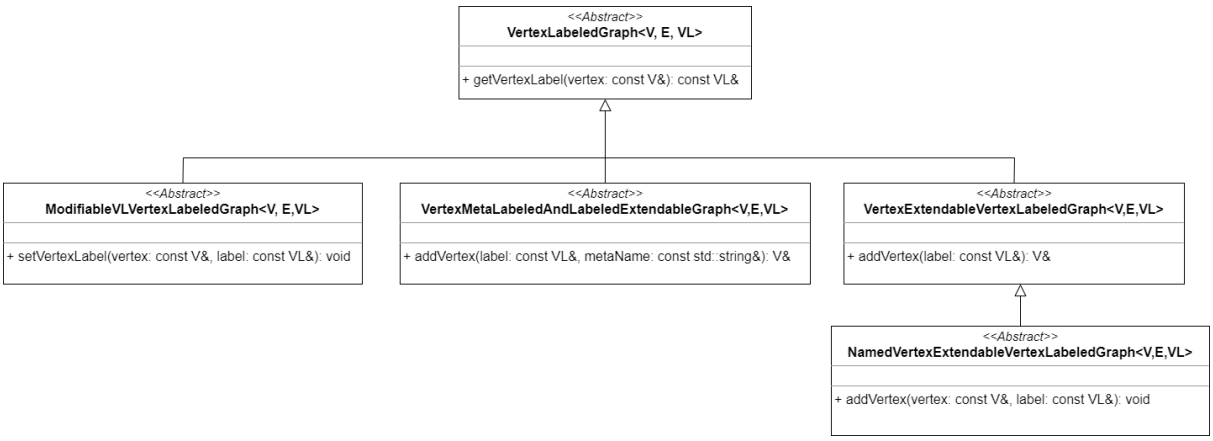

Figura 3.11: Diagramma UML delle classi VertexLabeledGraph, ModifiableVLVertexLabeledGraph, VertexMetaLabeledAndLabeledExtendableGraph, VertexExtendableVertexLabeledGraph, NamedVertexExtendableVertexLabeledGraph.

Con queste classi astratte si definisce il concetto di *grafo etichettato sui vertici*. Permettono di definire grafi i cui vertici avranno un'etichettatura associata. Ognuna di queste consentirà di specializzare ancora di più il concetto definito specificando caratteristiche aggiuntive.

La classe astratta *VertexLabeledGraph* estende la classe Graph, da solo modella il concetto di grafo etichettato sui vertici nella sua accezione più semplice: non estendibile, non riducibile, non orientato, non modificabile.
Possiede un solo metodo:

- *VL& getVertexLabel(const V& vertex)* consente di conoscere l'etichetta associata al vertice *vertex*.

La classe astratta *ModifiableVLVertexLabeledGraph* estende la classe *VertexLabeledGraph* andando a specializzare il concetto di grafo etichettato sui vertici rendendo tali etichette modificabili, ha un'ulteriore metodo oltre a quello ereditato dalla sua superclasse che permette questa operazione:

- *void setVertexLabel(const V& vertex, const VL& label)* consente di inserire l'etichetta *label* e di associarla al vertice *vertex*.

La classe astratta *VertexExtendableVertexLabeledGraph* estende la classe *VertexLabeledGraph*, rappresenta il concetto di *grafo etichettato sui vertici ed estendibile* in cui l'utilizzatore di una sua concretizzazione può non specificare il tipo di vertice da aggiungere, compito che è lasciato allo sviluppatore che avrà, così, piena libertà di implementazione. Possiede un solo metodo:

- *V& addVertex(const VL& label)* consente di aggiungere un vertice con etichetta *label* al grafo su cui viene effettuta la chiamata. Restituisce il vertice appena inserito.

La classe astratta *NamedVertexExtendableVertexLabeledGraph* estende la classe *VertexExtendableVertexLabeledGraph* e definisce lo stesso concetto. Per come è stata progettata, presenta un duplice vantaggio, da una parte l'implementatore potrebbe riuscire a concretizzarla lasciandola parametrica sui vertici e quindi permettendo all'utilizzatore di definire il tipo di dato degli oggetti che vorrà utilizzare come tali; dall'altro, invece, si dà la possibilità a colui che svilupperà la concretizzazione di poter usare come parametri il tipo di dato che più desidera per raggiungere l'efficienza voluta.
Possiede il seguente metodo:

- *void addVertex(const V& vertex, const VL& label)* consente di aggiungere al grafo su cui viene effettuata la chiamata il vertice *vertex* ed associarvi l'etichetta *label*.

La classe astratta *VertexMetaLabeledAndLabeledExtendableGraph* estende la classe *VertexLabeledGraph* e definisce il concetto di *grafo etichettato, meta-etichettato ed estendibile per i vertici*.
Possiede il seguente metodo:

- *V& addVertex(const VL& label, const std::string& metaName)* consente di aggiungere al grafo su cui viene effettuata la chiamata un vertice con etichetta *label* e meta-etichetta *metaName*. Restituisce il vertice appena inserito.

3.1.2.12 Le classi EdgeLabeledGraph ModifiableELEdgeLabeledGraph EdgeMetaLabeledAndLabeledExtendableGraph EdgeExtendableEdgeLabeledGraph EdgeExtendableNamedEdgeLabeledGraph

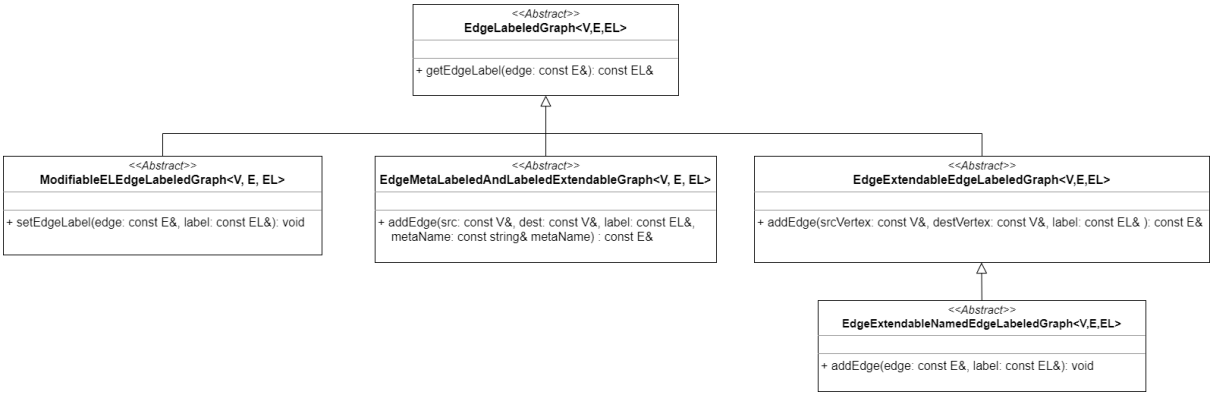

Figura 3.12: Diagramma UML delle classi EdgeLabeledGraph, ModifiableELEdgeLabeledGraph, EdgeMetaLabeledAndLabeledExtendableGraph, EdgeExtendableEdgeLabeledGraph, EdgeExtendableNamedEdgeLabeledGraph.

Con queste classi astratte si definisce il concetto di *grafo etichettato sugli archi*. Permettono di definire grafi i cui archi avranno un'etichettatura associata. Ognuna di queste consentirà di specializzare ancora di più il concetto definito specificando caratteristiche aggiuntive.

La classe astratta *EdgeLabeledGraph* estende la classe Graph, da sola modella il concetto di grafo etichettato sugli archi nella sua accezione più semplice: non estendibile, non riducibile, non orientato, non modificabile.
Possiede un solo metodo:

- *const EL& getEdgeLabel(const E& edge)* consente di conoscere l'etichetta associata all'arco *edge*.

La classe astratta *ModifiableELEdgeLabeledGraph* estende la classe *EdgeLabeledGraph* andando a specializzare il concetto di grafo etichettato sugli archi rendendo tali etichette modificabili, ha un'ulteriore metodo oltre a quello ereditato dalla sua superclasse che permette questa operazione:

- *void setEdgeLabel(const V& src, const V& dest, const EL& label)* consente di inserire l'etichetta *label* e di associarla all'arco che collega il vertice *src* al vertice *dest*.

La classe astratta *EdgeExtendableEdgeLabeledGraph* estende la classe *VertexLabeledGraph* caratterizzando il concetto di *grafo etichettato sugli archi ed estendibile* in cui l'utilizzatore di

una sua concretizzazione stabilisce solamente i due vertici tra cui ci dovrà essere il collegamento. Lo sviluppatore si occuperà di decidere in che modo memorizzarli. Possiede un solo metodo:

- *const E& addEdge(const V& src, const V& dest, const EL& label)* consente di aggiungere al grafo su cui viene effettuta la chiamata un arco che collega il vertice src al vertice *dest* con etichetta *label*. Restituisce l'arco appena inserito.

La classe astratta *EdgeExtendableNamedEdgeLabeledGraph* estende la classe *EdgeExtendableEdgeLabeledGraph* e definisce lo stesso concetto. Per come è stata progettata, presenta un duplice vantaggio, da una parte l'implementatore potrebbe riuscire a concretizzarla lasciandola parametrica sugli archi e quindi permettendo all'utilizzatore di definire il tipo di dato dell'oggetto che vorrà utilizzare come arco; dall'altro, invece, si dà la possibilità a colui che svilupperà la concretizzazione di poter usare come parametri il tipo di dato che più desidera per raggiungere l'efficienza voluta.
Possiede il seguente metodo:

- *void addEdge(const E& edge, const EL& label)* consente di aggiungere al grafo su cui viene effettuata la chiamata l'arco *edge* ed associarvi l'etichetta *label*.

La classe astratta *EdgeMetaLabeledAndLabeledExtendableGraph* estende la classe *EdgeLabeledGraph* e definisce il concetto di *grafo etichettato, meta-etichettato ed estendibile per gli archi.* Possiede il seguente metodo:

- *const E& addEdge(const V& src, const V& dest, const EL& label, const std::string& metaName)* consente di aggiungere al grafo su cui viene effettuata la chiamata un arco che collega il vertice *src* al vertice *dest* con etichetta *label* e meta-etichetta *metaName*. Restituisce l'arco appena inserito.

3.1.2.13 La classe Arena

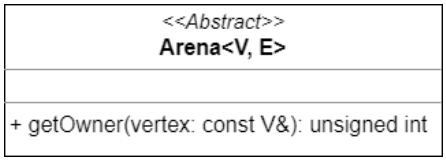

Figura 3.13: Diagramma UML della classe Arena.

La classe astratta *Arena* estende la classe Graph e modella il concetto più semplice di Arena: rappresenta una specializzazione di grafo in cui ogni vertice è associato ad un *owner* rappresentato da un intero senza segno, è non estendibile, non riducibile, non etichettata, non modificabile.
Possiede un solo metodo:

- *unsigned int getOwner(const V& vertex)* consente di conoscere l'*owner* del vertice *vertex* dell'arena su cui viene effettuata la chiamata.

3.1.2.14 La classe VertexExtendableArena

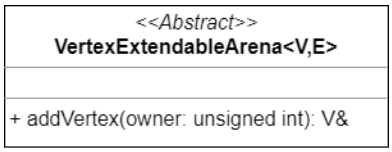

Figura 3.14: Diagramma UML della classe VertexExtendableArena.

La classe astratta *VertexExtendableArena* estende la classe *Arena* e modella il concetto di *arena estendibile per i vertici*, non estendibile per gli archi, non riducibile, non etichettata, non modificabile.
Possiede il seguente metodo:

- *V& addVertex(unsigned int owner)* consente di estendere l'arena su cui viene effettuata la chiamata inserendo un nuovo vertice ed associandolo ad un *owner*. Restituisce il vertice appena aggiunto.

3.1.2.15 La classe VertexMetaLabeledExtendableArena

La classe astratta *VertexMetaLabeledExtendableArena* estende la classe *Arena* che specializza andando a renderla *estendibile e con meta-etichettatura sui vertici*. Concretizzazioni di questa classe astratta permettono di definire arene non estendibili per gli archi, non riducibili, non orientate, non etichettate, non modificabili, in cui è possibile inserire dei vertici con meta-etichettatura.
Possiede il seguente metodo:

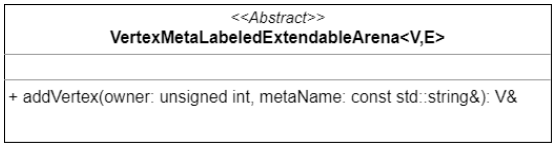

Figura 3.15: Diagramma UML della classe VertexMetaLabeledExtendableArena.

- *V& addVertex(unsigned int owner, const std::string& metaName)* consente di estendere l'arena su cui viene effettuata la chiamata inserendo un nuovo vertice, associandovi un *owner* ed un *metaName*. Restituisce il vertice appena aggiunto.

3.1.2.16 La classe VertexMetaLabeledAndLabeledExtendableArena

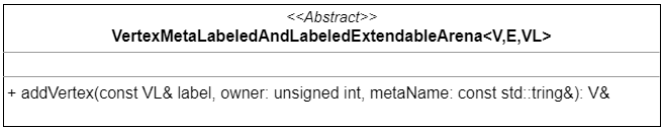

Figura 3.16: Diagramma UML della classe VertexMetaLabeledAndLabeledExtendableArena.

La classe astratta *VertexMetaLabeledAndLabeledExtendableArena* estende la classe *Arena* che specializza andando a renderla *estendibile, con etichettatura e meta-etichettatura sui vertici*. Concretizzazioni di questa classe astratta permettono di definire arene estendibili e con etichettatura solo per i vertici, non riducibili, non orientate, non modificabili.
Possiede il seguente metodo:

- *V& addVertex(const VL& label, unsigned int owner, const std::string& metaName)* consente di estendere l'arena su cui viene effettuata la chiamata inserendo un nuovo vertice, associandovi un *owner*, un *metaName* ed l'etichetta *label*. Restituisce il vertice appena aggiunto.

3.1.2.17 La classe OwnerModifiableArena

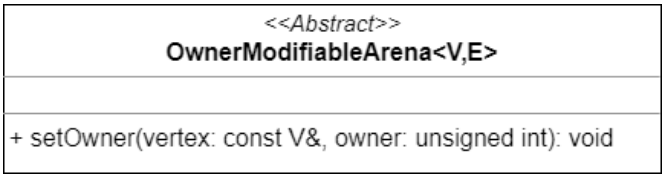

Figura 3.17: Diagramma UML della classe OwnerModifiableArena.

La classe astratta *OwnerModifiableArena* estende la classe *Arena*, modella il concetto di *arena con owner modificabili*. Concretizzazioni di questa classe astratta permettono di definire arene non estendibili, non etichettate, non riducibili, non orientate, ma modificabili, cioè con

la possibilità di cambiare l'owner associato ad un vertice.
Possiede il seguente metodo:

- *void setOwner(const V& vertex, unsigned int owner)* consente di cambiare l'*owner* di un vertice dell'arena su cui viene effettuata la chiamata.

3.1.2.18 La classe MutableDirectedGraph

La classe astratta *MutableDirectedGraph* modella il concetto di *grafo non etichettato, estendibile, riducibile, orientato e modificabile.* E' una cosiddetta *classe unione* che di per se non ha metodi ma raggruppa quelli delle classi che va ad estendere. E' parametrica sia sui vertici che sugli archi. La sua utilità, come quella di tutte le altre classi unioni che vedremo di seguito, è di fornire un'interfaccia completa per i tipi di strutture maggiormente usati.
Tale classe estende: *VertexMetaLabeledExtendableGraph, EdgeMetaLabeledExtendableGraph, VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableVertexMetaLabeledGraph, ModifiableEdgeMetaLabeledGraph* ereditando da queste tutti i loro metodi.

3.1.2.19 La classe MutableFullLabeledDirectedGraph

La classe astratta *MutableFullLabeledDirectedGraph* modella il concetto di *grafo etichettato sia su vertici che su archi, estendibile, riducibile, orientato e modificabile.* E' una *classe unione* ed è parametrica sui vertici, sugli archi e sulla loro etichettatura.
Tale classe estende ereditando da queste tutti i loro metodi: *VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableVLVertexLabeledGraph, VertexMetaLabeledAndLabeledExtendableGraph, ModifiableELEdgeLabeledGraph, EdgeMetaLabeledAndLabeledExtendableGraph, ModifiableVertexMetaLabeledGraph, ModifiableEdgeMetaLabeledGraph.*

3.1.2.20 La classe MutableNodeLabeledDirectedGraph

La classe astratta *MutableNodeLabeledDirectedGraph* modella il concetto di *grafo etichettato solo sui vertici, estendibile, riducibile, orientato e modificabile.* E' una *classe unione* ed è parametrica sui vertici, sugli archi e sull'etichettatura dei vertici.
Tale classe estende ereditando da queste tutti i loro metodi: *ModifiableVertexMetaLabeledGraph, EdgeReducibleGraph, VertexReducibleGraph, VertexMetaLabeledAndLabeledExtendableGraph, ModifiableVLVertexLabeledGraph, EdgeMetaLabeledExtendableGraph, InDirectedGraph, OutDirectedGraph, ModifiableEdgeMetaLabeledGraph.*

3.1.2.21 La classe MutableEdgeLabeledDirectedGraph

La classe astratta *MutableEdgeLabeledDirectedGraph* modella il concetto di *grafo etichettato solo sugli archi, estendibile, riducibile, orientato e modificabile.* E' una *classe unione* ed è

parametrica sui vertici, sugli archi e sull'etichettatura di quest'ultimi.
Tale classe estende ereditando da queste tutti i loro metodi: *ModifiableVertexMetaLabeledGraph, VertexMetaLabeledExtendableGraph, VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableELEdgeLabeledGraph, EdgeMetaLabeledAndLabeledExtendableGraph, ModifiableEdgeMetaLabeledGraph.*

3.1.2.22 La classe MutableDirectedArena

La classe astratta *MutableDirectedArena* modella il concetto di *arena non etichettata, estendibile, riducibile, orientata e modificabile.* E' una *classe unione* ed è parametrica sulle posizioni (o vertici) e sulle mosse (o archi).
Tale classe estende ereditando da queste tutti i loro metodi: *ModifiableVertexMetaLabeledGraph, EdgeMetaLabeledExtendableGraph, VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, VertexMetaLabeledExtendableArena, OwnerModifiableArena, ModifiableEdgeMetaLabeledGraph.*

3.1.2.23 La classe MutablePositionLabeledDirectedArena

La classe astratta *MutablePositionLabeledDirectedArena* modella il concetto di *arena etichettata solo sulle posizioni, estendibile, riducibile, orientata e modificabile.* E' una *classe unione* ed è parametrica sulle posizioni (o vertici), sull'etichettatura di queste e sulle mosse (o archi).
Tale classe estende ereditando da queste tutti i loro metodi: *ModifiableVertexMetaLabeledGraph, EdgeMetaLabeledExtendableGraph, VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableVLVertexLabeledGraph, VertexMetaLabeledAndLabeledExtendableArena, OwnerModifiableArena, ModifiableEdgeMetaLabeledGraph.*

3.1.2.24 La classe MutableMoveLabeledDirectedArena

La classe astratta *MutableMoveLabeledDirectedArena* modella il concetto di *arena etichettata solo sulle mosse, estendibile, riducibile, orientata e modificabile.* E' una *classe unione* ed è parametrica sulle posizioni (o vertici), sulle mosse (o archi) e sull'etichettatura di queste.
Tale classe estende ereditando da queste tutti i loro metodi: *ModifiableVertexMetaLabeledGraph, VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableELEdgeLabeledGraph, EdgeMetaLabeledAndLabeledExtendableGraph, VertexMetaLabeledExtendableGraph, OwnerModifiableArena, ModifiableEdgeMetaLabeledGraph.*

3.1.2.25 La classe MutablePositionTargetLabeledDirectedGame

La classe astratta *MutablePositionTargetLabeledDirectedGame* modella il concetto di *gioco avente posizioni target*, ovvero giochi in cui delle posizioni vengono marcate e prese in considerazione

ai fini della vittoria. Rappresenta giochi estendibili, riducibili, modificabili, orientati, con etichettatura di tipo *bool* per indicare le posizioni target.
E' una *classe unione* ed estende, ereditando da queste tutti i loro metodi: *VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableVLVertexLabeledGraph, VertexMetaLabeledAndLabeledExtendableArena, OwnerModifiableArena, EdgeMetaLabeledExtendableGraph, ModifiableVertexMetaLabeledGraph, ModifiableEdgeMetaLabeledGraph*

3.1.2.26 La classe MutableMoveTargetLabeledDirectedGame

La classe astratta *MutableMoveTargetLabeledDirectedGame* modella il concetto di *gioco avente mosse target*, ovvero giochi in cui delle mosse vengono marcate e prese in considerazione ai fini della vittoria. Rappresenta giochi estendibili, riducibili, modificabili, orientati, con etichettatura di tipo *bool* per indicare le mosse target.
E' una *classe unione* ed estende, ereditando da queste tutti i loro metodi: *VertexMetaLabeledExtendableArena, VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableELEdgeLabeledGraph, EdgeMetaLabeledAndLabeledExtendableGraph, OwnerModifiableArena, ModifiableVertexMetaLabeledGraph, ModifiableEdgeMetaLabeledGraph.*

3.1.2.27 La classe MutablePositionPriorityLabeledDirectedGame

La classe astratta *MutablePositionPriorityLabeledDirectedGame* modella il concetto di *gioco avente priorità sulle posizioni*, ovvero giochi in cui le posizioni possiedono dei numeri naturali associati, a cui si dà il nome di *priorità*, che verranno utilizzate per il calcolo del vincitore. Rappresenta giochi estendibili, riducibili, modificabili, orientati, con etichettatura di tipo *unsigned int* per le priorità sulle posizioni.
E' una *classe unione* ed estende, ereditando da queste tutti i loro metodi: *VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableVLVertexLabeledGraph, VertexMetaLabeledAndLabeledExtendableArena, OwnerModifiableArena, EdgeMetaLabeledExtendableGraph, ModifiableVertexMetaLabeledGraph, ModifiableEdgeMetaLabeledGraph.*

3.1.2.28 La classe MutableMovePriorityLabeledDirectedGame

La classe astratta *MutableMovePriorityLabeledDirectedGame* modella il concetto di *gioco avente priorità sulle mosse*, ovvero giochi in cui le mosse possiedono dei numeri naturali associati, a cui si dà il nome di *priorità*, che verranno utilizzate per il calcolo del vincitore. Rappresenta giochi estendibili, riducibili, modificabili, orientati, con etichettatura di tipo *unsigned int* per le priorità sulle mosse.
E' una *classe unione* ed estende, ereditando da queste tutti i loro metodi: *ModifiableVertexMetaLabeledGraph, VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirec-*

tedGraph, ModifiableELEdgeLabeledGraph, EdgeMetaLabeledAndLabeledExtendableGraph, VertexMetaLabeledExtendableGraph, OwnerModifiableArena, ModifiableEdgeMetaLabeledGraph.

3.1.2.29 La classe MutablePositionPayoffLabeledDirectedGame

La classe astratta *MutablePositionPayoffLabeledDirectedGame* modella il concetto di *gioco avente payoff sulle posizioni*, ovvero giochi in cui le posizioni possiedono dei numeri decimali associati, a cui si dà il nome di *payoff*, che verranno utilizzati per il calcolo del vincitore. Rappresenta giochi estendibili, riducibili, modificabili, orientati, con etichettatura di tipo *double* per i payoff sulle posizioni.
E' una *classe unione* ed estende, ereditando da queste tutti i loro metodi: *VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, ModifiableVLVertexLabeledGraph, VertexMetaLabeledAndLabeledExtendableArena, OwnerModifiableArena, EdgeMetaLabeledExtendableGraph, ModifiableVertexMetaLabeledGraph, ModifiableEdgeMetaLabeledGraph.*

3.1.2.30 La classe MutableMovePayoffLabeledDirectedGame

La classe astratta *MutableMovePayoffLabeledDirectedGame* modella il concetto di *gioco avente payoff sulle mosse*, ovvero giochi in cui le mosse possiedono dei numeri decimali associati, a cui si dà il nome di *payoff*, che verranno utilizzati per il calcolo del vincitore. Rappresenta giochi estendibili, riducibili, modificabili, orientati, con etichettatura di tipo *double* per i payoff sulle mosse.
E' una *classe unione* ed estende, ereditando da queste tutti i loro metodi: *VertexReducibleGraph, EdgeReducibleGraph, InDirectedGraph, OutDirectedGraph, VertexMetaLabeledExtendableArena, OwnerModifiableArena, ModifiableELEdgeLabeledGraph, EdgeMetaLabeledAndLabeledExtendableGraph, ModifiableVertexMetaLabeledGraph, ModifiableEdgeMetaLabeledGraph.*

3.1.2.31 Le classi PositionTargetLabeledReachabilityGame PositionTargetLabeledSafetyGame PositionTargetLabeledBuchiGame PositionTargetLabeledCoBuchiGame

Tutte queste classi sono astratte ed estendono la classe *MutablePositionTargetLabeledDirectedGame*, rappresentano, quindi, dei giochi in cui la winning condition è specificata da un'insieme di posizioni target. Queste classi sono state definite, per ora, senza alcun metodo poichè non se ne è avuta la necessità, ma sono state progettate per poter contenere al loro interno metodi che differiscono in base al tipo di gioco che modellano.
I nomi risultano essere esplicativi sul tipo di gioco che rappresentano.

3.1.2.32 Le classi MoveTargetLabeledReachabilityGame MoveTargetLabeledSafetyGame MoveTargetLabeledBuchiGame MoveTargetLabeledCoBuchiGame

Tutte queste classi sono astratte ed estendono la classe *MutableMoveTargetLabeledDirectedGame*, rappresentano, quindi, dei giochi in cui la winning condition è specificata da un'insieme di mosse target. Come le precedenti, anche queste classi sono state definite, per ora, senza alcun metodo poichè non se ne è avuta la necessità, ma sono state progettate per poter contenere al loro interno metodi che differiscono in base al tipo di gioco che modellano.
Anche qui i nomi risultano essere esplicativi sul tipo di gioco che rappresentano.

3.1.2.33 La classe PositionPriorityLabeledParityGame

Tale classe astratta estende la classe *MutablePositionPriorityLabeledDirectedGame* vista in precedenza e modella il concetto di *gioco di parità* in cui le priorità sono specificate sulle posizioni.

3.1.2.34 La classe MovePriorityLabeledParityGame

Tale classe astratta estende la classe *MutableMovePriorityLabeledDirectedGame* vista in precedenza e modella il concetto di *gioco di parità* in cui le priorità sono specificate sulle mosse.

3.1.2.35 La classe PositionLabeledMeanPayoffGame

La classe astratta *PositionLabeledMeanPayoffGame* estende la classe *MutablePositionPayoffLabeledDirectedGame* vista in precedenza e modella il concetto di *Mean-Payoff Game*, in cui il payoff è specificato sulle posizioni.

3.1.2.36 La classe MoveLabeledMeanPayoffGame

La classe astratta *MoveLabeledMeanPayoffGame* estende la classe *MutableMovePayoffLabeledDirectedGame* vista in precedenza e modella il concetto di *Mean-Payoff Game*, in cui il payoff è specificato sulle mosse.

3.1.2.37 Le classi DiscountedPayoffGame e DiscountModifiableDiscountedPayoffGame

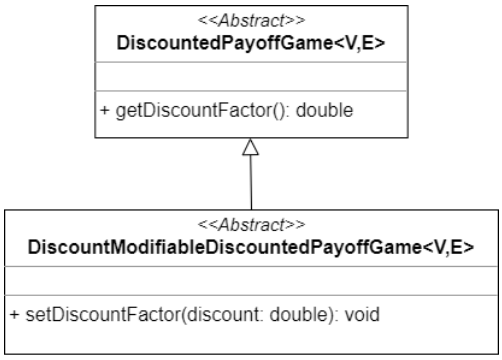

Figura 3.18: Diagramma UML delle classi DiscountedPayoffGame e DiscountModifiableDiscountedPayoffGame.

Entrambe le classi definiscono il concetto di *Discounted Payoff Game* nella sua accezione più semplice: non estendibile, non riducibile, non modificabile in nessun modo, infatti di per sè non presentano alcun metodo che permettano queste operazioni nè ne ereditano alcuno.

La classe *DiscountedPayoffGame* estende la classe *Arena* e possiede il seguente metodo:

- *double getDiscountFactor()* consente di conoscere il fattore di sconto del Discounted-Payoff Game su cui viene effettuata la chiamata.

La classe *DiscountModifiableDiscountedPayoffGame* estende la classe *DiscountedPayoffGame* andando a rendere il fattore di sconto modificabile, possiede il seguente metodo:

- *void setDiscountFactor(double discount)* consente di impostare a *discount* il fattore di sconto del Discounted-Payoff Game su cui viene effettuata la chiamata.

3.1.2.38 La classe PositionLabeledDiscountedPayoffGame

La classe *PositionLabeledDiscountedPayoffGame* va ad estendere le classi *MutablePositionPayoffLabeledDirectedGame* e *DiscountModifiableDiscountedPayoffGame* andando quindi ad unire i concetti definiti in queste due classi. Rappresenta un *Discounted-Payoff Game* in cui i payoff sono specificati sulle posizioni ed il fattore di sconto è modificabile.

3.1.2.39 La classe MoveLabeledDiscountedPayoffGame

La classe *MoveLabeledDiscountedPayoffGame* va ad estendere le classi *MutableMovePayoffLabeledDirectedGame* e *DiscountModifiableDiscountedPayoffGame* unendo i concetti definiti in queste due classi. Rappresenta un *Discounted-Payoff Game* in cui i payoff sono specificati sulle mosse ed il fattore di sconto è modificabile.

3.2 Implementazione

In questa sezione verranno discusse le concretizzazioni della parte astratta della libreria, verranno presentate e studiate le classi che effettivamente si occupano del funzionamento del tool più a basso livello, si vedrà in che modo determinati comportamenti sono stati ottenuti.

3.2.1 Il CLI Lyra

Il CLI utilizzato per poter interagire col tool tramite riga di comando è LYRA. Fornisce un parser della riga di comando facile da usare per la maggior parte dei casi d'uso con la minima scrittura di codice. Non mira a fornire tutte le funzionalità per tutti gli utenti ma si adatta al meglio per le funzionalità del *GamesOnGraphTool*.

3.2.1.1 Analisi dei comandi

I comandi di cui si può usufruire con il GamesOnGraphTool, in questa versione, sono cinque:

- *create* che permette la creazione di nuovi grafi
- *graph2arena* che permette l'estensione del grafo ad arena.
- *arena2game* che permette l'estensione dell'arena ad un determinato tipo di gioco.
- *game2arena* che permette di estrarre l'arena da un determinato gioco.
- *arena2graph* che permette di estrarre il grafo da una determinata arena.

Per ognuno di questi comandi è stata realizzata una classe a sè stante con una ben precisa nomenclatura: nome del comando seguito dalla keyword *Command*.
Le classi progettate per questi sono rispettivamente le seguenti: *CreateCommand, GraphToArenaCommand, ArenaToGameCommand, GameToArenaCommand, ArenaToGraphCommand* contenute all'interno della directory del tool al seguente percorso: *prg/cli/*.
Ognuna di queste classi ha due elementi principali imprescindibili: un costruttore che prende come parametro di ingresso un riferimento all'oggetto che rappresenta il CLI ed un metodo privato chiamato *doCommand* che verrà chiamato quando il parser del CLI considera corretto il comando specificato e che contiene al suo interno tutte le operazioni che serviranno per la sua esecuzione.

3.2.1.2 Integrare nuovi comandi

Se si è in possesso del codice sorgente del tool e si vuole integrare un nuovo comando, i passaggi per farlo sono pochi e semplici: come visto in precedenza basta creare, all'interno della directory del tool, al percoso *prg/cli/*, una classe con il nome di *NomeComandoCommand* dove

NomeComando è il nome del comando che si vuole creare e Command, invece, è la keyword che indica che la classe in questione rappresenta un comando per il CLI. Una volta creata, al suo interno dovranno essere definiti i due elementi imprescindibili esposti nella sezione precedente: un costruttore pubblico fatto in questo modo:

```
NomeComandoCommand(lyra::cli& cli);
```

ed il seguente metodo privato:

```
void doCommand(const lyra::group& group);
```

La classe dovrà essere contenuta all'interno del namespace *gogtl::prg::cli*. In conclusione si dovrà avere un file header il cui nome sarà *NomeComandoCommand.hpp* ed un file cpp il cui nome sarà *NomeComandoCommand.cpp* che conterrà le definizioni del costruttore e del metodo doCommand.
All'interno del costruttore dovrà essere presente la seguente riga di codice:

```
cli.add_argument(lyra::group([this](const lyra::group& group)
    { this->doCommand(group); }
)
```

si potranno chiamare poi sull'oggetto restituito tutti i metodi descritti nella documentazione [2] per poter formare il comando che si vuole implementare.
All'interno del metodo *doCommand* invece si andrà a definire tutte le operazioni necessarie, tutto ciò che effettivamente si vuole far eseguire al comando che si sta aggiungendo al tool. Tale metodo, infatti, verrà chiamato dopo che il CLI avrà validato l'input dell'utente.

3.2.2 Le classi concrete

Tutte le classi concrete relative alle tre strutture (grafi, arene, giochi) sono state realizzate come implementazioni della parte astratta della libreria del GamesOnGraphTool e utilizzando come base i vari tipi di grafo definiti nella *libreria LEDA*. Queste classi, dunque, non faranno altro che estendere in modo protetto una particolare concretizzazione di grafo della LEDA per far si che le classi contenute in quest'ultima possano interfacciarsi con la parte astratta della libreria del GamesOnGraphTool.
Il percorso all'interno della directory del tool in cui si trovano è: *lib/concrete/*.

3.2.2.1 I grafi

Le classi concrete relative ai grafi sono *LedaGraph, FullLabeledLedaGraph, NodeLabeledLedaGraph, EdgeLabeledLedaGraph*.

LedaGraph è realizzata andando ad implementare la classe astratta *MutableDirectedGraph<leda::node_struct, leda::edge_struct>* (concretizza quindi il concetto

da essa definito) e estendendo in modo protetto *leda::graph*. Siccome *leda::graph* non permette di inserire meta-etichettatura sono state definite all'interno di *LedaGraph* due attributi che garantissero questa funzionalità: *std::unordered_map<leda::node_struct*, std::string> nodeMetaNames* e *std::unordered_map<leda::edge_struct*, std::string> edgeMetaNames*. Queste due strutture consentiranno di associare e ottenere la meta-etichettatura relativa ai nodi e agli archi in tempo costante.

FullLabeledLedaGraph, NodeLabeledLedaGraph, EdgeLabeledLedaGraph sono state costruite seguendo lo stesso principio definito sopra, implementano rispettivamente le classi astratte: *MutableFullLabeledDirectedGraph<leda::node_struct,leda::edge_struct,VL,EL>*, *MutableNodeLabeledDirectedGraph<leda::node_struct,leda::edge_struct,VL>*, *MutableEdgeLabeledDirectedGraph<leda::node_struct,leda::edge_struct,EL>* ed estendendo in modo protetto le seguenti classi *leda::GRAPH<VL,EL>*, *leda::GRAPH<VL,int>*, *leda::GRAPH<int,EL>*. I parametri VL ed EL sono i parametri formali della varie classi concrete. Siccome la libreria LEDA non mette a disposizione grafi parzialmente etichettati, e cioè etichettati o su vertici o su archi, si è utilizzato come base il grafo etichettato su entrambi, andando ad usare un parametro attuale "fake" laddove non era necessario il suo utilizzo.

3.2.2.2 Le arene

Le classi concrete relative alle arene sono *DirectedArena, FullLabeledDirectedArena, PositionLabeledDirectedArena, MoveLabeledDirectedArena*. Queste implementano rispettivamente *MutableDirectedArena<leda::node_struct, leda::edge_struct>*, *MutableFullLabeledDirectedArena<leda::node_struct,leda::edge_struct,VL,EL>*, *MutablePositionLabeledDirectedArena<leda::node_struct,leda::edge_struct,VL>*, *MutableMoveLabeledDirectedArena<leda::node_struct,leda::edge_struct,EL>* ed estendono in modo protetto rispettivamente *leda::graph*, *leda::GRAPH<VL,EL>*, *leda::GRAPH<VL,int>*, *leda::GRAPH<int,EL>*. A differenza dei grafi, possiedono un'informazione in più relativa alle posizioni: gli *owner*. La memorizzazione di questo attributo è stata accorpata a quella della meta-etichettatura nella struttura *std::unordered_map<leda::node_struct*, std::pair<unsigned int,std::string»* dove alla posizione viene associata sia l'intero relativo all'owner, sia la stringa relativa alla meta-etichetta.

3.2.2.3 I giochi

Le classi relative ai vari giochi implementano le classi astratte che definscono i vari concetti di gioco ed estendono le arene viste nella sezione precedente andando a specificarne i parametri formali.

Ad esempio: la classe *LEDAPositionPriorityLabeledParityGame* implementa *PositionPriorityLabeledParityGame<leda::node_struct,leda::edge_struct>* ed estende la classe concreta *Po-*

sitionLabeledDirectedArena<unsigned int>. Estendendo quest'ultima sarà possibile, infatti, associare alle posizioni sia *gli owner* che le priorità tipiche dei Parity Game.

3.2.3 Funzione di generazione random

Il *GamesOnGraphTool* possiede un comando che permette di generare in modo random dei grafi orientati, non etichettati, a questo scopo è stata realizzata all'interno della libreria del tool al percorso *lib/concrete/io* la classe *DirectedGraphCreator* con all'interno il seguente metodo:

```
void create(
    gogtl::lib::abstract::MutableDirectedGraph<V,E>& graph,
    unsigned int numNodes,
    unsigned int minDeg,
    unsigned int maxDeg
);
```

Tale metodo prende in ingresso un riferimento ad un *MutableDirectedGraph*, il numero di nodi che possiederà il grafo generato, il numero minimo ed il numero massimo di archi uscenti da un nodo (ogni nodo, dunque, avrà un numero di adiacenti compreso tra *minDeg* e *maxDeg*).
Con il riferimento al *MutableDirectedGraph* si ha un duplice vantaggio: uno è relativo al passaggio per riferimento e non per valore rendendo il metodo più efficiente ed uno è relativo al fatto che MutableDirectedGraph è una classe astratta (con metodi virtuali puri) quindi non si è legati ad una particolare implementazione e ciò favorisce il disaccoppiamento.
Analizziamo come è stato realizzato il metodo:

Per prima cosa è stato definito un modo per associare un identificativo di tipo intero ad ogni vertice creato

```
std::unordered_map<int,V*> map;
```

Per fare ciò è stata utilizzata un *unorderded_map* della *Standard Library*, poichè per questa struttura le operazioni da usare, inserimento e ottenimento dei valori data la key, hanno una complessità in termini di tempo di $O(1)$.

La struttura *map* viene inizializzata nel seguente modo:

```
for(int i = 0; i<numNodes; ++i){
    metaName = std::to_string(i);
    std::pair<int, V*> pair(i,&graph.addVertex(metaName));
    map.insert(pair);
}
```

Segue la definizione del generatore di numeri casuali in questo modo:

```
std::random_device rd;
```

```
std::mt19937 gen(rd());
std::uniform_int_distribution<> deg_distr(minDeg, maxDeg);
```

Tutto ciò che segue è definito all'interno di un for che va da $i = 0$ a $i < numNodes$, ogni valore di i rappresenterà l'id di un vertice contenuto all'interno di *map*.
È stato definito un array di *nodi target* di dimensione *numNodes* ovvero un vettore contente al suo interno interi che vanno da 0 a $numNodes - 1$, rappresenta il pool da cui verranno pescati gli adiacenti di un nodo:

```
int *tgt = new int[numNodes];

for(int j = 0; j<numNodes; j++){
  tgt[j] = j;
}
```

Segue la definizione della variabile *degree* che conterrà di volta in volta, per ogni nodo contenuto all'interno di *map* il suo grado uscente generato col generatore di numeri casuali definito in precedenza.

```
int degree = deg_distr(gen);
```

Infine si avrà l'aggiunta degli archi, realizzata dal seguente spezzone di codice:

```
for(int j = 0; j<degree; ++j){
  std::uniform_int_distribution<> target_distr(0, n-1);
  int x = target_distr(gen);
  metaName = std::to_string(i);
  metaName += "->";
  metaName += std::to_string(tgt[x]);
  graph.addEdge(*map[i], *map[tgt[x]], metaName);
  tgt[x] = tgt[n-1];
  n--;
}
```

Analizziamo il codice sopracitato che verrà eseguito un numero di volte pari al valore di *degree*:

```
std::uniform_int_distribution<> target_distr(0, n-1);
```

permette di definire l'oggetto che si occuperà della generazione casuale di numeri che vanno da 0 a $n - 1$ (n è un altro nome dato al parametro *numNodes*).

```
int x = target_distr(gen);
```

In x sarà contenuto il numero casuale generato e rappresenterà una posizione all'interno dell'array *tgt*.

```
metaName = std::to_string(i);
metaName += "->";
metaName += std::to_string(tgt[x]);
```

In *metaName* sarà contenuta la meta etichettatura che sarà associata all'arco creato.

```
graph.addEdge(*map[i], *map[tgt[x]], metaName);
```

Qui si ha la creazione dell'arco vero e proprio, andando ad utilizzare come vertice sorgente il vertice che in *map* è associato all'identificativo i (il cui valore è dato dall'iterazione corrente del for esterno) e come vertice destinazione quello che in *map* è associato all'identificativo che si trova in posizione x all'interno dell'array *tgt*.

```
tgt[x] = tgt[n-1];
n--;
```

A questo punto in posizione x ci sarà l'id di un vertice che è stato già aggiunto come adiacente del nodo corrente e siccome non sono ammessi archi multipli tra stesse coppie di vertici non si fa altro che spostare in questa posizione l'ultimo valore dell'array *tgt* che si troverà in posizione $n-1$ e si decrementa n in modo tale che la nuova lunghezza dell'array non prenderà in considerazione l'ultima posizione (che conterrà un valore già presente). Così facendo, all'interno del vettore ci saranno sempre e solo identificativi diversi e non ancora utilizzati.

3.2.4 I parser

I parser sono uno degli strumenti fondamentali di cui si fa largo uso all'interno del tool in quanto permettono di leggere dallo *stdin* oppure da un file le specifiche di un grafo, di un'arena o di un gioco e validarle secondo le regole definite all'interno della sezione 2.4 di questo documento (*"Specifica dei formati"*). In caso di validazione, ciò che è stato analizzato verrà tramutato in un oggetto della classe opportuna che sarà messa a disposizione del programmatore.

Tutti i parser sono definiti all'interno della directory del tool al percorso *lib/concrete/io* ed estendono la classe astratta *Parser* che ha la seguente definizione:

```
namespace gogtl::lib::abstract {
/* ************************************************** */
template <typename Data>
class Parser{
  protected:
    void skipWhitespace(std::istream& istream);
    void toUppercase(std::string& str);
  public:
    virtual void parse(Data& data, std::istream& istream) = 0;
    virtual void parse(Data& data, std::string filePath);
};
/* ************************************************** */
}
```

I metodi *skipWhitespace* e *toUppercase* sono dei metodi di utilità, dichiarati protetti in quanto dovranno essere usati solo dalle classi che estenderanno questa. Il primo permette di saltare gli spazi fino al prossimo carattere dello stream dato in input, il secondo permette di trasformare in maiuscolo la stringa passata come parametro.
Il metodo *parse* che prende in input un *std::istream* è virtuale puro ed è quello che le sottoclassi dovranno implementare per poter effettuare il parsing vero e proprio. Il metodo *parse* che prende in input un percorso sottoforma di stringa, invece, ha già un'implementazione molto semplice che non fa altro che aprire lo stream che opera sul file specificato passandolo in input all'altro metodo *parse*.

3.2.4.1 Parser per Parity Game

Analizziamo come è fatta la funzione che ci permette di effettuare il parsing di un *parity game* con priorità sulle posizioni e di creare un *PositionPriorityLabeledParityGame*.
Tale parser permetterà di leggere e validare parity game nel formato stabilito da PGsolver [3].
La firma del metodo è la seguente:

```
void parse(
gogtl::lib::abstract::PositionPriorityLabeledParityGame<V,E>& game,
std::istream& istream)
```

Di seguito i passaggi che sono stati effettuati all'interno:

```
this->skipWhitespace(istream);
if(!(istream>>ch)){
    throw std::runtime_error("Empty file.");
}
```

Si saltano tutti gli spazi bianchi fin quando non si trova il primo carattere, in modo tale da rendere il parser flessibile e funzionante anche se lo stream ha degli spazi bianchi (lasciati inconsciamente) all'inizio; se si incontra la fine dello stream la funzione lancia un'eccezione di runtime con descrizione "Empty file".

```
do{
  if(ch == ' ' || ch == '\n' || ch == '\t' || ch == '\r' ||
  ch == ';'){
    this->toUppercase(buff);
    if(buff != "PARITY"){
      buff.clear();
    }
    if(ch == ';'){
      istream.putback(ch);
    }
    break;
  }else{
    buff += ch;
  }
}while(istream>>ch);
this->skipWhitespace(istream);
```

Il parsing della *header line* è effettuato da questo spezzone di codice. Fintantoché c'è un carattere da leggere nello stream, si prende tale carattere e, se non è un qualsiasi tipo di spazio vuoto, lo si concatena alla stringa *buff*. Appena si arriva ad uno spazio vuoto significa che avremo letto la parte della header line che ci permette di capire di fronte a che tipo di gioco ci troviamo, prendiamo la stringa *buff* e la trasformiamo in maiuscolo in modo tale da rendere il parser più flessibile in quanto potrà accettare sia stringhe scritte in maiuscolo o minuscolo sia con un miscuglio delle due. Siccome questo è il parser che permette la lettura di un parity game e siccome la header line secondo le specifiche del formato è opzionale, se *buff* conterrà

una stringa diversa da "PARITY" allora è possibile che si sia letta la parte di specifica di una posizione, svuotiamo tale stringa e se l'ultimo carattere letto è ";" lo si rimette nello stream. A questo punto si esce dal ciclo e si saltano tutti gli spazi bianchi fino al prossimo carattere.

```
if(!buff.empty()){
  if(!(istream>>ch)){
    throw std::runtime_error("Missing position number.");
  }
  buff.clear();
  if(ch != ';'){
    do{
      if(ch == ' ' || ch == '\n' || ch == '\t' || ch == '\r' ||
      ch == ';'){
        try{
          id = std::stoi(buff);
        }catch(std::invalid_argument& e){
          throw std::runtime_error("Invalid number of positions
          specification.");
        }catch(std::out_of_range& e){
          throw std::runtime_error("Number of positions out of range
          of representable values.");
        }
        buff.clear();
        if(ch == ';'){
          istream.putback(ch);
        }
        break;
      }else{
        buff += ch;
      }
    }while(istream>>ch);
    this->skipWhitespace(istream);
    if(!(istream>>ch) || ch != ';'){
      throw std::runtime_error("Missing ; at the end of the line.");
    }
  }else{
      throw std::runtime_error("Missing position number.");
  }
```

```
}else{
    istream.seekg(0,istream.beg);
}
```

Se la stringa *buff* non è vuota allora significa che avremmo letto dallo *stream* "Parity" ed effettivamente ci troviamo di fronte ad un parity game, in caso contrario si suppone che si stia leggendo le specifiche di un parity game senza header line dunque ci si riposiziona all'inizio dello stream.

Se non ci sono più caratteri viene lanciata un'eccezione poichè il parser si aspetta di trovare il numero delle posizioni contenute all'interno del gioco, stessa cosa succede anche se in *ch* è presente il carattere ";".

Nel caso in cui in *ch* sia presente qualcosa di diverso da ";" ci si aspetta che sia il momento di leggere il numero delle posizioni, tutto ciò che è racchiuso all'interno del ciclo *do...while* ci permette di fare ciò. In *buff* dovrà essere presente una stringa numerica in caso contrario verrà sollevata un'eccezione, lo stesso succede se il numero eccede il range dei valori rappresentabili da un intero.

```
nodeAlreadyAdded = false;
do{
    if(ch == ' ' || ch == '\n' || ch == '\t' || ch == '\r'){
        try{
            id = std::stoi(buff);
        }catch(std::invalid_argument& e){
            throw std::runtime_error("Invalid identifier specification.");
        }catch(std::out_of_range& e){
            throw std::runtime_error("Identifier out of range of
            representable values.");
        }
        buff.clear();
        break;
    }else{
        buff += ch;
    }
}while(istream>>ch);
this->skipWhitespace(istream);
```

Da qui in poi inizia il parsing della specifica delle posizioni. Tutto ciò si trova racchiuso in un *while(istream»ch)* e, dunque, verrà eseguito fin quando all'interno dello stream rimangono caratteri da leggere.

Qui non vi è nulla di diverso da ciò che è stato visto finora. Un ciclo *do...while* permette di

costruire la stringa *buff* finchè non ci si trova davanti ad un qualsiasi spazio bianco, a quel punto si termina la concatenazione. Siccome ci si aspetta di trovare, all'interno della stringa, l' identificativo di una posizione, si effettua su di essa la chiamata alla funzione *std::stoi* che lancerà eccezioni se viene chiamata su una stringa che non contiene un valore numerico, oppure se tale numero eccede il range dei valori interi rappresentabili.
L'unica particolarità è rappresentata dal booleano *nodeAlreadyAdded* utile poichè ci permetterà di non aggiungere più di una volta la stessa posizione.

```
if(!(istream>>ch)){
  throw std::runtime_error("Missing priority specification.");
}
do{
  if(ch == ' ' || ch == '\n' || ch == '\t' || ch == '\r'){
    try {
      priorityLabel = std::stoi(buff);
    }catch(std::invalid_argument& e){
      throw std::runtime_error("Invalid priority specification.");
    }catch(std::out_of_range& e){
      throw std::runtime_error("Priority out of range of
      representable values.");
    }
    buff.clear();
    break;
  }else{
    buff += ch;
  }
}while(istream>>ch);
this->skipWhitespace(istream);
```

Dopo il parsing dell'identificativo di una posizione ci si aspetta di trovare la *priorità* ad essa associata, nel caso in cui si sia raggiunta la fine dello stream, infatti, verrà sollevata un'eccezione che informa che tale priorità è mancante.

```
if(!(istream>>ch)){
  throw std::runtime_error("Missing owner.");
}
do{
  if(ch == ' ' || ch == '\n' || ch == '\t' || ch == '\r'){
    try{
      owner = std::stoi(buff);
```

```
        }catch(std::invalid_argument& e){
          throw std::runtime_error("Invalid owner specification.");
        }catch(std::out_of_range& e){
          throw std::runtime_error("Owner out of range of
          representable values.");
        }
        if(owner < 0 || owner > 1){
          throw std::runtime_error("Invalid owner.");
        }
        buff.clear();
        break;
      }else{
        buff += ch;
      }
    }while(istream>>ch);
    this->skipWhitespace(istream);
```

Se nello stream non sono più presenti caratteri allora il parsing termina lanciando l'eccezione, notificando che non è presente la definizione dell'*owner* come previsto dalla specifica del formato per i parity game. Altrimenti si procede con l'analisi.

```
    if(specifiedPositions.find(id) == specifiedPositions.end()){
      specifiedPositions[id] =
      std::make_pair(true,
        &game.addVertex(
          priorityLabel,owner, std::to_string(id)
        )
      );
      positionCount++;
    }else{
      if(specifiedPositions[id].first == true){
        nodeAlreadyAdded = true;
      }else{
        specifiedPositions[id].first = true;
        game.setVertexLabel(*(specifiedPositions[id].second),
        priorityLabel);
        game.setOwner(*(specifiedPositions[id].second), owner);
        game.setVertexMetaName(*(specifiedPositions[id].second),
        std::to_string(id));
```

```
    }
  }
```

Il parser funziona in questo modo: ogni volta che si trova davanti ad un identificativo (sia questo relativo ad una posizione o quello di un adiacente) lo aggiunge come nuova posizione all'interno del gioco. Come conseguenza a ciò potrebbe accadere che questa venga aggiunta più volte, basti pensare ad una situazione in cui la prima volta lo si incontra come identificativo e la seconda volta come adiacente. Per ovviare a questo problema si è usato il seguente oggetto *std::unordered_map<int,std::pair<bool,V*»* *specifiedPositions* che con il primo parametro permette di dare un identificativo di tipo intero alla posizione che si aggiunge al gioco, con il secondo parametro si associa all'intero un puntatore alla posizione appena inserita ed un booleano. Questo assumerà il valore *true* se come secondo elemento della coppia vi è il puntatore ad una posizione reale, false se invece vi sarà il puntatore di una posizione la cui specifica non è stata ancora trovata all'interno dello stream. A questo punto si possono verificare due scenari: o la sua definizione arriverà dopo ed in questo caso il parser nelle prossime iterazioni vi aggiungerà gli adiacenti, o non arriverà mai e si procederà con la sua eliminazione.
Il codice sopracitato si comporta in questo modo: se in *specifiedPositions* non è presente l'identificativo *id* allora ci si trova di fronte ad una vera e propria posizione non ancora inserita all'interno del gioco, la si inserisce, e la si associa ad *id* insieme al booleano *true*. In caso contrario vuol dire che era già presente e bisogna distinguere tra due possibilità: il booleano associato è true, significa che è stato già aggiunto al gioco e non deve subire alcuna modifica; il booleano associato è false, lo si fa diventare true e gli si modificano la priorità, l'owner e il metaname, poichè vuol dire che, quando è stato scoperto come adiacente, era stato aggiunto con parametri "fake" (in tale situazione non era ancora possibile conoscere quei dati).

```
while(true){
  do{
    if(ch == ' ' || ch == '\n' || ch == '\t' || ch == '\r' ||
      ch == ',' || ch == ';' || ch == '\"'){
      try{
        adjacentID = std::stoi(buff);
      }catch(std::invalid_argument& e){
        throw std::runtime_error("Invalid adjacent specification.");
      }catch(std::out_of_range& e){
        throw std::runtime_error("Reward out of range of
      representable values.");
      }
      if(ch == ',' || ch == ';' || ch == '\"'){
        istream.putback(ch);
```

```
        }
        if(!nodeAlreadyAdded){
          moveMetaName += std::to_string(id);
          moveMetaName += "->";
          moveMetaName += std::to_string(adjacentID);
          if(specifiedPositions.find(adjacentID) == specifiedPositions.end()){
            positionCount++;
            pair = std::make_pair(false,&game.addVertex(0,0,"fake"));
            specifiedPositions[adjacentID] = pair;
          }
        }
        buff.clear();
        break;
      }else{
        buff += ch;
      }
    }while(istream>>ch);
    this->skipWhitespace(istream);
    if(!(istream>>ch) && ch != ';'){
      throw std::runtime_error("Missing ; at the end of the line.");
    }
    if(ch == ';' || ch == '\"'){
      break;
    }
    if(ch == ','){
      this->skipWhitespace(istream);
      if(!(istream>>ch)){
        throw std::runtime_error("Missing ; at the end of the line.");
      }
        continue;
    }
  }
```

Qui si ha il parsing degli adiacenti. Una volta ottenuto in *adjacentID* il valore, e verificato che tale valore, quando convertito ad intero, non sollevi eccezioni, si va a controllare se la posizione di cui è adiacente è stata già specificata, in caso negativo allora si procede. Si verifica se tale posizione che rappresenta l'adiacente è stata già aggiunta al gioco, se non è stato fatto la si aggiunge e la si inserisce nella struttura *specifiedPositions* associandovi il booleano false, che ci

indicherà che dovrà ancora essere trovata, all'interno dello stream, la specifica di quel nodo. L'arco tra *id* e *adjacentID* viene aggiunto indifferentemente se adjacentID è una posizione reale o meno (cioè la cui specifica la si troverà prima o poi).
Seguono poi dei controlli per vedere se ci sono ulteriori adiacenti o si è raggiunta la fine della riga di specifica di una posizione.

```
if(ch == ';'){
  this->skipWhitespace(istream);
  continue;
}
if(ch == '\"'){
  while(true){
  if(!(istream>>ch)){
    throw std::runtime_error("Missing \" at the end
    of the optional label.");
  }
  if(ch == '\"'){
    break;
  }
    buff+=ch;
  }
  positionMetaName = buff;
  buff.clear();
}
```

Qui viene effettuato il parsing della meta-etichettatura opzionale.

```
if(!positionMetaName.empty()){
  if(!nodeAlreadyAdded){
    game.setVertexMetaName(*(specifiedPositions[id].second),
      positionMetaName);
  }
  positionMetaName.clear();
}
this->skipWhitespace(istream);
if(!(istream>>ch)){
  throw std::runtime_error("Missing ; at the end of the line.");
}
if(ch == ';'){
  this->skipWhitespace(istream);
```

```
    continue;
  }
```

Si inserisce la meta-etichettatura alla posizione associata all'identificativo *id* all'interno di *specifiedPositions* e si effettuano controlli sulla presenza del ";" che indica la fine di una specifica di una posizione.

```
  for(auto& val : specifiedPositions){
    if(val.second.first == false){
      game.removeVertex(*(val.second.second));
      positionCount--;
    }
  }
```

Si controlla se tutte le posizioni incontrate come adiacenti sono state specificate. Se nella struttura *specifiedPositions* hanno come valore booleano *false* vuol dire che la lora specifica all'interno dello stream su cui è stato effettuato il parsing è mancante e dunque bisogna eliminarle dal gioco.

```
  if(positionNumber != -1){
    if(positionNumber < positionCount){
      throw std::runtime_error("Invalid positionNumber.");
    }
  }
```

Per terminare, si controlla se il numero delle posizioni specificato nella header line è minore di quelle di cui si è effettuato il parsing, in questo caso si lancia un'eccezione.

3.2.5 I printer

Fondamentali quanto i parser sono i *printer* che permettono di stampare su console o salvare in un file di testo grafi, arene e giochi secondo le regole delle specifiche dei rispettivi formati.

3.2.5.1 Printer per Parity Game

Analizziamo come è fatto il printer per i parity game nel formato specificato da PGSolver [3]. Tale printer è legato fortemente alla classe *LEDAPositionPriorityLabeledParityGame*, dunque a differenza del parser che lavorava su una classe astratta, qui si va a lavorare su una classe concreta il che non lo rende riutilizzabile per altre implementazioni dei giochi di parità. Il motivo per cui si è dovuti fare questa scelta è legato al funzionamento della libreria LEDA che non si interfacciava al meglio con la parte astratta della libreria del GamesOnGraphTool per cui si sono dovuti usare dei metodi protetti della struttura sottostante, a tal proposito il printer

in questione è una classe interna a *LEDAPositionPriorityLabeledParityGame*.
Per effettuare il printing del parity game è stata adottata una variante della BFS per i grafi, ciò ha permesso di avere dei tempi di esecuzione ottimali, poichè la complessità di tempo è pari ad un $O(V + E)$.
La firma della funzione è la seguente:

```
void print(const LEDAPositionPriorityLabeledParityGame& game,
    std::ostream& ostream)
```

Vediamo nel dettaglio il codice:

```
ostream<<"parity "<<game.number_of_nodes()<<";"<<std::endl;
```

Questo spezzone di codice ci permette di stampare la header line. Si stampa la stringa "parity" seguita dal numero delle posizioni del gioco e da ";", dopodichè si termina la riga.

```
leda::list<leda::node> nodeList = game.all_nodes();
for(int i = 0; i < nodeList.length(); ++i){
    item = nodeList.get_item(i);
    node = nodeList.contents(item);
    colorsAndIDs[node] = std::make_pair(false, -1);
}
```

Si ottiene in *nodeList* una lista delle posizioni contenute nel gioco e si inizializza la *std::unorderd_map colorsAndIDs* andando ad associare ad ogni nodo contenuto in *nodeList* un booleano che indica se quella posizione è stata già inserita nella coda (secondo le regole della BFS) ed un intero che rappresenterà il suo id all'interno dello stream su cui verrà effettuato il printing.

Tutto ciò che segue sarà inserito all'interno di un ciclo che va da $j = 0$ a $j < nodeList.length()$

```
item = nodeList.get_item(j);
node = nodeList.contents(item);
if(!(colorsAndIDs[node].first)){
    queue.push(node);
    colorsAndIDs[node] = std::make_pair(true, id);
    id++;
}
```

Si prende un *node* da *nodeList*, si verifica se è stato già visitato o meno andando a leggere il valore del booleano ad esso associato in *colorsAndIDs*, se questo è *false*, lo si inserisce nella coda, si porta il booleano a *true* e gli si associa un *id*.

Tutto ciò che segue sarà inserito all'interno di un *while(!queue.empty())* che andrà ad ogni iterazione a concatenare informazioni alla stringa *buff*.

```
node = queue.front();
```

```
buff += std::to_string(colorsAndIDs[node].second);
buff += " ";
buff += std::to_string(game.inf(node));
buff += " ";
buff += std::to_string(game.getOwner(*node));
```

Viene preso il primo nodo della coda, in *buff* vengono concatenati rispettivamente il suo id, l'etichetta associata e l'owner.

```
outGoingEdges = game.out_edges(node);
for(int i = 0 ; i < outGoingEdges.length(); ++i){
  buff += " ";
  item = outGoingEdges.get_item(i);
  edge = outGoingEdges.contents(item);
  node1 = leda::target(edge);

  if(!(colorsAndIDs[node1].first)){
    queue.push(node1);
    colorsAndIDs[node1] = std::make_pair(true, id);
    buff += std::to_string(id);
    id++;
  }else{
    buff += std::to_string(colorsAndIDs[node1].second);
  }
  buff +=",";
}
```

Si ottengono in *outGoingEdges* tutti gli adiacenti a *node*. Ogni adiacente viene messo in *node1*, se il booleano ad esso associato, in *colorsAndIDs*, è false lo si mette in coda e vi si associa un id, tale id viene concatenato alla stringa *buff* ed incrementato (in modo tale che i successivi saranno tutti diversi da questo). Nel caso in cui il booleano fosse true, significa che si sta cercando di visitare un nodo già visto, a cui sarà già stato associato un identificativo, si prende quest'ultimo dalla struttura *colorsAndIDs* e lo si concatena a *buff*.

```
if(outGoingEdges.length() > 0){
  buff.pop_back();
}
positionMetaName = game.getVertexMetaName(*node);
if(!positionMetaName.empty()){
  buff += " \"";
```

```
    buff += positionMetaName;
    buff += "\"";
  }
  buff += ";\n";
  ostream<<buff;
  buff.clear();
  queue.pop();
```

Terminato l'inserimento degli adiacenti, ci potrebbe essere in *buff* un "," che viene cancellato grazie alla chiamata della funzione *pop_back*. Segue la concatenazione della meta-etichettatura, dopodichè tutto ciò che è in *buff* viene mandato in output sullo stream, si pulisce la stringa, si elimina l'elemento corrente dalla coda e si ritorna a valutare la condizione del *while(!queue.empty())*.

Capitolo 4

Valutazioni

Sommario

4.1 Testing **66**
4.2 Analisi statistica **67**
4.3 Conclusioni **86**

4.1 Testing

Al fine di testare il funzionamento dei vari comandi del GamesOnGraphTool è stato stabilito un *Test Plan* da seguire.
Tale piano prevedeva i seguenti punti:

1. *Creazione* di 500 giochi di parità con 500 posizioni l'uno e 2 mosse per ogni posizione.
2. *Creazione* di 500 giochi di parità con 500 posizioni l'uno e un numero di mosse random compreso tra 2 e 10 per ogni posizione.
3. *Estrazione* delle arene dai giochi ottenuti al punto 1.
4. *Estrazione* delle arene dai giochi ottenuti al punto 2.
5. *Estrazione* dei grafi dalle arene ottenute al punto 3.
6. *Estrazione* dei grafi dalle arene ottenute al punto 4.
7. *Creazione* di 25 giochi di parità random a partire da ogni arena ottenuta al punto 3.
8. *Creazione* di 25 giochi di parità random a partire da ogni arena ottenuta al punto 4.
9. *Creazione* di 25 giochi di parità random a partire da ogni grafo ottenuto al punto 5.
10. *Creazione* di 25 giochi di parità random a partire da ogni grafo ottenuto al punto 6.
11. *Creazione* di 500 giochi di parità con 500 posizioni l'uno e 2 mosse per ogni posizione con il tool di OINK.
12. *Creazione* di 500 giochi di parità con 500 posizioni l'uno e un numero di mosse random compreso tra 2 e 10 per ogni posizione con il tool di OINK.
13. Ripetere sui giochi ottenuti ai punti 11 e 12, utilizzando i comandi di estrazione e creazione di GamesOnGraphTool, le operazioni effettuate dai punti 3 a 10.

L'intero test plan è stato eseguito grazie a degli script bash scritti appositamente e contenuti nella directory del tool *script*, in modo tale che si possono effettuare le dovute verifiche di correttezza.
Ognuno di questi punti è stato eseguito con successo e senza alcun tipo di errore.
Non c'è stato bisogno di dover scrivere funzioni fatte con lo scopo preciso di testare i vari comandi del GamesOnGraphTool, a tal fine sono stati sufficienti le funzioni *parser* e le funzioni *printer* le quali possono essere viste l'una il modo per testare l'altra. Ai parser, infatti, sarà dato in input l'output derivante dai printer e se non viene sollevato alcun tipo di eccezione significa che quest'ultimo ha funzionato correttamente. Al contrario ai printer, per verificarne il funzionamento, può essere dato in input l'output dei parser.

4.2 Analisi statistica

I giochi ottenuti in seguito alle operazioni di estrazione e creazione specificate all'interno del *Test Plan* sono serviti per poter effettuare un'analisi statistica abbastanza dettagliata.
In primo luogo, è stato utilizzato il tool di OINK per il calcolo dei tempi di risoluzione utilizzando prima l'algoritmo *ZLK* e poi l'algoritmo *NPP* sui giochi ottenuti ai punti 1, 2, 7, 8, 9, 10, 11, 12, 13 del Test Plan. Al fine di poter minimizzare l'errore sui tempi, appiattendo i picchi dovuti a switch di contesto e/o altre operazioni della macchina adoperata, per ogni gioco è stato calcolato 10 volte il tempo di risoluzione ed è stata presa in considerazione la media aritmetica dei valori ottenuti. Questi dati sono stati raccolti nei seguenti istogrammi.

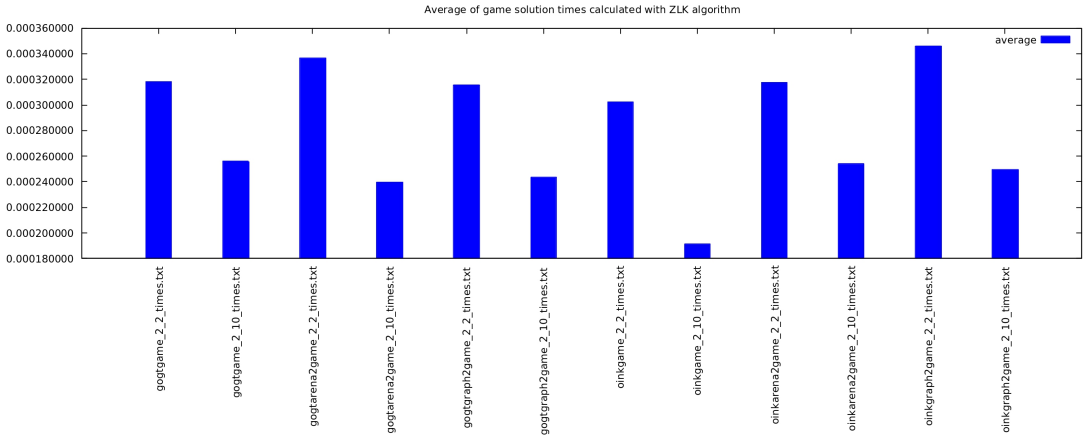

Figura 4.1: Istogramma della media dei tempi di risoluzione dei giochi calcolati con l'algoritmo ZLK

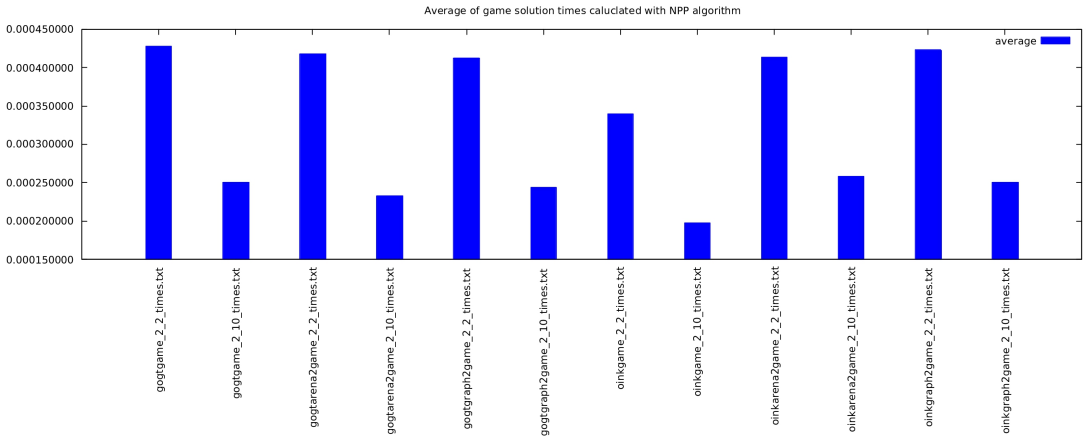

Figura 4.2: Istogramma della media dei tempi di risoluzione dei giochi calcolati con l'algoritmo NPP

Da questi grafici risulta, in modo abbastanza evidente, dando quindi conferma ad uno studio effettuato dal Professore Mogavero e dal Professore Benerecetti dell'Università di Napoli Federico II, che i giochi aventi 2 mosse hanno in ogni caso dei tempi di risoluzione media molto più alti rispetto ai giochi con un numero maggiore di mosse (in questo caso queste variano da 2 a 10).

Mettendo a confronto i due grafici si è ottenuto il seguente istogramma.

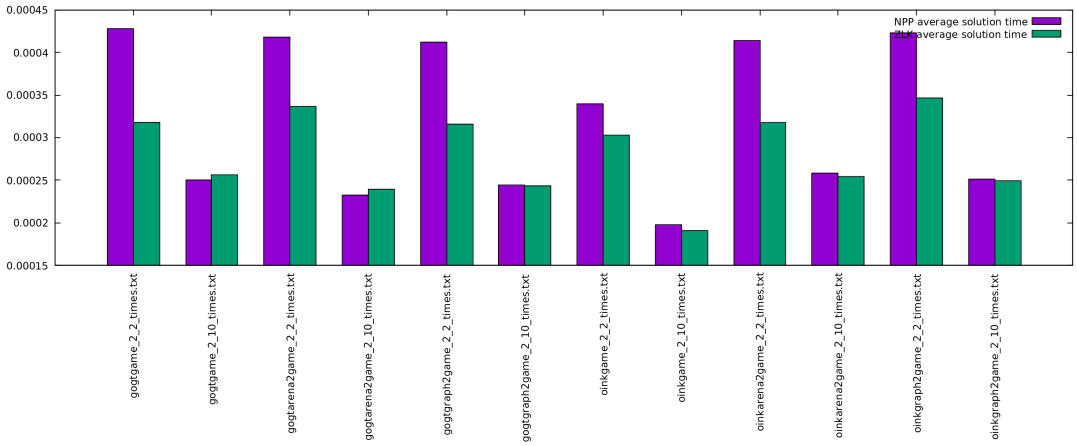

Figura 4.3: Istogramma comparativo della media dei tempi di risoluzione dei giochi calcolati con gli algoritmi NPP e ZLK

L'osservazione che viene fuori studiando questo grafico è che l'algoritmo NPP, nel calcolo dei tempi di risoluzione dei giochi, sembra essere più lento dell'algoritmo ZLK ma solamente sui giochi aventi 2 mosse per posizione, per quelli invece aventi da 2 a 10 mosse sembrano

comportarsi in modo molto simile. Si può notare, inoltre, come i giochi creati tramite il GamesOnGraphTool risultino essere più complessi di quelli creati interamente con il tool OINK. Se prendiamo in considerazione l'algoritmo NPP, i tempi di risoluzione dei primi risultano essere circa 1.3 volte maggiori dei secondi. Questo vale sia con quelli aventi 2 mosse che con quelli aventi da 2 a 10 mosse per posizione. Per i secondi invece la differenza, pur essendoci ed in favore dei giochi creati col GamesOnGraphTool, non è così evidente.

Un'ulteriore considerazione che è possibile fare studiando questo grafico è che la generazione della priorità tramite il GamesOnGraphTool mostra un caso più aleatorio rispetto a quella ottenuta tramite il tool di OINK. A testimoniare ciò è il fatto che, se prendiamo in esame il confronto tra i giochi generati tramite quest'ultimo tool e quelli ottenuti prendendo questi, estrapolandone l'arena e generando il nuovo gioco (quindi aggiungendovi le priorità) mediante il GamesOnGraphTool, i tempi di risoluzione dei primi appaiono più alti.

Dall'istogramma delle frequenze dei tempi calcolati con l'algoritmo NPP, si è notato un andamento della probabilità molto simile a quello di una *distribuzione lognormale*, talvolta tale distribuzione si adattava quasi perfettamente per i giochi con 2 mosse per posizione generati col GamesOnGraphTool, come possiamo vedere dai seguenti grafici.

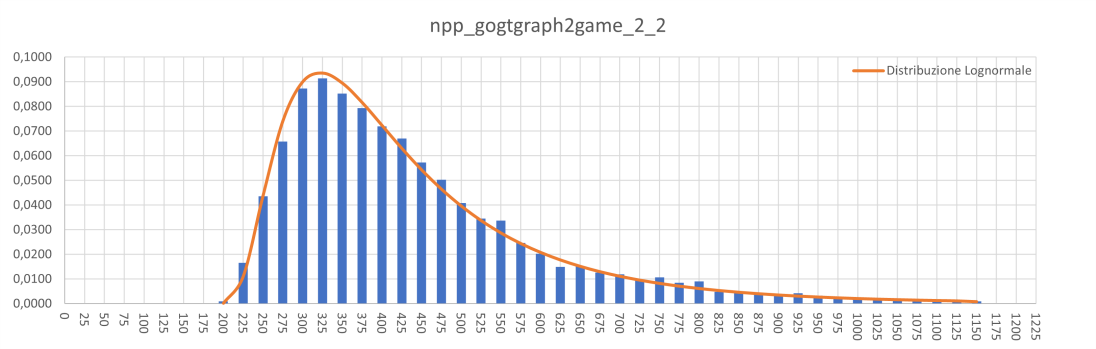

Figura 4.4: Istogramma delle frequenze relativo ai giochi con 2 mosse per ogni posizione ottenuti a partire dal grafo estrapolato dai giochi generati col GamesOnGraphTool.

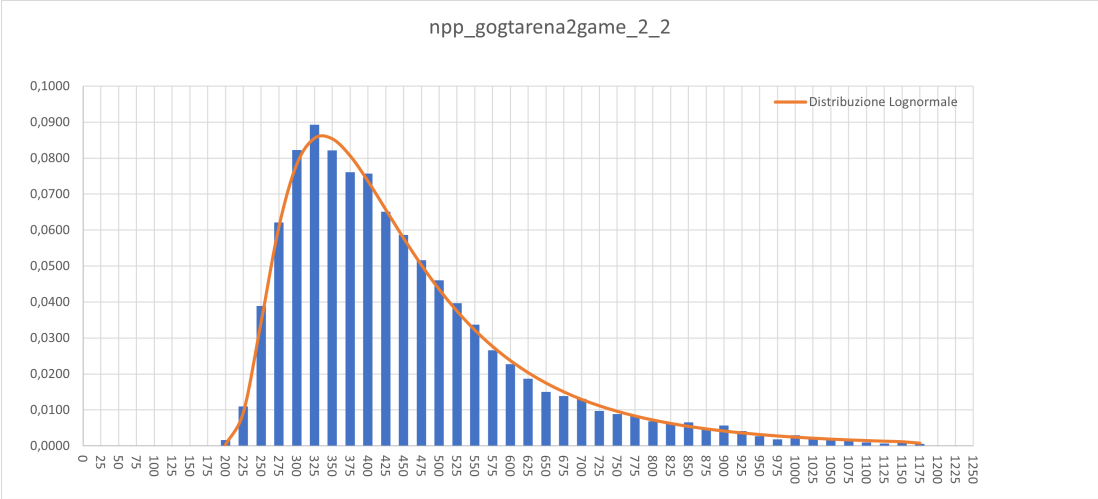

Figura 4.5: Istogramma delle frequenze relativo ai giochi con 2 mosse per ogni posizione ottenuti a partire dall'arena estrapolato dai giochi generati col GamesOnGraphTool.

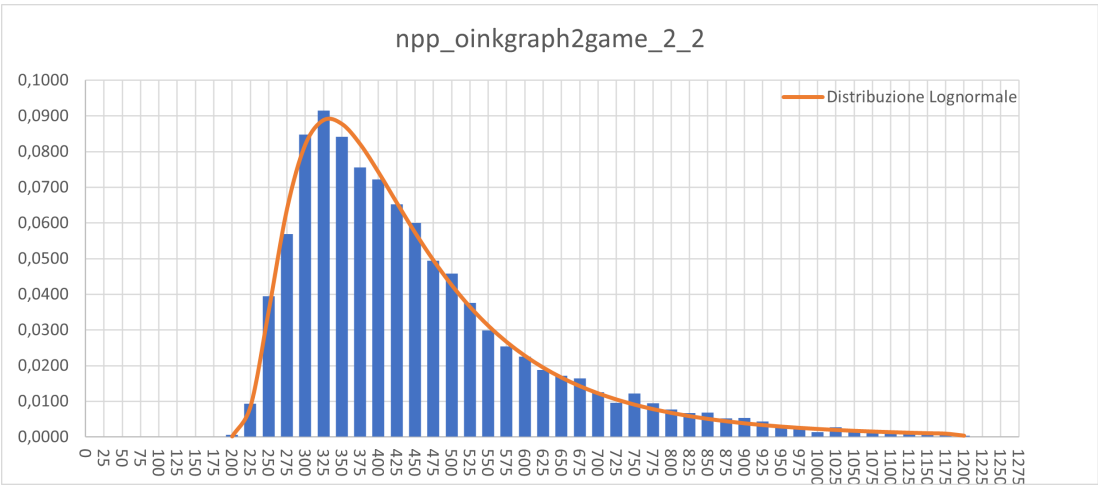

Figura 4.6: Istogramma delle frequenze relativo ai giochi con 2 mosse per ogni posizione ottenuti utilizzando il comando di creazione del GamesOnGraphTool a partire dal grafo estrapolato dai giochi generati col tool di OINK.

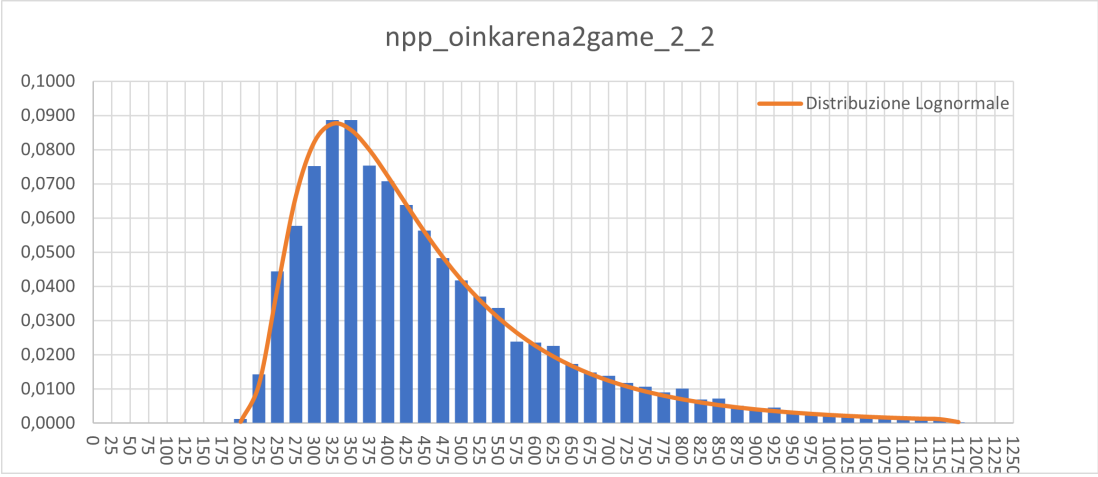

Figura 4.7: Istogramma delle frequenze relativo ai giochi con 2 mosse per ogni posizione ottenuti utilizzando il comando di creazione del GamesOnGraphTool a partire dall' arena estrapolata dai giochi generati col tool di OINK.

Di seguito invece, vediamo l'istogramma delle frequenze con relativa distribuzione lognormale dei tempi calcolati mediante utilizzo dell'algoritmo NPP per i giochi con 2 mosse creati interamente col tool di OINK. Anche qui la distribuzione che meglio approssima è quella lognormale, anche se non perfettamente, probabilmente dato dal fatto che il campione utilizzato è costituito da soli 500 giochi.

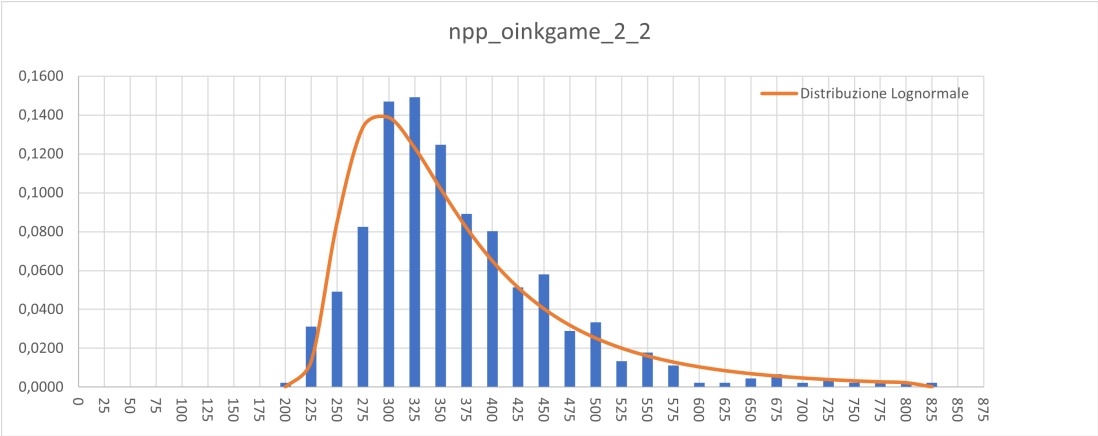

Figura 4.8: Istogramma delle frequenze relativo ai giochi con 2 mosse per ogni posizione ottenuti interamente tramite il tool di OINK.

Per quanto riguarda, invece, i giochi aventi da 2 a 10 mosse per posizione, la distribuzione notevole che più approssima l'andamento della probabilità in base ai tempi è ancora quella lognormale ma l'approssimazione risulta essere più grossolana poichè il numero di cluster è

minore, conseguenza del fatto che i giochi con più mosse sono più facili da risolvere e dunque il range dei tempi si restringe e si raggruppano in meno cluster.

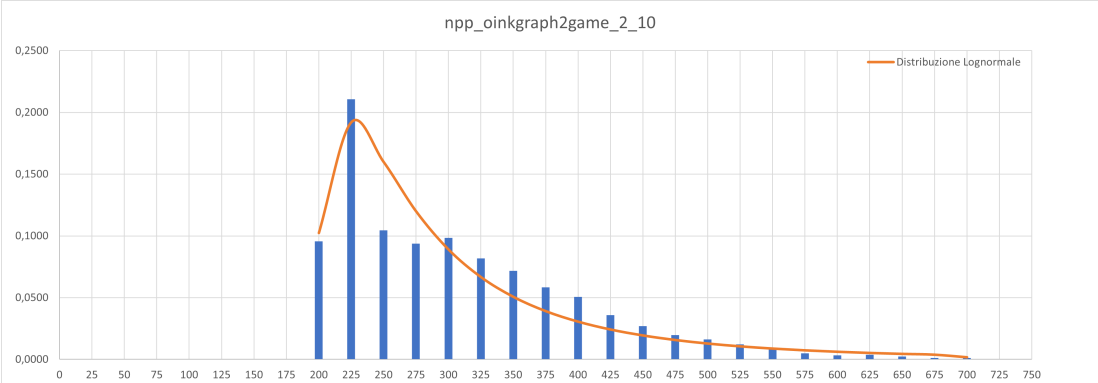

Figura 4.9: Istogramma delle frequenze relativo ai giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti utilizzando il comando di creazione del GamesOnGraphTool a partire dal grafo estrapolato dai giochi generati col tool di OINK.

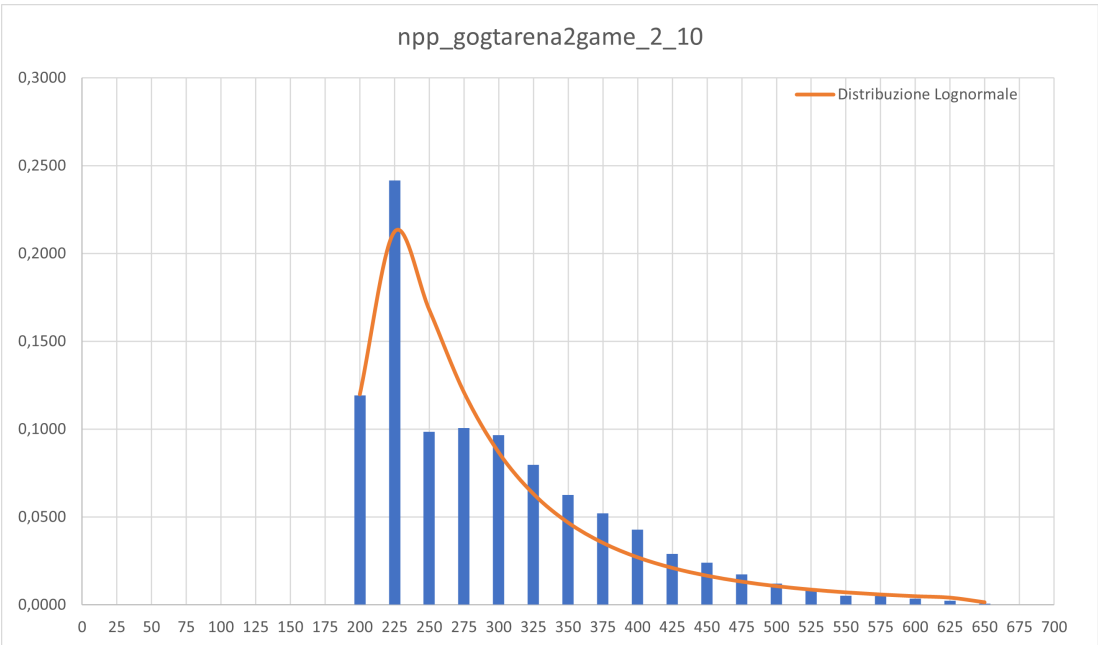

Figura 4.10: Istogramma delle frequenze relativo ai giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti a partire dall'arena estrapolato dai giochi generati col GamesOnGraphTool.

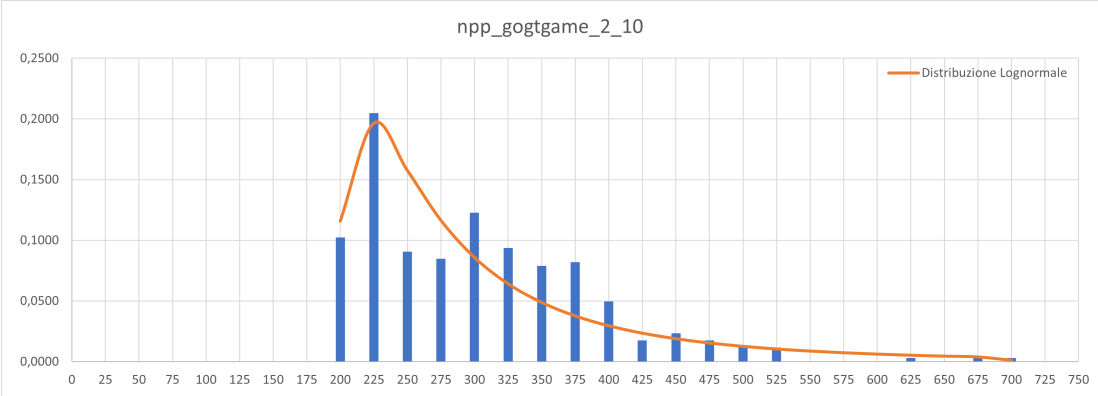

Figura 4.11: Istogramma delle frequenze relativo ai giochi con un numero di mosse per posizione che va da 2 a 10 generati col GamesOnGraphTool

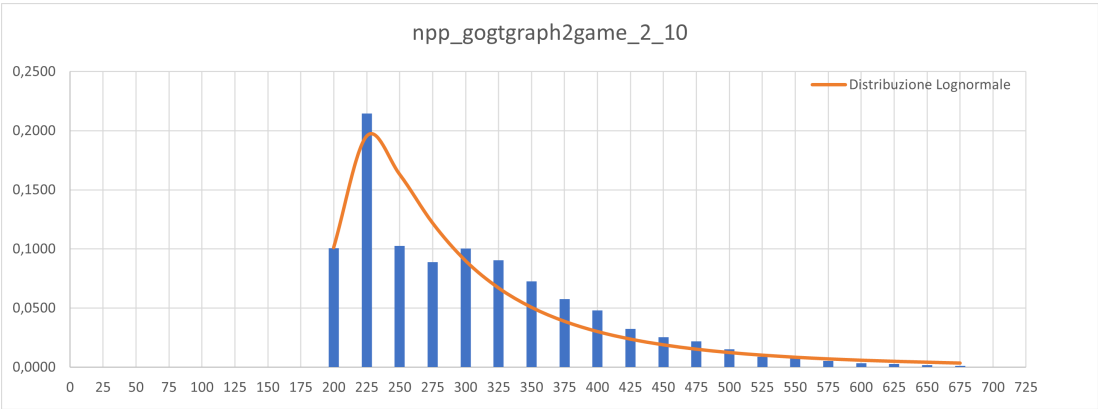

Figura 4.12: Istogramma delle frequenze relativo ai giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti a partire dal grafo estrapolato dai giochi generati col GamesOnGraphTool.

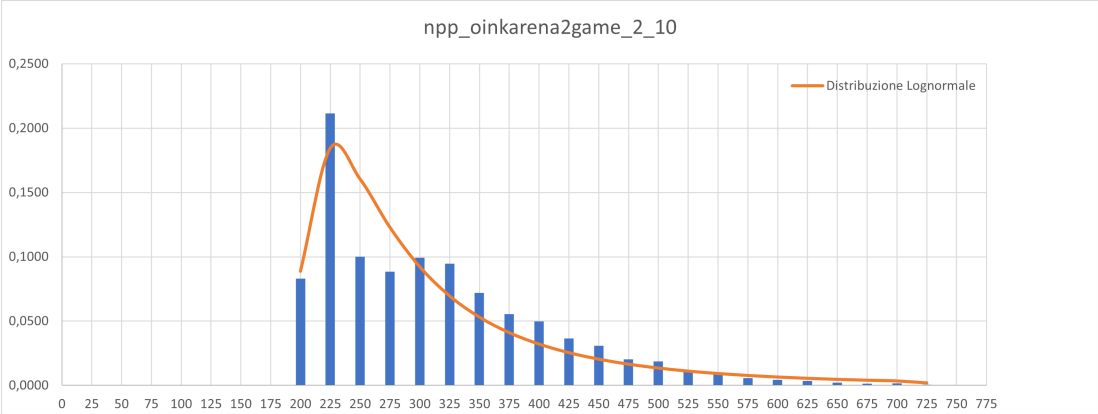

Figura 4.13: Istogramma delle frequenze relativo ai giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti utilizzando il comando di creazione del GamesOnGraphTool a partire dall' arena estrapolata dai giochi generati col tool di OINK.

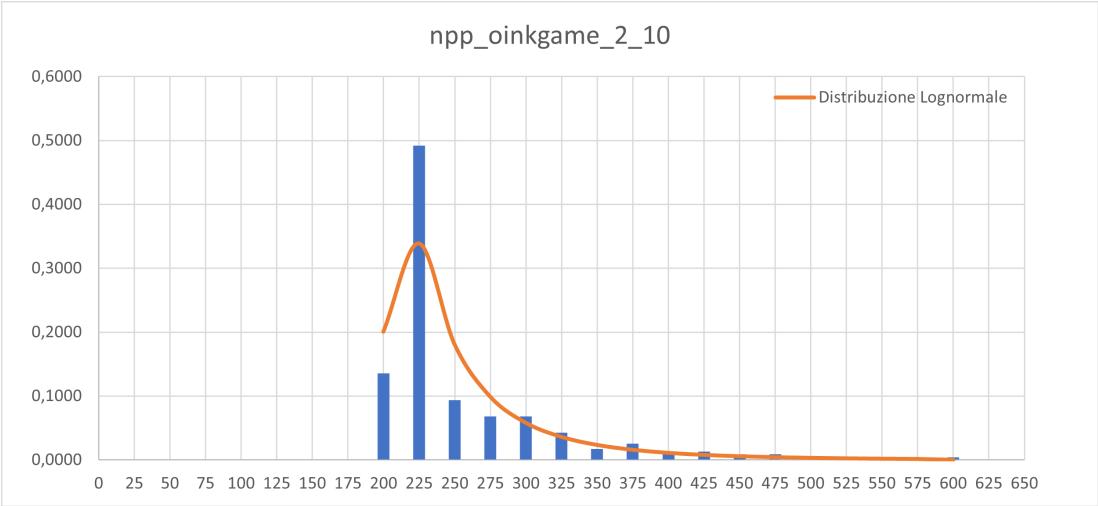

Figura 4.14: Istogramma delle frequenze relativo ai giochi con un numero di mosse per posizione che va da 2 a 10 per ogni posizione ottenuti interamente tramite il tool di OINK

Anche per le soluzioni ottenute eseguendo l'algoritmo ZLK si è potuto constatare che la distribuzione notevole che meglio approssima l'andamento della probabilità in base ai tempi è ancora quella lognormale, anche se non si adatta allo stesso modo in cui si adatta per le soluzioni ottenute con l'algoritmo NPP. Ecco i grafici che ci hanno portato a questa conclusione.

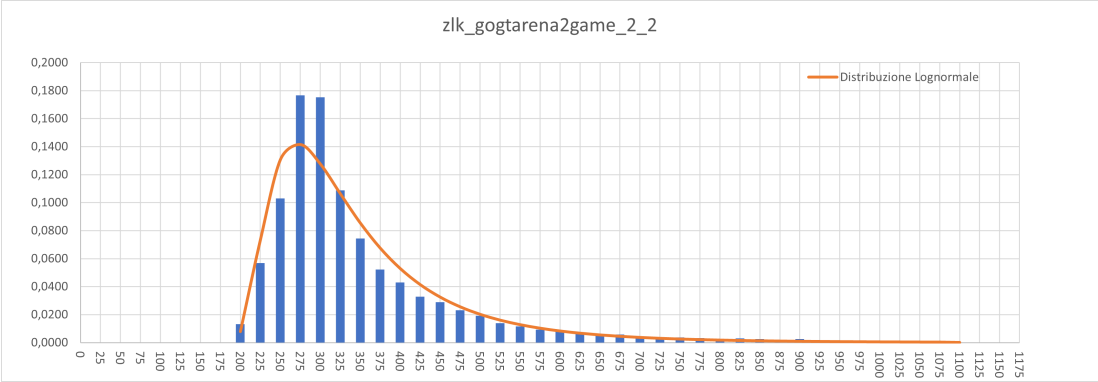

Figura 4.15: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con 2 mosse per ogni posizione ottenuti a partire dall'arena estrapolata dai giochi generati col GamesOnGraphTool

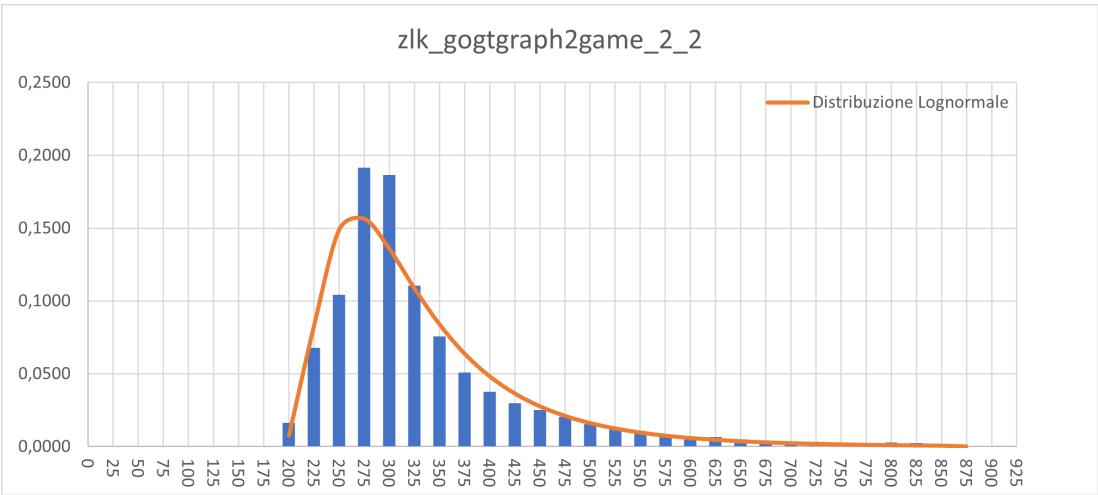

Figura 4.16: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con 2 mosse per ogni posizione ottenuti a partire dal grafo estrapolato dai giochi generati col GamesOnGraphTool.

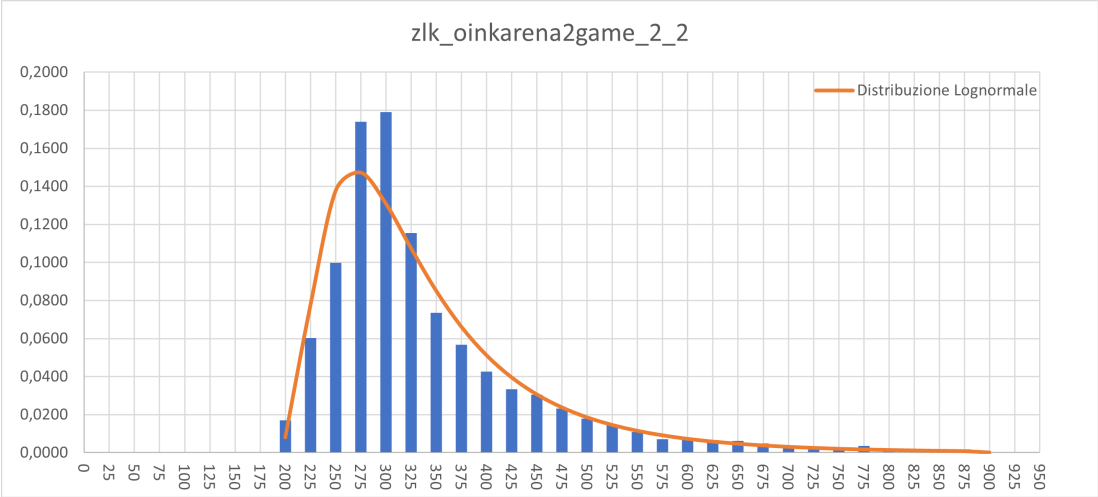

Figura 4.17: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con 2 mosse per ogni posizione ottenuti utilizzando il comando di creazione del GamesOnGraphTool a partire dall' arena estrapolata dai giochi generati col tool di OINK.

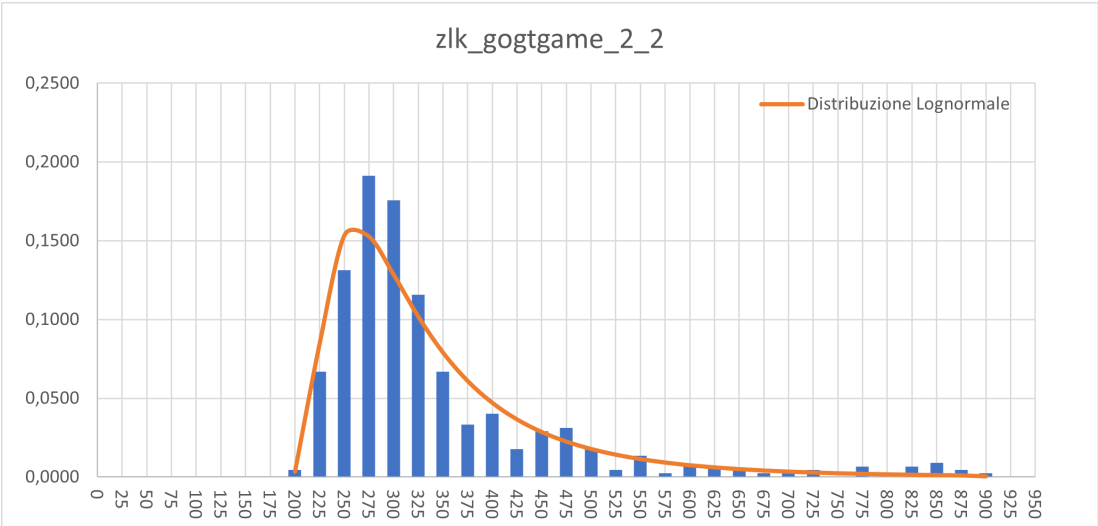

Figura 4.18: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con 2 mosse per ogni posizione generati utilizzando il GamesOnGraphTool.

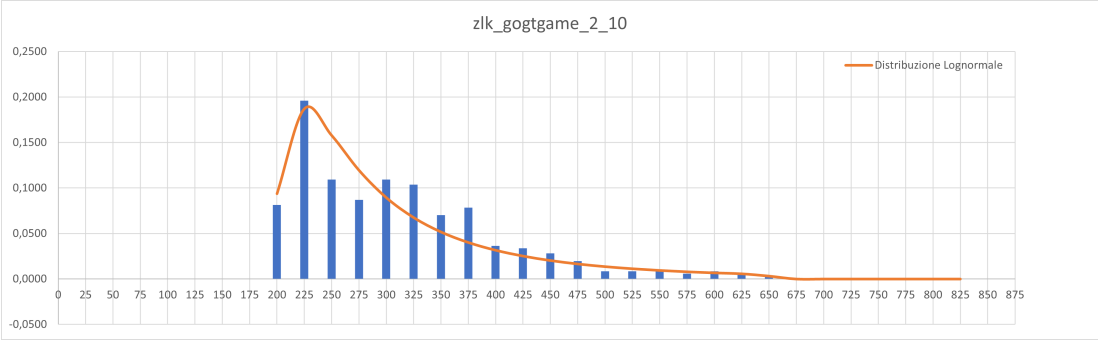

Figura 4.19: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con un numero di mosse che va da 2 a 10 per ogni posizione generati utilizzando il GamesOnGraphTool.

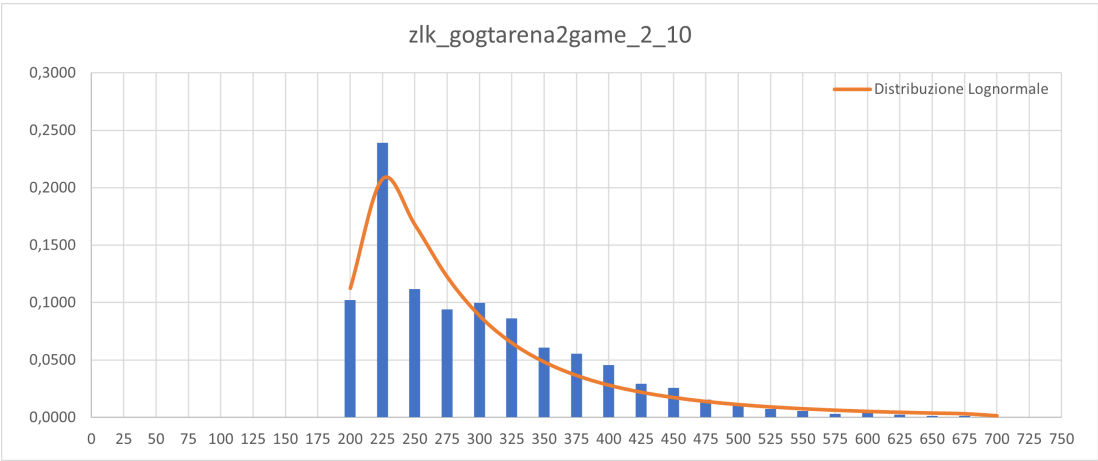

Figura 4.20: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con un numero di mosse che va da 2 a 10 per ogni posizione ottenuti a partire dall' arena estrapolata dai giochi generati col GamesOnGraphTool.

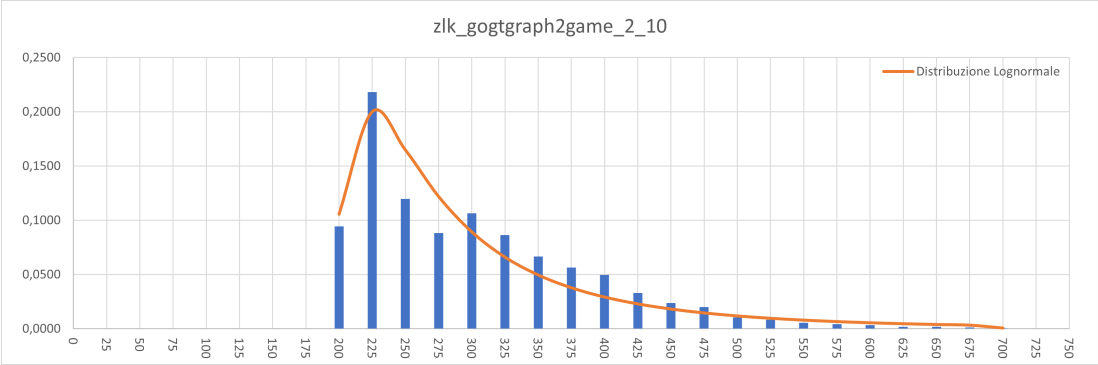

Figura 4.21: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con un numero di mosse che va da 2 a 10 per ogni posizione ottenuti a partire dal grafo estrapolato dai giochi generati col GamesOnGraphTool.

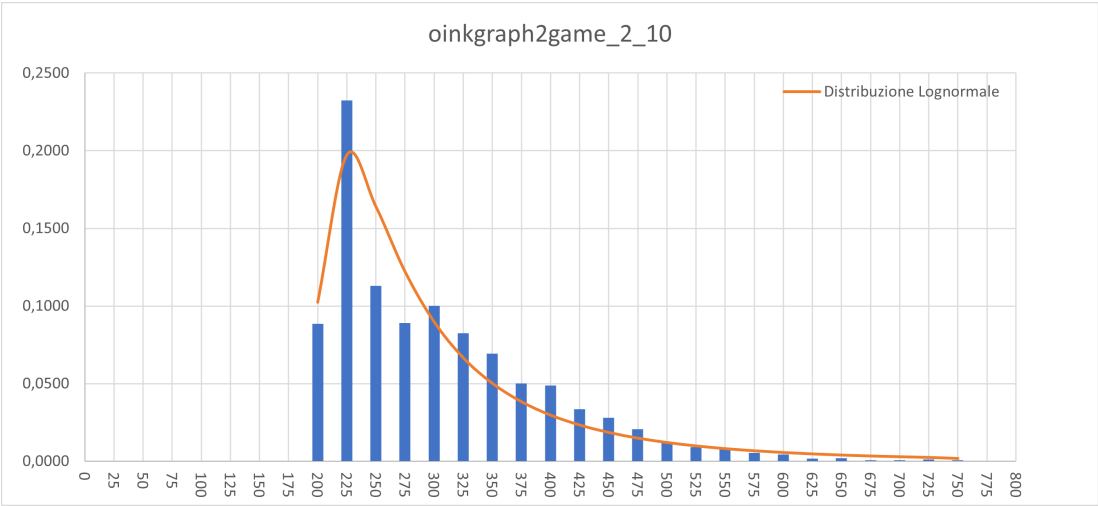

Figura 4.22: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con un numero di mosse che va da 2 a 10 per ogni posizione ottenuti utilizzando il comando di creazione del GamesOnGraphTool a partire dal grafo estrapolato dai giochi generati col tool di OINK.

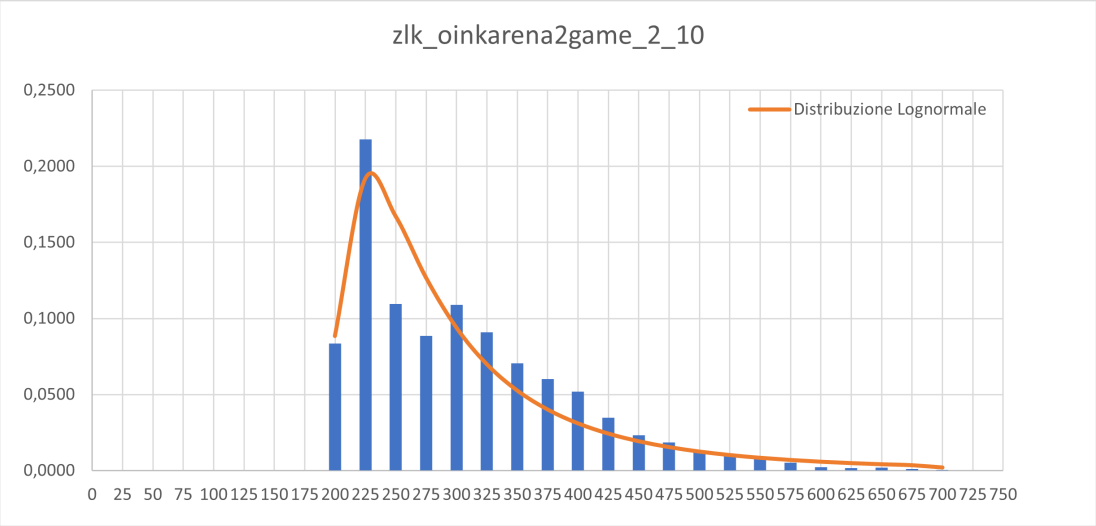

Figura 4.23: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con un numero di mosse che va da 2 a 10 per ogni posizione ottenuti utilizzando il comando di creazione del GamesOnGraphTool a partire dall'arena estrapolata dai giochi generati col tool di OINK.

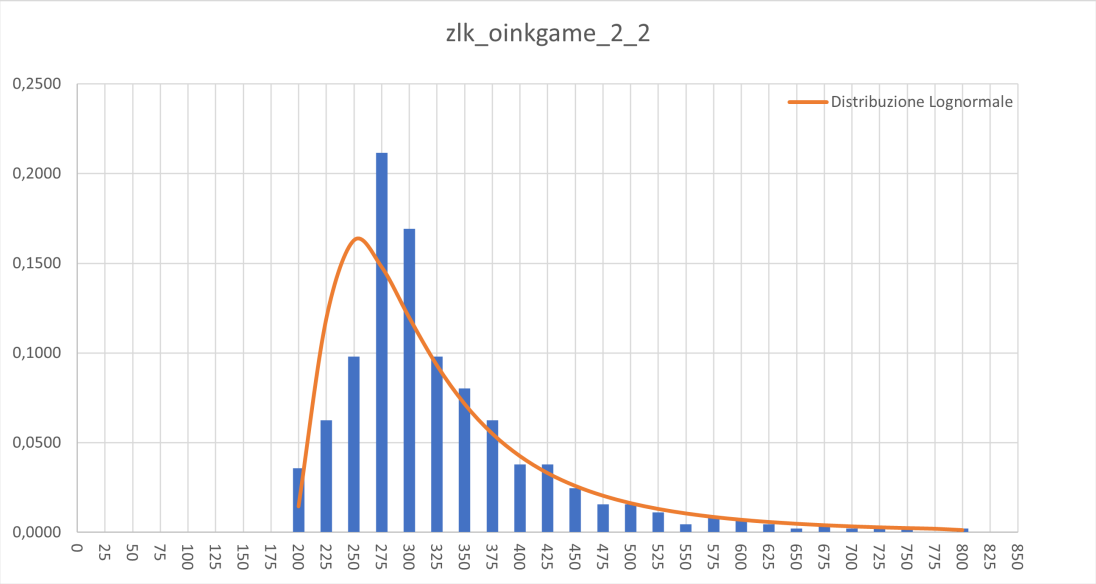

Figura 4.24: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con 2 mosse per ogni posizione generati utilizzando il tool di OINK.

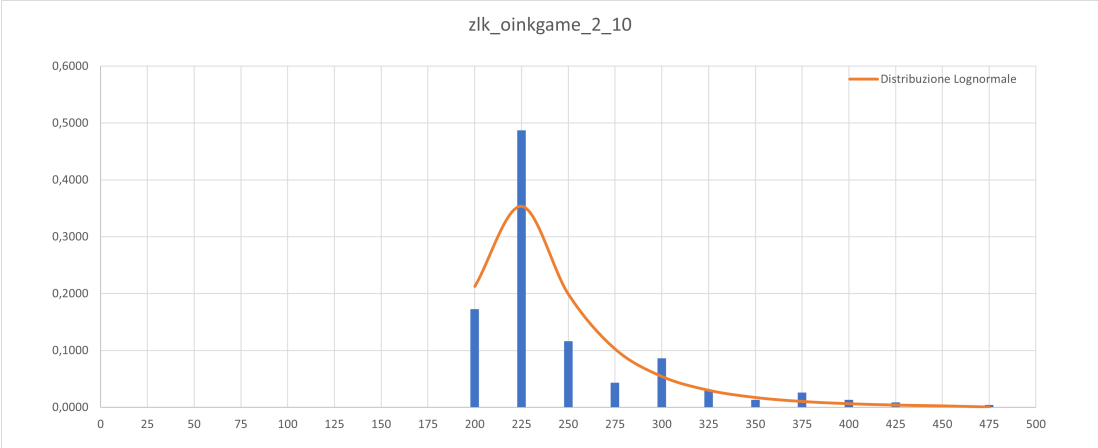

Figura 4.25: Istogramma delle frequenze dei tempi che l'algoritmo ZLK impiega per la risoluzione dei giochi con un numero di mosse che va da 2 a 10 per ogni posizione generati utilizzando il tool di OINK.

Una volta visto che la distribuzione lognormale è quella che meglio approssima sia i tempi calcolati con l'algoritmo NPP che quelli calcolati con l'algoritmo ZLK, sia per i giochi generati col GamesOnGraphTool che per quelli generati interamente col tool di OINK, sia per i giochi con 2 mosse per posizione che per quelli con un numero di mosse che va da 2 a 10, possiamo andare a confrontare direttamente le funzioni di distribuzione di probabilità. Prenderemo in considerazione quanto è piccata o allungata la distribuzione. Più la distribuzione risulta essere piccata minore sarà la varianza, più i valori dei tempi saranno vicini alla media e più il picco si trova verso sinistra più bassa sarà tale media. Più allungata è la distribuzione minore sarà l'altezza del picco e più alta sarà la varianza, cioè maggiormente diversi tra loro saranno i tempi di risoluzione e maggiormente diversi tra loro saranno i giochi generati.

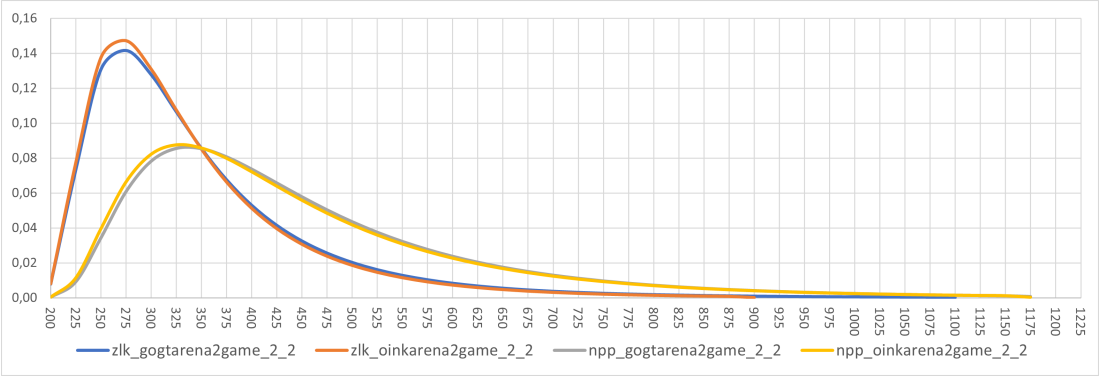

Figura 4.26: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con 2 mosse per posizione ottenuti a partire dall'arena estratta da giochi generati con il GamesOnGraphTool e da quella estratta da giochi generati con il tool di OINK.

Da questo grafico possiamo notare come l'algoritmo NPP sia più lento dell'algoritmo ZLK sia per i giochi con 2 mosse per posizione ottenuti a partire dall'arena estratta da giochi generati con il GamesOnGraphTool sia da quelli ottenuti dall'arena estratta da giochi generati con il tool di OINK.
Risultano, invece, molto simili tra loro i tempi ottenuti con lo stesso algoritmo. Possiamo dire che per lo stesso algoritmo non c'è dissomiglianza tra i primi giochi e i secondi e che la generazione del grafo e dell'arena sottostante non impatta sulla differenza dei tempi, ciò significa che queste vengono create in modo simile in entrambi i tool.

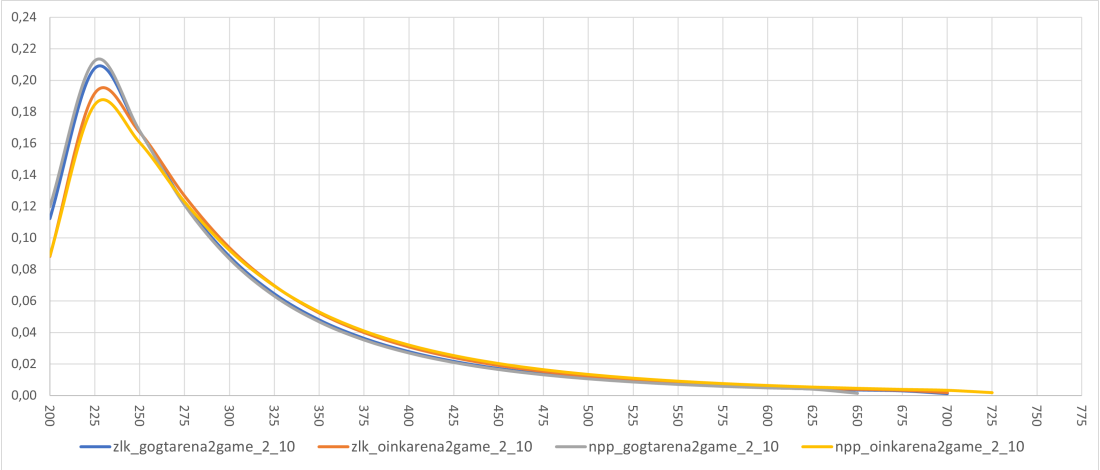

Figura 4.27: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti a partire dall'arena estratta da giochi generati con il GamesOnGraphTool e da quella estratta da giochi generati con il tool di OINK.

I giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti a partire dall'arena estratta da giochi generati con il GamesOnGraphTool e quelli ottenuti dall'arena estratta da giochi generati con il tool di OINK, risultano essere sia simili tra di loro che per entrambi gli algoritmi di risoluzione. I tempi risultano in leggero favore per i giochi generati completamente tramite il GamesOnGraphTool, rispetto a quelli che hanno di base il grafo e l'arena del tool di OINK.

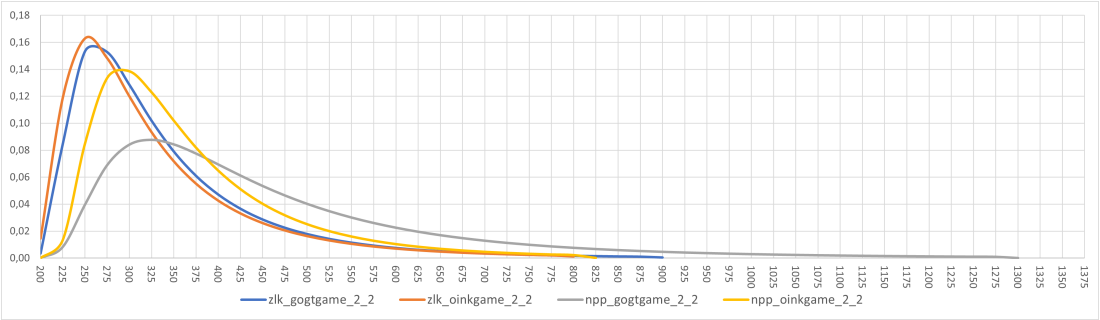

Figura 4.28: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con 2 mosse per posizione generati con il GamesOnGraphTool e con il tool di OINK.

Da qui si evince che per l'algoritmo ZLK i giochi con 2 mosse per posizione generati interamente col GamesOnGraphTool e quelli generati interamente col tool di OINK sono simili tra

di loro; per l'algoritmo NPP i primi, invece, risultano più complessi in quanto hanno tempi di risoluzione maggiori. Anche qui si nota come l'algoritmo ZLK sia più veloce di quello NPP.

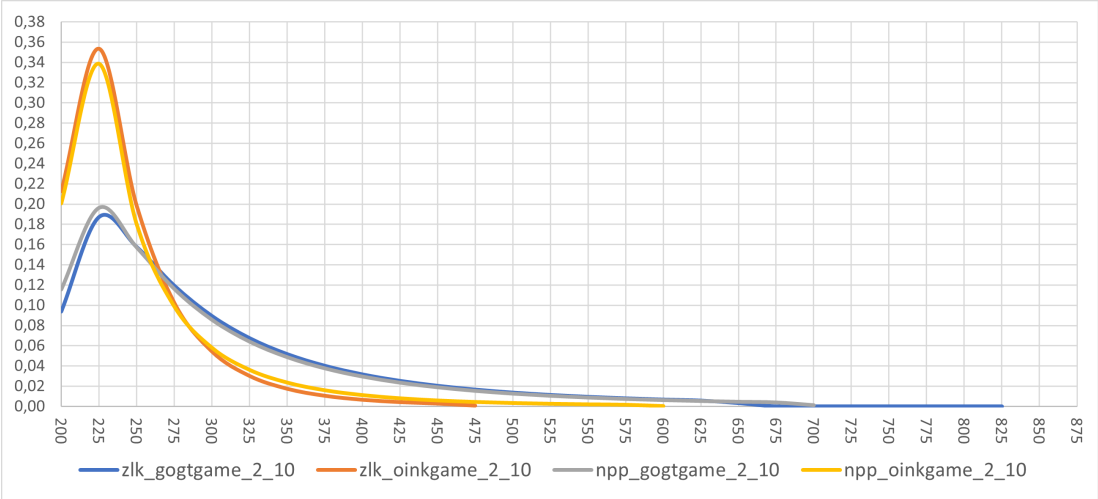

Figura 4.29: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con un numero di mosse per posizione che va da 2 a 10 generati con il GamesOnGraphTool e con il tool di OINK.

I giochi aventi un numero di mosse per posizione che va da 2 a 10 interamente generati col GamesOnGraph tool, risultano, più complessi di quelli generati interamente col tool di OINK, sia per l'algoritmo ZLK che per quello NPP.

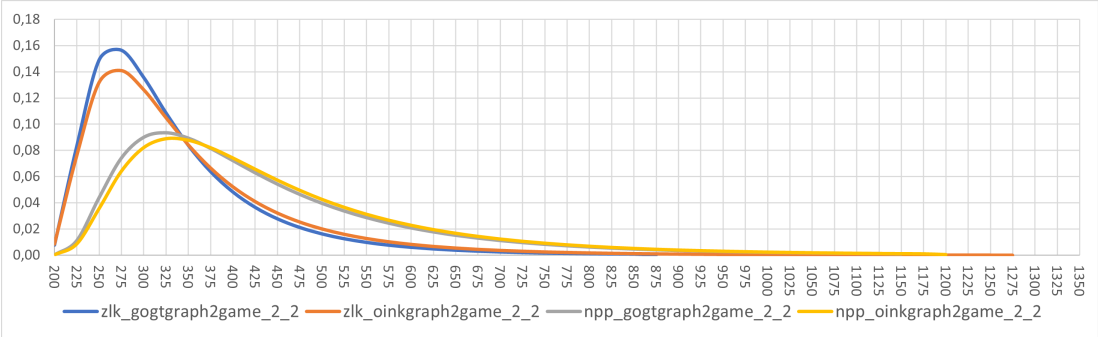

Figura 4.30: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con 2 mosse per posizione ottenuti a partire dal grafo estratto da giochi generati con il GamesOnGraphTool e da quello estratto da giochi generati con il tool di OINK.

Anche da qui si evince che l'algoritmo NPP è più lento dell'algoritmo ZLK. Si osserva che la generazione dei grafi alla base dei giochi con 2 mosse per posizione è simile per entrambi i

tool o non interessante ai fini dei tempi di risoluzione.

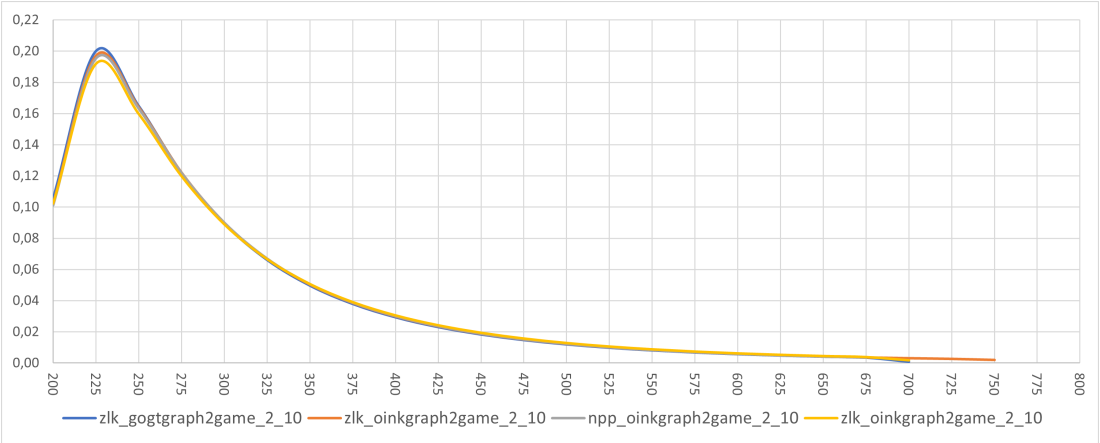

Figura 4.31: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti a partire dal grafo estratto da giochi generati con il GamesOnGraphTool e da quello estratto da giochi generati con il tool di OINK.

I giochi con un numero di mosse per posizione che va da 2 a 10 ottenuti a partire dal grafo estratto da giochi generati con il GamesOnGraphTool e quelli ottenuti dal grafo estratto da giochi generati con il tool di OINK, risultano essere sia simili tra di loro che per entrambi gli algoritmi di risoluzione.

Dall'unione di questi grafici si possono ottenere ulteriori informazioni importanti.

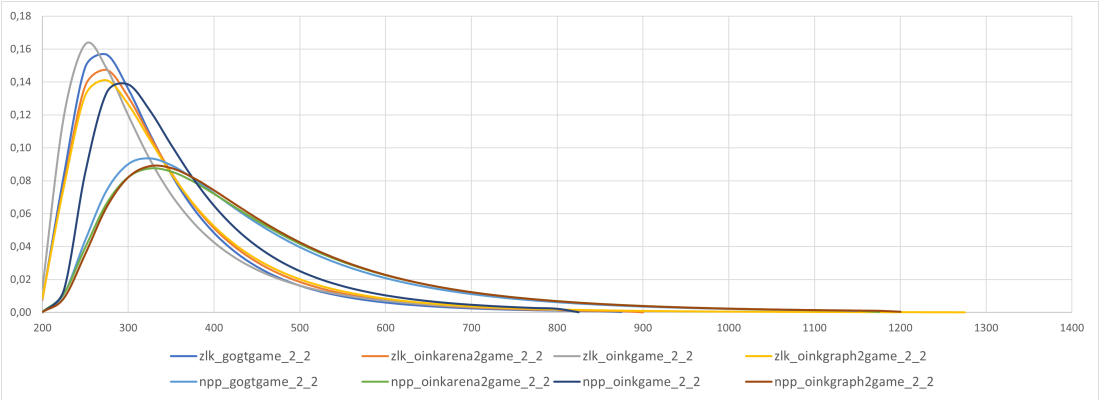

Figura 4.32: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con 2 mosse per posizione

L'algoritmo ZLK risulta essere più veloce dell'algoritmo NPP sia per i giochi con 2 mosse creati col GamesOnGraphTool sia per quelli con 2 mosse creati col tool di OINK.

I giochi con 2 mosse interamente creati dal GamesOnGraphTool sono più complessi di quelli con 2 mosse creati interamente dal tool di OINK per l'algoritmo NPP, mentre per l'algoritmo ZLK la differenza è quasi trascurabile ma leggermente a favore dei primi.

Se prendiamo i giochi interamente creati dal tool di OINK, estraiamo da questi il grafo e a parire da questo col GamesOnGraphTool andiamo a generare un nuovo gioco noteremo che quest' ultimo avrà, per l'algoritmo NPP, gli stessi tempi di risoluzione di quelli generati interamente col GamesOnGraphTool e dunque quello che ne viene fuori è un aumento della complessità. La stessa cosa accade se estraiamo l'arena e ne generiamo il gioco a partire da questa. Ciò vuol dire che la generazione del grafo e dell'arena risulta essere simile per entrambi i tool, in quanto questi non hanno impatto sulla differenza dei tempi di risoluzione, mentre la generazione della priorità risulta essere migliore per il GamesOnGraphTool poichè utilizzando quelle prodotte da questo i giochi diventano più complessi.

Per l'algoritmo ZLK invece non vi è alcuna differenza tra i giochi prodotti dai due tool.

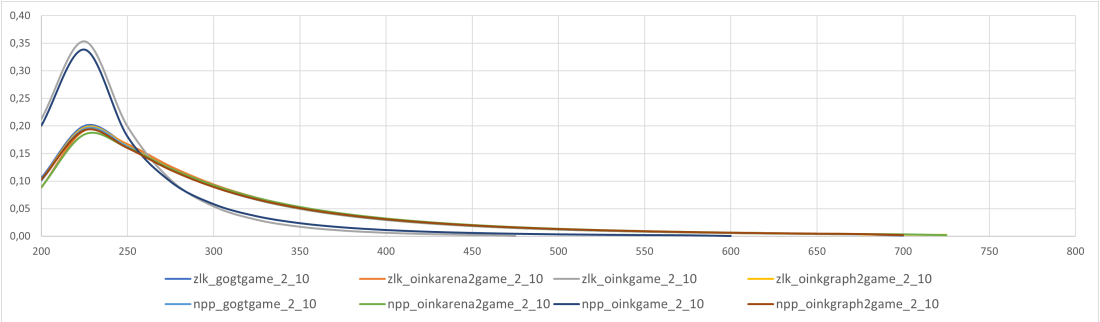

Figura 4.33: Grafico di comparazione delle distribuzioni lognormali delle probabilità dei tempi di risoluzione calcolati con gli algoritmi ZLK e NPP per giochi con un numero di mosse che va da 2 a 10.

Cioè che si deduce da questo grafico è che per il calcolo dei tempi di risoluzione dei giochi con un numero di mosse per posizione che va da 2 a 10 i due algoritmi ZLK e NPP impiegano gli stessi tempi ed hanno lo stesso comportamento: i giochi interamente generati col tool di OINK sono meno complessi di quelli generati col GamesOnGraphTool. Se poi prendiamo i primi e vi estraiamo prima l'arena e poi il grafo, i giochi generati dal GamesOnGraphTool a partire da questi vedranno la loro complessità aumentare.

4.3 Conclusioni

Il GamesOnGraphTool nella versione corrente risulta essere uno strumento molto utile per il confronto e la valutazione degli algoritmi di risoluzione di giochi di parità. E' un tool dalle buone potenzialità che non si prefigge di rimanere nello stato attuale ma porta con sé le intenzioni di essere ampliato al fine di diventare un tool di riferimento nel campo della ricerca dei giochi su grafi.

Quello che consente di fare nello stato corrente è: permettere la creazione di giochi di parità random (nel formato definito nella sezione "Specifica dei formati"), estrarre da giochi specificati nel formato PGSolver l'arena ed il grafo, e crearne di nuovi a partire da questi. Tutto ciò consente di andare a confrontare tra di loro algoritmi di risoluzione esterni al tool stesso, e di confrontare funzioni di generazione di altri tool con quella (unica) del GamesOnGraphTool.

Ciò che ci si aspetta possa essere implementato in futuro è, come prima cosa, l'ampliamento dell'insieme dei giochi supportati e quindi un arricchimento dei comandi di creazione ed estrazione affinchè possano essere utilizzati, oltre che per i giochi di parità, anche per gli altri giochi le cui specifiche dei formati sono già state descritte (buchi, co-buchi, safety, mean-payoff, discounted payoff) e non solo; poter confrontare e valutare non solo l'efficienza degli algoritmi ma anche quella delle strutture dati utilizzate per la rappresentazione dei giochi, motivo per cui ci si è soffermati molto sulla progettazione di una buona parte astratta della libreria; poter scegliere il formato con cui effettuare sia il parsing che il printing, come ad esempio DIMACS e GraphML, con cui in rete sono definiti enormi database di grafi e che permetterebbero il confronto con un maggior numero di tool di generazione; permettere l'aggiunta facile e veloce di diversi algoritmi di risoluzione in modo tale da non dover attingere a fonti esterne e sparse.

Bibliografia

Capitolo 1

[1] Jurdzinski M. *Deciding the winner in parity games is in UP ∩ co-UP*. 1998, pp. 119–124.

[2] Zwick U. e Paterson M. *The complexity of mean payoff games on graphs*. 1996, pp. 343–359.

Capitolo 2

[1] Friedman O. e Lange M. *The PGSolver collection of Parity Game Solvers*. 2017, pp. 31–33.

Capitolo 3

[1] Tom van Dijk. *Oink*. 2017. URL: `https://github.com/trolando/oink` (visitato il 30/07/2022).

[2] Build Framework Group. *Lyra*. URL: `https://www.bfgroup.xyz/Lyra/` (visitato il 18/08/2022).

[3] Friedman O. e Lange M. *The PGSolver collection of Parity Game Solvers*. 2017, pp. 31–33.

Ringraziamenti

Desidero ringraziare innanzitutto il relatore di questa tesi, il professor Fabio Mogavero, per la disponibilità, la scrupolosità, l'attenzione dedicatami, con cui ho voluto ad ogni costo svolgere l'attività di tirocinio, volontà dettata dal piacere, dal coinvolgimento che ha trasmesso durante le sue lezioni e dall'interessamento che ha saputo suscitare per la sua materia. Professore per cui provo grande ammirazione per la passione, la bravura e l'estrema dedizione con cui svolge il proprio lavoro.

Ringrazio i miei familiari per il sostegno che sono riusciti a darmi durante tutto il percorso universitario, per essersi messi a disposizione in ogni modo possibile ed immaginabile, per l'ottimismo che li ha contraddistinti, per avermi insegnato a vedere i fallimenti come un mezzo per maturare e a godere al meglio dei tanti successi ottenuti.

Ringrazio tutti i miei amici, quelli conosciuti in ambito universitario con cui ho avuto il piacere di condividere questa esperienza e che hanno reso il percorso leggero e gioioso. Quelli che ci sono da sempre, che mi hanno aiutato ad affrontare la vita al di fuori dell'università e che si sono dovuti subire tante volte la mia assenza, o magari goderne, chissà. Vi voglio bene.

Ringrazio Maria Gabriella Cecilia, con cui ho la gioia di condividere la mia quotidianità, si, senza virgola tra i nomi, una sola persona, una ma, veramente, con la determinazione e la forza di volontà di tre persone insieme, per avermi trasmesso tali virtù senza le quali non sarei giunto a questo punto.