Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

CORSO DI LAUREA IN INFORMATICA

TESI DI LAUREA TRIENNALE

Calcolo e performance di equilibri di Nash per il gioco della k-colorazione generalizzata

Relatore

Prof. Gianpiero Monaco

Laureando

Valentino Di Giosaffatte

ANNO ACCADEMICO 2017 - 2018

Indice

Parte I

Equilibri di Nash

Capitolo 1

Introduzione agli equilibri di Nash

1.1 Teoria dei giochi

La teoria dei giochi è la disciplina scientifica che si occupa dello studio e dell'analisi del comportamento e delle decisioni di soggetti razionali in un contesto di interdipendenza strategica.

Si definisce interdipendenza strategica, o interazione strategica, lo scenario in cui le decisioni di un individuo influenzano anche le scelte e gli scenari relativi agli altri individui.

Il principale oggetto di studio della teoria dei giochi sono le situazioni di conflitto nelle quali gli attori sono costretti ad intraprendere una strategia di competizione o cooperazione. Tale scenario è definito gioco strategico e gli individui sono denominati giocatori.

Sulla base delle premesse e delle regole compositive del gioco in oggetto, viene costruito un modello matematico nel quale ciascun giocatore effettua le proprie decisioni (mosse migliorative) seguendo una strategia finalizzata ad aumentare il proprio vantaggio netto.

A ciascuna scelta positiva corrisponde un ritorno favorevole in termini di beneficio (payoff), il medesimo concetto vale in modo contrario in caso di scelta negativa, in tal caso il ritorno sarà sfavorevole.

In tali scenari le decisioni di un soggetto possono influire direttamente sui risultati conseguibili dagli altri e viceversa secondo un meccanismo di retroazione.

La teoria dei giochi è un concetto di soluzione applicabile ad un ingente molteplicità di casi nei quali una pluralità di agenti decisionali possono operare in maniere competitiva, seguendo interessi contrastanti, o in maniera cooperativa, seguendo l'interesse comune.

1.1.1 Giochi non-cooperativi

In questo documento, la trattazione sarà incentrata sull'analisi di una particolare tipologia di giochi : i giochi non-cooperativi.

I giochi non-cooperativi, detti anche competitivi, rappresentano una specifica classe di giochi nella quale i giocatori, indipendentemente dai propri obiettivi, non possono stipulare accordi vincolanti di cooperazione (anche normativamente).

Il criterio di comportamento razionale adottato nei giochi non-cooperativi è di carattere individuale ed è denominato strategia del massimo.

La suddetta definizione di razionalità va a modellare il comportamento di un individuo intelligente e ottimista che si prefigge l'obiettivo di prendere sempre la decisione che consegue il massimo guadagno possibile, perseguendo di conseguenza sempre la strategia più vantaggiosa per se stesso.

Si parla dunque di punto di equilibrio qualora nel gioco esista una strategia che presenti il massimo guadagno per tutti i giocatori, ovvero uno stato stabile del gioco nel quale tutti gli attori ottengono il massimo profitto individuale e collettivo.

1.2 Equilibri di Nash

La precedente affermazione muove l'oggetto della trattazione verso l'argomento centrale di questo studio, ovvero gli equilibri di Nash.

L'equilibrio di Nash è una combinazione di strategie nella quale ciascun giocatore effettua la migliore scelta possibile, seguendo cioè una strategia dominante, sulla base delle aspettative di scelta dell'altro giocatore.

L'equilibrio di Nash è la combinazione di mosse $(m1, m2)$ in cui la mossa di ciascun giocatore è la migliore risposta alla mossa effettuata da un altro giocatore.

Ciascun giocatore formula delle aspettative sulla scelta dell'altro giocatore e in base a queste decide la propria strategia, con l'obiettivo di massimizzare il proprio profitto e di conseguenza quello degli altri.

Un equilibrio di Nash è un equilibrio stabile, poiché nessun giocatore ha interesse a modificare la propria strategia.

Ciascun giocatore trae la massima utilità possibile dalle proprie scelte, tenendo conto della migliore scelta dell'altro giocatore, e dunque qualunque variazione alla propria strategia potrebbe soltanto peggiorare il proprio valore di tornaconto (payoff o utilità).

L'equilibrio di Nash è conosciuto anche con il nome di equilibrio non-cooperativo poiché rappresenta una situazione di equilibrio ottimale per un gioco non-cooperativo.

L'equilibrio di Nash non deriva dall'accordo tra i giocatori, bensì dall'adozione di strategie dominanti perseguite da tutti i giocatori, tali da garantire sia il miglior profitto possibile per ciascun giocatore (ottimo individuale), sia il miglior equilibrio collettivo (ottimo sociale).

1.2.1 Definizione formale

Definiamo ora alcuni concetti basilari e chiariamo alcuni aspetti matematici della teoria dei giochi in modo da delineare in modo più accurato il concetto di equilibrio di Nash.

Un gioco è caratterizzato da :

- un insieme G di giocatori, o agenti, che indicheremo con $i = 1, \ldots, N$
- un insieme S di strategie, costituito da un insieme di M vettori

  $$S_i = (s_{i,1}, s_{i,2}, \ldots, s_{i,j}, \ldots, s_{i,M_i})$$

  ciascuno dei quali contiene l'insieme delle strategie che il giocatore *i-esimo* ha a disposizione, cioè l'insieme delle azioni che esso può compiere.
  (indichiamo con s_i la strategia scelta dal giocatore i)
- un insieme U di funzioni

  $$u_i = U_i\,(s_1, s_2, \ldots, s_i, \ldots, s_N)$$

  che associano ad ogni giocatore i il guadagno (detto anche payoff) u_i derivante da una data combinazione di strategie (il guadagno di un giocatore in generale non dipende solo dalla propria strategia ma anche dalle strategie scelte dagli avversari)

Un equilibrio di Nash per un dato gioco è una combinazione di strategie (che indichiamo con l'apice e)

$$s_1^e, s_2^e, ..., s_N^e$$

tale che

$$U_i\left(s_1^e, s_2^e, ..., s_i^e, ..., s_N^e\right) \geq U_i\left(s_1^e, s_2^e, ..., s_i, ..., s_N^e\right)$$

$\forall i$ e $\forall s_i$ scelta dal giocatore *i-esimo.*

Il significato di quest'ultima disuguaglianza è il seguente :
se un gioco ammette almeno un equilibrio di Nash, ciascun agente ha a disposizione almeno una strategia s_i^e dalla quale non ha alcun interesse ad allontanarsi se tutti gli altri giocatori hanno giocato la propria strategia s_j^e.

Come si può facilmente desumere direttamente dalla suddetta disequazione, se il giocatore i gioca una qualunque strategia a sua disposizione diversa da s_i^e, mentre tutti gli altri giocatori hanno giocato la propria strategia s_j^e, può solo peggiorare il proprio guadagno o, al più, lasciarlo invariato.

Da qui si può dedurre quindi che se i giocatori raggiungono un equilibrio di Nash, nessuno può più migliorare il proprio risultato modificando solo la propria strategia, ed è quindi vincolato alle scelte degli altri.

Poiché questo vale per tutti i giocatori, è evidente che se esiste un equilibrio di Nash ed è unico, esso rappresenta la soluzione del gioco, in quanto nessuno dei giocatori ha interesse a cambiare strategia.

1.3 Nozioni di base

1.3.1 Rappresentazione con matrici di payoff e descrizione del procedimento decisionale

Nella seguente matrice di payoff è rappresentato un esempio di equilibrio di Nash in un gioco non-cooperativo a 2 giocatori (tale equilibrio può essere esteso a N giocatori).

		G2	
		S1	**S2**
G1	**S1**	A [2, 2]	B [2, 1]
	S2	C [1, 2]	D [1, 1]

Tabella 1.1: Matrice di payoff

Nella suddetta matrice ciascun giocatore può scegliere la strategia S1 o la strategia S2. Il giocatore 1 si aspetta che il giocatore 2 scelga S1 (strategia dominante) e, quindi, adotta anch'egli la strategia S1 in quanto gli consente di ottenere un payoff individuale pari a 2.

Anche il giocatore 2 formula delle aspettative sulle scelte dell'avversario e si attende che il giocatore 1 scelga S1, scegliendo a sua volta la strategia S1.

L'equilibrio del gioco converge verso la cella A della matrice nella quale entrambi i giocatori massimizzano il proprio payoff individuale (ottimo individuale) dopo aver scelto la propria strategia dominante.

Entrambi i giocatori hanno formulato un'ipotesi sulla migliore scelta del giocatore avversario e, sulla base di questa, hanno scelto la propria strategia dominante.

Cerchiamo ora di comprendere al meglio il funzionamento del processo decisionale alla base dell'equilibrio di Nash.

		G2	
		S1	**S2**
G1	**S1**	A [2, 2]	B [1, 1]
	S2	C [1, 2]	D [2, 1]

Tabella 1.2: Processo decisionale

Il giocatore 1 potrebbe scegliere sia S1 che S2, in entrambi i casi potrebbe sperare di ottenere per sé il payoff più alto (2) nelle celle A e D.

Il giocatore 1 sa bene però che se scegliesse S2, il giocatore 2 sceglierebbe in seguito S1 e l'equilibrio finale si collocherebbe nella cella C, nella quale egli otterrebbe il payoff più basso (1).

Qualora scegliesse invece S1, il giocatore 1 sarebbe consapevole che anche il giocatore 2 sceglierebbe S1 e l'equilibrio finale in questo caso si collocherebbe nella cella A, nella quale il giocatore 1 otterrebbe il payoff più alto (2).

Seguendo il medesimo ragionamento, qualora spettasse al giocatore 2 scegliere per primo, questi sarebbe consapevole che scegliendo S1 anche il giocatore 1 sceglierebbe S1. Dunque anche il questo caso l'equilibrio strategico si collocherebbe nella cella A.

Il giocatore 2 non sceglierebbe mai S2 in quanto in ogni caso avrebbe il payoff più basso (1).

La cella A è un equilibrio di Nash, tutte le altre non lo sono.

1.3.2 Equilibri di Nash e ottimo sociale

		G2	
		S1	**S2**
G1	**S1**	A [2, 2]	B [2, 1]
	S2	C [1, 2]	D [1, 1]

Tabella 1.3: Ottimo sociale

Nell'esempio 1.1 appena trattato possiamo inoltre constatare facilmente che l'equilibrio di Nash trovato (cella A della matrice) è anche un ottimo sociale. Nel suddetto equilibrio coesistono una situazione ottimale per entrambi i giocatori (entrambi i giocatori possiedono un payoff massimo) e una situazione ottimale per l'intera collettività (in quanto la somma dei valori di payoff di entrambi i giocatori è uguale 4, il valore maggiore all'interno della matrice)

1.3.3 Equilibri di Nash multipli

Specifichiamo inoltre che un gioco non-cooperativo può presentare più equilibri di Nash. Anche il presenza di equilibri multipli, ciascun equilibrio è comunque stabile, poiché dalla propria posizione di equilibrio locale qualsiasi scelta è peggiorativa per ogni giocatore.

Al fine di verificare quest'ultima affermazione viene presentata la seguente matrice di payoff nella quale sono presenti 2 equilibri di Nash simmetrici (nelle celle A e D).

		G2	
		S1	**S2**
G1	**S1**	A [2, 2]	B [2, 1]
	S2	C [1, 2]	D [2, 2]

Tabella 1.4: Equilibri di Nash multipli

1.3.4 Assenza di equilibri di Nash

In alcuni casi possono essere del tutto assenti le condizioni per determinare un equilibrio di Nash e di conseguenza molti giochi sono privi di un'istanza di questo concetto di soluzione.

Per verificare tale asserzione mostriamo la matrice di payoff relativa ad un gioco nel quale non è possibile trovare un equilibrio di Nash.

		G2	
		S1	**S2**
G1	**S1**	A [1, 1]	B [1, 0]
	S2	C [2, 1]	D [0, 2]

Tabella 1.5: Assenza di equilibri di Nash

Nell'esempio appena descritto, se il giocatore 1 sceglie la strategia S2 (tentando di ottenere il payoff 2), il giocatore 2 sceglierà la strategia S2 (payoff 2) e l'equilibrio si posizionerà nella cella D della matrice.

Se al contrario è il giocatore 2 a scegliere la strategia S2 (tentando a sua volta di ottenere il payoff 2), il giocatore 1 sceglierà la strategia S1 (payoff 1) e l'equilibrio si posizionerà nella cella B della matrice. Nel suddetto esempio non esiste dunque un equilibrio di Nash.

1.3.5 Il dilemma del prigioniero : sub-ottimalità individuale e sociale

L'adozione di strategie dominanti non assicura sempre un equilibrio di ottimo sociale. In assenza di informazioni, un gioco non-cooperativo potrebbe convergere verso un equilibrio stabile ma non ottimale.

L'ipotesi è dimostrata nel problema "il dilemma del prigioniero" in cui due attori, pur formulando delle aspettative razionali sul comportamento dell'avversario e adottando strategie dominanti, determinano un equilibrio sub-ottimale (D) sia dal punto individuale che sociale.

La seguente matrice di payoff evidenzia il tipico caso del dilemma del prigioniero : vi sono 2 giocatori separati in stanze differenti, senza che abbiano la possibilità di comunicare (informazione imperfetta) ; i 2 giocatori sono accusati di un reato e interrogati contemporaneamente.

- il giocatore che confessa il reato accusando l'altro ottiene la scarcerazione (utilità 9), mentre il giocatore accusato subisce il massimo della pena (utilità 0)
- se entrambi i giocatori evitano di confessare, ai 2 viene applicata un pena molto lieve (utilità 5)
- se entrambi i giocatori confessano, vengono condannati alla pena ordinaria (utilità 2)

		G2	
		non confessa	**confessa**
G1	**non confessa**	A [5, 5]	B [0, 9]
	confessa	C [9, 0]	D [2, 2]

Tabella 1.6: Il dilemma del prigioniero

Nell'esempio il giocatore 1 si aspetta che il giocatore 2 confessi, poiché la strategia dominante del giocatore 2 è confessare, grazie alla confessione quest'ultimo otterrebbe la libertà (payoff 9).

Sulla base di questa aspettativa il giocatore 1 sceglie la propria strategia dominante tra B e D, scegliendo a sua volta di confessare per ottenere un payoff uguale a 2. Lo stesso ragionamento viene adottato, in modo inverso, dal giocatore 2.

L'equilibrio D è un equilibrio di Nash ed è un equilibrio stabile poiché nessuno dei 2 giocatori ha interessa a modificare le proprie scelte.

In tale equilibrio però entrambi i giocatori ottengono un payoff subottimale pari a 2 rispetto a quello che avrebbero se decidessero entrambi di non confessare (payoff 5 - equilibrio A) e in più l'equilibrio D presenta un valore di ottimo sociale (4) inferiore rispetto a quello che si potrebbe ottenere dall'equilibrio A (10) che rappresenta l'ottimo individuale e sociale del gioco.

In conclusione, il dilemma del prigioniero rappresenta una situazione in cui le scelte individuali del giocatori, pur essendo strategie dominanti, determinano un equilibrio inefficiente. Nel dilemma del prigioniero l'equilibrio del gioco non è né un ottimo individuale né un ottimo sociale.

Parte II

Gioco della k-colorazione generalizzata

Capitolo 2

Gioco della k-colorazione generalizzata

Esaminiamo ora gli equilibri di Nash puri per il gioco della k-colorazione generalizzata nel quale viene fornito un grafo non-orientato e un insieme di k colori.

I nodi rappresentano i giocatori e gli archi catturano i loro reciproci interessi.

La strategia di ciascun giocatore è composta da k colori.

L'utilità di un giocatore v in un dato stato o colorazione è data dalla somma dei pesi degli archi (v, u) incidenti a v tale che il colore scelto da v sia diverso da quello scelto da u, più il profitto guadagnato dall'utilizzo del colore scelto.

Per prima cosa dimostriamo che il gioco della k-colorazione generalizzata è convergente e dunque esiste sempre almeno un equilibrio di Nash per ogni istanza del gioco in questione.

Valutiamo dunque in seguito una descrizione delle prestazioni dei giochi della k-colorazione generalizzata per mezzo delle nozioni largamente utilizzate di prezzo dell'anarchia (price of anarchy) e prezzo della stabilità (price of stability).

Forniamo inoltre limiti stretti per 2 tipi di benessere sociale ampiamente utilizzati, il benessere sociale utilitario (utilitarian social welfare) e il benessere sociale egalitario (egalitarian social welfare).

2.1 Descrizione generale

Le istanze appartenenti al gioco della k-colorazione generalizzata sono giocate su grafi non-orientati pesati in cui i nodi corrispondono ai giocatori e in cui gli archi identificano le connessioni sociali o le relazioni tra i giocatori.

Il set di strategie di ciascun giocatore è un insieme di k colori disponibili (assumiamo che i colori siano gli stessi per ogni giocatore).

Quando i giocatori selezionano un colore inducono una colorazione k (o semplicemente una colorazione).

Ciascun giocatore possiede una funzione di profitto legata all'apprezzamento da parte di quest'ultimo del colore scelto (vale per tutti i colori disponibili per il giocatore).

Data una colorazione, l'utilità (o il guadagno) di un giocatore v colorato con il colore i è la somma dei pesi degli archi (v, u) incidenti a v, tale che il colore scelto da v è diverso da quello scelto da u, più il profitto derivante dalla scelta del colore i da parte del giocatore v.

Assumiamo che i giocatori siano egoisti, dunque un concetto di soluzione ben noto per questo tipo di impostazione è l'equilibrio di Nash.

L'equilibrio di Nash è uno dei concetti più importanti nella teoria dei giochi e fornisce una soluzione stabile che è robusta alle deviazioni dei singoli giocatori.

Formalmente, una colorazione è un equilibrio di Nash puro se nessun giocatore può migliorare la propria utilità deviando unilateralmente dalla propria strategia attuale.

L'egoismo dei giocatori può causare in molti casi la perdita di benessere sociale e quindi una soluzione stabile non è sempre buona rispetto al benessere della società.

Consideriamo ora 2 nozioni di benessere sociale, naturali e ampiamente utilizzate.

Data una colorazione k, il benessere sociale utilitario (utilitarian social welfare) è definito come la somma delle utilità dei giocatori nella colorazione k, mentre il benessere sociale egalitario (egalitarian sociale welfare) è definito come l'utilità minima tra tutti i giocatori nella colorazione k.

Utilizziamo inoltre 2 metodi per misurare la bontà di un equilibrio di Nash rispetto a un benessere sociale, il prezzo dell'anarchia (price of anarchy) e il prezzo della stabilità (price of stability).

Il prezzo dell'anarchia descrive, nel peggiore dei casi, come l'efficienza di un sistema degrada a causa del comportamento egoistico dei suoi giocatori, mentre il prezzo della stabilità ha un naturale significato di stabilità, poiché è la soluzione ottimale tra quelle che possono essere accettate da giocatori egoisti.

Studiamo ora l'esistenza e le performance degli equilibri di Nash nei giochi della k-colorazione generalizzata.

Ci concentriamo solo sui grafi non-orientati poiché per i grafi orientati anche il problema di decidere se un'istanza ammetta un equilibrio di Nash è un problema difficile (NP-Hard), inoltre esistono casi per i quali un equilibrio di Nash non esiste affatto.

2.1.1 Nozioni sul problema

Sappiamo che, in caso di grafi non-orientati non-pesati, sia possibile calcolare un equilibrio di Nash in tempo polinomiale.

Nel nostro caso, il problema di calcolare un equilibrio di Nash su grafi non-orientati pesati è PLS-Completo anche per $k = 2$, dato che il gioco del taglio massimo (Max-Cut game) è un caso speciale del nostro gioco.

Proprio riguardo questo aspetto è bene delineare la relazione che esiste tra il gioco della k-colorazione generalizzata e il gioco del taglio massimo, un problema molto importante e ampiamente trattato in letteratura.

Il gioco della k-colorazione generalizzata è un estensione del gioco del taglio massimo, infatti quest'ultimo può essere ottenuto ponendo a 0 il valore dei profitti relativi ai colori e ponendo a 2 il numero di colori presenti nel set disponibile per ciascun giocatore.

Inoltre il gioco della k-colorazione generalizzata è un'estensione del gioco della k-colorazione nel quale vi sono k-colori ma i profitti relativi ai colori sono impostati a 0.

2.2 Dettagli sul modello

Dato un grafo semplice non-orientato $G = (V, E, w)$, dove $|V| = n$, $|E| = m$ e $w : E \to \mathbb{R}_{\geq 0}$ è la funzione relativa ai pesi sugli archi che associa un peso positivo a ciascun arco.

Denotiamo con $\delta^v(G) = \sum_{u \in V:\{v,u\} \in E} w(\{v, u\})$ la somma dei pesi di tutti gli archi incidenti a v.

L'insieme dei nodi con cui un nodo v ha un arco in comune è chiamato insieme dei vicini di v (insieme dei nodi adiacenti a v).

Un'istanza di gioco della k-colorazione generalizzata è una tupla (G, K, P). $G = (V, E, w)$ è un grafo pesato non-orientato senza self loops, in cui ogni $v \in V$ è un giocatore egoista.

k è un insieme di colori disponibili (assumiamo $k \geq 0$).

Il set di strategia di ciascun giocatore è dato dai k colori disponibili, ovvero i giocatori hanno lo stesso insieme di azioni.

Denotiamo con $P : V \times K \to \mathbb{R}_{\geq 0}$ la funzione di profitto del colore, che definisce quanto un giocatore apprezza un colore, ovvero se il giocatore v scegli di usare il colore i, allora guadagna $P_v(i)$.

Per ciascun giocatore v, definiamo P_v^M come il massimo profitto che v può guadagnare da un colore, formalmente $P_v^M = max_{i=1,\ldots,k} P_v(i)$.

Quando $P_v(i) = 0 \forall v \in V$ e $\forall i \in k$, si ha il caso in cui non vi sono profitti associati ai colori scelti, quindi possiamo riferirci a questo gioco come a un gioco della k-colorazione (graph k-coloring game).

Uno stato del gioco $c = \{c_1, \ldots, c_n\}$ è una k-colorazione, o semplicemente una colorazione, dove c_v è il colore (cioè un numero $1 \leq c_v \leq k$) scelto dal giocatore v.

In una determinata colorazione c, il payoff (o l'utilità) di un giocatore v è la somma dei pesi degli archi (v, u) incidenti a v, tale che il colore scelto da v è diverso da quello scelto da u, oltre al profitto ottenuto dall'utilizzo del colore scelto.

In modo formale, per una colorazione c, il payoff di un giocatore v è $\mu_c(v) = \sum_{u \in V:\{v,u\} \in E \wedge c_v \neq c_u} w(\{v, u\}) + P_v(c_v)$.

Quando un arco (v,u) fornisce utilità ai suoi endpoints in una colorazione c, cioè quando $c_v \neq c_u$, diciamo che tale arco è corretto.

Diciamo anche che un arco (v,u) è monocromatico in una colorazione c quanto $c_v = c_u$.

Sia c_{-v}, c'_u la colorazione ottenuta da c cambiando la strategia del giocatore v da c_v a c'_v.

Data una colorazione $c = \{c_1, \ldots, c_n\}$, una mossa migliorativa (improving move) del giocatore v nella colorazione c è una strategia c'_v tale che $\mu_{(c_{-v}, c'_v)}(v) > \mu_c(v)$.

Uno stato del gioco è un equilibrio di Nash puro o equilibrio stabile se e so se nessun giocatore può effettuare una mossa migliorativa.

In modo formale, $c = \{c_1, \ldots, c_n\}$ è un equilibrio di Nash se $\mu_c(v) \geq \mu_{(c_{-v}, c'_v)}(v)$ per ogni possibile colorazione c'_v e per ogni giocatore $v \in V$.

Una dinamica di miglioramento (improving dynamic), o brevemente dinamica (dynamic), è una sequenza di mosse migliorative. Si dice che un gioco sia convergente se, dato un qualsiasi stato iniziale c, qualsiasi sequenza di mosse migliorative porta a un equilibrio di Nash.

Data una colorazione c, definiamo una funzione di benessere sociale utilitario (utilitarian social welfare) $SW_{UT}(c)$ e una funzione di benessere sociale egalitario (egalitarian sociale welfare) $SW_{EG}(c)$ come segue :

$$SW_{UT}(c) = \sum_{v \in V} \mu_c(v) = \sum_{v \in V} P_v(c_v) + \sum_{\{v,u\} \in E : c_v \neq c_u} 2w(\{v,u\})$$

$$SW_{EG}(c) = min_{v \in V} \mu_c(v)$$

Indichiamo con C l'insieme di tutte le possibili colorazioni e denotiamo con Q l'insieme di tutte le colorazioni stabili.

Data una funzione di benessere sociale SW, definiamo il prezzo dell'anarchia (price of anarchy) (PoA) per il gioco della k-colorazione generalizzata come il rapporto tra il massimo benessere sociale tra tutte le possibili colorazioni sul minimo benessere sociale tra tutte le possibili colorazioni stabili.

In modo formale, $PoA = \frac{max_{c \in C} SW(c)}{min_{c' \in Q} SW(c')}$.

Definiamo inoltre il prezzo della stabilità (price of stability) (PoS) per il gioco della k-colorazione generalizzata come il rapporto tra il massimo be-

nessere sociale tra tutte le possibili colorazioni sul massimo benessere sociale tra tutte le possibili colorazioni stabili.
In modo formale, $PoS = \frac{max_{c \in C} SW(c)}{max_{c' \in Q} SW(c')}$.

Intuitivamente, il PoA (rispettivamente PoS) ci dice quanto è peggiore il benessere sociale nel peggiore (rispettivamente migliore) equilibrio di Nash, relativo al benessere sociale dell'ottimo.

2.2.1 Convergenza ed esistenza degli equilibri di Nash

Per prima cosa mostriamo che il gioco della k-colorazione generalizzata è convergente. Ciò implica chiaramente che gli equilibri di Nash esistono sempre.

Proposizione 1. *$\forall k$, ogni gioco della k-colorazione generalizzata (G, K, P) finito è convergente.*

Notiamo che, da un lato, se il grafo non è pesato, la dinamica, partendo dalla colorazione in cui ogni giocatore v selezione il colore in modo tale da ottenere il massimo profitto possibile, che è, il colore i tale che $P_v(i) = P_v^M$, converge ad un equilibrio di Nash in al massimo $|E|$ mosse migliorative.

D'altra parte, se il grafo è pesato, il calcolo di un equilibrio di Nash è PLS-Completo.

Ne consegue il fatto che, quando $k = 2$, il nostro gioco è una generalizzazione del gioco del taglio (Cut game) che è uno dei primi problemi che si sono dimostrati essere PLS-Completi.

2.3 Benessere sociale utilitario (utilitarian social welfare)

In questa sezione ci concentreremo sul benessere sociale utilitario. Mostriamo limiti stretti per il prezzo dell'anarchia utilitario e tralasciamo quelli per il prezzo della stabilità utilitario.

2.3.1 Prezzo dell'anarchia utilitario (utilitarian price of anarchy)

Ricordiamo che nel caso che presenta l'assenza di profitti associati ai colori, il prezzo dell'anarchia utilitario è esattamente $\frac{k}{(k-1)}$.

Qui dimostriamo che nel gioco della k-colorazione generalizzata il prezzo dell'anarchia utilitario è pari a 2, cioè è indipendente dal numero di colori.

Iniziamo dimostrando che il prezzo dell'anarchia utilitario è al più 2.

Teorema 1. *Il prezzo dell'anarchia per il gioco della k-colorazione generalizzata è al più 2.*

Mostriamo ora che il prezzo dell'anarchia utilitario è almeno 2 anche per il caso speciale di grafi stella non-pesati.

Teorema 2. *Il prezzo dell'anarchia utilitario per il gioco della k-colorazione generalizzata è almeno 2, anche per il caso speciale di grafi stella non-pesati.*

2.4 Benessere sociale egalitario (egalitarian social welfare)

In questa sezione ci concentriamo sul benessere sociale egalitario. Mostriamo limiti stretti per il prezzo dell'anarchia egalitario e tralasciamo quelli per il prezzo della stabilità egalitario.

2.4.1 Prezzo dell'anarchia egalitario (egalitarian price of anarchy)

Per un gioco della k-colorazione (ovvero senza profitti associati ai colori), un limite inferiore per il prezzo dell'anarchia egalitario è fornito.

In tal caso, la soluzione ottimale è tale che ciascun giocatore v ha un'utilità uguale al suo grado δ^v, tuttavia esiste una colorazione stabile in cui ciascun giocatore v ha un'utilità pari a $\frac{(k-1)}{k}\delta^v$, per qualsiasi numero di colori $k \geq 2$.

Questo risultato insieme al fatto che secondo il pidgehole principle, per qualsiasi colorazione stabile ogni giocatore v raggiunge almeno un'utilità pari a $\frac{(k-1)}{k}\delta^v$, implica che il prezzo dell'anarchia egalitario è esattamente

$\frac{(k-1)}{k}$ per il gioco della k-colorazione generalizzata.

Pertanto ora consideriamo il gioco della k-colorazione generalizzata. Per prima cosa notiamo che l'istanza definita nel Teorema 2 fornisce un limite inferiore di 2 per il prezzo dell'anarchia egalitario.

Infatti, nella colorazione ottimale l'utilità minima è 2 ed esiste una colorazione stabile in cui l'utilità minima è 1.

Inoltre, si sottolinea che la dimostrazione del Teorema 1 mostra sostanzialmente che per ogni giocatore i, la propria utilità in un qualsiasi risultato stabile è almeno la metà del profitto che ha nell'ottimo.

Quindi otteniamo facilmente il seguente teorema relativo al prezzo dell'anarchia egalitario per il gioco della k-colorazione generalizzata.

Teorema 3. *Il prezzo dell'anarchia egalitario per il gioco della k-colorazione generalizzata è 2.*

Parte III

Implementazione

Capitolo 3

Implementazione

In questo capitolo andremo a delineare e descrivere gli aspetti fondamentali ed essenziali relativi all'attività di implementazione.

Inizieremo con una presentazione della struttura generale dei programmi assieme ad una descrizione accurata dei componenti utilizzati e delle scelte effettuate in fase di progettazione.

In seguito procederemo con un'attenta analisi degli algoritmi implementati, ovvero l'algoritmo per il calcolo degli equilibri di Nash, quello per il calcolo dell'ottimo relativo alla funzione di benessere sociale utilitario e quello per il calcolo dell'ottimo relativo alla funzione di benessere sociale egalitario.
Tale sezione sarà caratterizzata dall'utilizzo di pseudocodice per ciascuno degli algoritmi trattati, in modo tale da rendere più comprensibile la descrizione dei cicli e delle operazioni.

Tratteremo in modo approfondito questa porzione del documento poiché precede la sezione relativa alla sperimentazione effettuata attraverso i programmi implementati e dunque è di fondamentale importanza.

3.1 Struttura generale

Procediamo presentando la struttura generale relativa programmi implementati.
Quest'ultima è rappresentata attraverso una composizione ad albero che riproduce una porzione di filesystem partendo dalla root del progetto.
Al fine di rendere più chiara la lettura viene inoltre fornita un breve leggenda sulla nomenclatura utilizzata.

- La nomenclatura **nome.dir** indica che l'oggetto è una cartella

- La nomenclatura **nome.edgelist** indica che l'oggetto è un file con estensione .edgelist (oggetto principale modellato dal programma)
- La nomenclatura **nome.dot** indica che l'oggetto è un file con estensione .dot (oggetto utilizzato su macchine GNU/Linux per il disegno attraverso l'uso della libreria PyGraphViz in fase di debug)
- le lettere **X,K,Y,Z,H,W** rappresentano numeri casuali (sono utilizzate per descrivere la moltitudine di cartelle, grafi creati e risultati, ottenuti durante un generale caso d'uso dei programmi)
- la nomenclatura **nome.init** indica che l'oggetto è un file con estensione .init (oggetto utilizzato in fase di lettura per salvare le caratteristiche dei grafi (nodi, archi, pesi, colorazione) e dei colori (colori, profitti))
- la nomenclatura **nome.out** indica che l'oggetto è un file con estensione .out (oggetto utilizzato in fase di lettura per salvare i risultati derivanti da esecuzioni singole o multiple usando gli algoritmi per il calcolo dell'equilibrio di nash, dell'ottimo con funzione di benessere sociale utilitario e dell'ottimo con funzione di benessere sociale egalitario)

Descriviamo ora la funzione basilare di alcuni componenti dell'albero sottostante che rappresenta un esempio generale relativo ad un caso d'uso dei programmi.

- La cartella **generator** contiene al suo interno l'intera struttura relativa al generatore di grafi
- La cartella **gen** contiene i risultati delle generazioni singole di grafi
  1. Al suo interno vi sono X cartelle gen-dir-X (il nome viene assegnato durante l'esecuzione dall'utente) generate dinamicamente
  2. Ciascuna cartella contiene il risultato di una generazione singola di un grafo, ciascuna generazione produce una coppia di file .edgelist e .dot (quest'ultimo utilizzato solo in fase di debug)
  3. Ogni gruppo cartella-file.edgelist-file.dot rappresenta il risultato di una generazione singola
- La cartella **m-gen** contiene i risultati delle generazioni multiple di grafi
  1. Al suo interno vi sono Y cartelle m-gen-dir-Y (il nome viene assegnato durante l'esecuzione dall'utente) generate dinamicamente
  2. Ciascuna cartella contiene il risultato di una generazione multipla di grafi, ciascuna generazione molteplici coppie (nell'esempio K,Z,...) di file .edgelist e .dot (quest'ultimo utilizzato solo in fase di debug)

3. Ogni gruppo cartella-file.edgelist-file.dot-file.edgelist-file.dot-... rappresenta il risultato di una generazione multipla

- Il file **generator.py** contiene al suo interno il codice del generatore di grafi scritto interamente in linguaggio Python

- La cartella **reader** contiene al suo interno l'intera struttura relativa al lettore di grafi

- La cartella **result** contiene i risultati delle letture / sperimentazioni su grafi singoli

  1. Al suo interno vi sono H cartelle result-dir-H (il nome viene preso automaticamente dal grafo in lettura) generate dinamicamente
  2. Ciascuna cartella contiene il singolo risultato di una lettura / sperimentazione su un grafo, ciascun lettura / sperimentazione produce una coppia di file .init e .out
  3. Ogni gruppo cartella-file.init-file.out rappresenta il risultato di una singola lettura / sperimentazione

- La cartella **m-result** contiene i risultati delle letture / sperimentazioni su grafi multipli

  1. Al suo interno vi sono W cartelle m-result-dir-W (il nome viene preso automaticamente dalla cartella relativa alla moltitudine di grafi in lettura) generate dinamicamente
  2. Ciascuna cartella contiene i risultati di letture / sperimentazioni su grafo multipli, l'insieme di tutte le letture / sperimentazioni effettuate produce una coppia di file .init e .out
  3. Ogni gruppo cartella-file.init-file.out rappresenta il risultato di una lettura / sperimentazione multipla

- Il file **reader.py** contiene al suo interno il codice del lettore di grafi scritto interamente in linguaggio Python

```
/
├── generator.dir
│   ├── gen.dir
│   │   ├── gen-dir-1.dir
│   │   │   ├── graph-1.edgelist
│   │   │   └── graph-1.dot
│   │   ├── ....
│   │   └── gen-dir-X.dir
│   │       ├── graph-X.edgelist
│   │       └── graph-X.dot
│   ├── m-gen.dir
│   │   ├── m-gen-dir-1.dir
│   │   │   ├── graph-1.edgelist
│   │   │   ├── graph-1.dot
│   │   │   ├── ....
│   │   │   ├── graph-K.edgelist
│   │   │   └── graph-K.dot
│   │   ├── ....
│   │   └── m-gen-dir-Y.dir
│   │       ├── graph-1.edgelist
│   │       ├── graph-1.dot
│   │       ├── ....
│   │       ├── graph-Z.edgelist
│   │       └── graph-Z.dot
│   └── generator.py
└── reader.dir
    ├── result.dir
    │   ├── result-dir-1.dir
    │   │   ├── graph-1.init
    │   │   └── graph-1.out
    │   ├── ....
    │   └── result-dir-H.dir
    │       ├── graph-H.init
    │       └── graph-H.out
    └── m-result.dir
        ├── m-result-dir-1.dir
        │   ├── graph-1.init
        │   └── graph-1.out
        ├── ....
        └── m-result-dir-W.dir
            ├── graph-W.init
            └── graph-W.out
```

3.2 Componenti utilizzati e progettazione

I programmi generator.py e reader.py sono stati interamente scritti utilizzando il linguaggio Python [versione 3.6.5].

Il codice è stato scritto utilizzando differenti editor di testo (vim [neovim], spacemacs, sublime-text, atom,...) e testato su diverse macchine GNU/Linux e Windows, in particolare su differenti shell (bash, zsh, fish) e su cmd.

Per agevolare il processo di progettazione e scrittura del codice l'intera struttura del progetto è stata caricata in un repository all'interno del sito github e gestita in remoto (attraverso il software git).

Per questioni di compatibilità e versatilità è stata utilizzata in modo massiccio la libreria standard relativa al linguaggio Python per effettuare la quasi totalità delle operazioni in entrambi i programmi.

La versione di riferimento della libreria è quella associata alla versione del linguaggio utilizzato, dunque la 3.6.5.

Al fine di trascurare alcuni aspetti relativi alla strutturazione e costruzione dei grafi è stata utilizzata una potente libreria per la creazione e la manipolazione di questi oggetti matematici, ovvero NetworkX.

Tale scelta di progettazione ha permesso al sottoscritto di concentrarsi maggiormente sulla progettazione e sull'ottimizzazione degli algoritmi.

Inoltre tale scelta ha consentito al sottoscritto di rendere totalmente dinamici i processi di creazione e lettura dei grafi, ciò ha alleggerito di molto il carico di lavoro in fase di sperimentazione.

La libreria standard del linguaggio Python è stata utilizzata in particolare per rendere totalmente dinamica e cross-platform la gestione del filesystem.

Ciò è stato necessario per garantire il funzionamento asincrono del generatore e del lettore, in modo tale da facilitare il lavoro in fase di sperimentazione.

Le fasi di generazione e lettura dei grafi infatti sono state completamente separate al livello di utilizzo, per fare ciò è stato necessario manipolare efficientemente il filesystem in modo da salvare i grafi creati e i risultati delle sperimentazioni su file.

Tali operazioni sono perfettamente funzionanti sia su sistemi che rispettano lo standard POSIX (Sistemi Unix-like) per il filesystem sia per sistemi che non lo rispettano (Sistemi Windows), dunque l'intero progetto è totalmente cross-platform e può essere facilmente migrato rendendo la portabilità un importante fattore di forza di quest'ultimo.

Il formato principale manipolato dai programmi è il formato .edgelist che analizzeremo in seguito.

In fase di debug è stata utilizzata la libreria di disegno MatPlotLib per rappresentare i grafi, quest'ultima è integrata in modo nativo all'interno della libreria NetworkX, dunque tale scelta di utilizzo ha reso più facili le operazioni di analisi e debug.

Un'altra libreria che è stata utilizzata in fase di analisi e debug per rappresentare i grafi in ambiente GNU/Linux è la libreria di disegno PyGraphViz.
Anche quest'ultima è integrata in modo nativo all'interno della libreria NetworkX.

Il formato di descrizione testuale dei grafi .dot è stato utilizzato solo in fase di debug in ambiente GNU/Linux assieme alla libreria di disegno PyGraphViz, dunque possiamo tralasciare la sua definizione e descrizione dato che non viene utilizzato all'interno dei programmi durante l'esecuzione.

Altre librerie minori sono state scelte in fase di progettazione ed utilizzate all'interno dei programmi, ad esempio la libreria per la colorazione dell'output testuale su terminal emulators cross-platform Colorama è stata utilizzata in fase di analisi e debug per semplificare e rendere più chiara la lettura dell'output relativo all'esecuzione degli algoritmi.

Un altro esempio è l'utilizzo della libreria cross-platform Pick, che rende semplice ed efficace la selezione delle opzioni all'interno dei terminal emulators durante l'esecuzione dei programmi.

L'elenco completo delle librerie utilizzate all'interno dei programmi è il seguente.

- Python Standard Library (Python 3.6.5 - Python 2.7.14)
  https://docs.python.org/3/library/

- NetworkX Library (NetworkX 2.1)
  https://pypi.org/project/networkx/

- Matplotlib Library (Matplotlib 2.2)
  https://pypi.org/project/matplotlib/
- Pick Library (Pick 0.6.4)
  https://pypi.org/project/pick/
- Pydot Library (Pydot 1.2.4)
  https://pypi.org/project/pydot/
- Graphviz Library (Graphviz 0.8.2)
  https://pypi.org/project/graphviz/
- PyParsing Library (PyParsing 2.2.0)
  https://pypi.org/project/pyparsing/
- Colorama Library (Colorama 0.3.9)
  https://pypi.org/project/colorama/

Per il funzionamento completo dei programmi è necessaria l'installazione di Python e dei suddetti componenti attraverso l'uso del modulo pip e/o l'uso di package manager (apt, pacman, ...).

Procediamo ora nella trattazione descrivendo il funzionamento dei programmi generator.py e reader.py.
Saranno poi analizzati in modo approfondito gli algoritmi per il calcolo degli equilibri di Nash e degli ottimi.

3.3 generator.py : il generatore di grafi

Il programma generator.py è un generatore dinamico di grafi scritto interamente in Python, in grado di semplificare le operazioni di creazione e manipolazione di questi oggetti matematici e in grado di creare automaticamente tutte le strutture di filesystem necessarie al salvataggio su file dei grafi generati.

Il programma, come specificato in precedenza, è completamente cross-platform e, per ciò che riguarda la gestione del filesystem, è stato accuratamente ottimizzato per non generare conflitti e problemi di inconsistenza dei dati.

Principalmente il programma utilizza due moduli principali necessari al corretto funzionamento dello stesso, la Standard Library del linguaggio Python per le funzioni di base e la libreria NetworkX per la creazione e manipolazione dei grafi.

Di seguito vengono riportate le caratteristiche del programma assieme ad un esempio generale di un caso d'uso.

Per prima cosa vengono impostati i motori di disegno, ovvero le librerie MatPlotLib e PyGraphViz.
Quest'ultima è stata utilizzata solo in ambiente GNU/Linux in fase di debug, dunque non è disponibile per l'utente.

In compenso è però disponibile per l'utente la libreria MatPlotLib che consente a quest'ultimo, qualora volesse, di disegnare al termine della generazione i grafi appena creati.
È di fondamentale importanza però specificare che il processo di disegno per grafi di grandi dimensioni è molto dispendioso e dunque può richiedere un tempo considerevole.

Come prima operazione, all'avvio del programma, è disponibile per l'utente una scelta della classe di grafo da generare effettuata attraverso un'interfaccia di selezione da console implementata grazie alla libreria Pick.

Sono disponibili per l'utente 2 modalità di funzionamento per la generazione, la modalità SINGLE MODE (la modalità di default del programma) e la modalità MULTIPLE MODE (accessibile attraverso la selezione dell'opzione MULTIPLE nella scelta della classe).

3.3.1 generator.py : SINGLE MODE

Le classi implementate fanno riferimento a quelle presenti nella libreria NetworkX, ovvero le seguenti (se ne citano solo alcune).

- Classic
- Expanders
- Small
- Random graphs
- Duplication divergence
- ...
- MULTIPLE

Una volta selezionata la classe desiderata, l'utente si troverà davanti una nuova interfaccia di selezione da console implementata attraverso l'uso del modulo Pick.

La selezione dell'opzione MULTIPLE cambia la tipologia di funzionamento del programma, che da SINGLE MODE (modalità di generazione di un singolo grafo per volta) passa a MULTIPLE MODE (nella quale possono essere generati da 1 a n grafi della stessa tipologia in un solo processo di creazione, con n scelto dall'utente).
Concentriamoci ora sulla SINGLE MODE, ovvero la la modalità di default del programma.

Questa volta l'utente dovrà scegliere la tipologia di grafo da creare.
Per ogni classe vi sono molteplici tipologie di grafo che l'utente può scegliere di generare.

Le tipologie implementate fanno riferimento a quelle presenti nel modulo NetworkX, in particolare ciascuna tipologia ha un costruttore di libreria corrispondente che provvede a generare le strutture elementari e a comporre l'oggetto matematico richiesto dall'utente.

Solo per far comprendere al lettore la vastità delle scelte possibili per l'utente in fase di selezione del grafo da generare, vengono qui si seguito riportate le tipologie di grafi implementate per la sola classe di grafi Classic.

- balanced tree
- complete graph
- circular ladder graph
- cycle graph
- dorogovtsev goltsev mendes graph
- ladder graph
- lollipop graph
- path graph
- star graph
- turan graph
- wheel graph
- ...

Una volta selezionata la tipologia di grafo da implementare, l'utente potrà inserire i parametri di creazione per quest'ultimo.
I parametri associati ai grafi variano da tipologia a tipologia e sono necessari e fondamentali per la corretta generazione del grafo scelto.

Per una migliore comprensione, ad esempio, i parametri associati alla tipologia balanced_tree sono il branching_factor e l'height dell'albero, invece per la tipologia di grafo complete_graph vi è un unico parametro da passare al programma, il node_number del grafo.
L'esempio ovviamente ricopre solo 2 tipologie di grafo, ma ciò vale per ogni tipologia implementata nel programma.

A questo punto l'utente è chiamato a scegliere quale tipologia di dato utilizzare per codificare i pesi associati agli archi del grafo, peso flottante (tipo float) o peso intero (tipo int).
Possiamo trascurare il tipo flottante, dato che quest'ultimo, non essendo interessante e significativo per la sperimentazione, è stato tralasciato in favore del tipo intero.

L'utente dunque dovrà scegliere i valori massimo e minimo relativi al range all'interno del quale oscilleranno randomicamente i pesi interi associati agli archi del grafo.
È possibile scegliere solo pesi interi positivi, ovvero, i valori minimo e massimo del range, devono essere compresi tra $0 \geq minimo \geq massimo \geq n$, con $n \to \infty$.

L'utente inoltre dovrà inserire il nome del grafo da creare che sarà utilizzato dal programma per costruire la sotto porzione di filesystem relativa al grafo generato, ovvero nome-scelto.dir/nome-scelto.edgelist, in modo da non creare conflitti tra i dati.

A questo punto il programma procederà a scrivere sul filesystem la struttura cartella/file.edgelist relativa al grafo appena creato.

Il file, nel quale è codificato il grafo generato, ha estensione .edgelist (il file .dot, come specificato sopra, è stato utilizzato solo in fase di debug e quindi non viene considerato nella trattazione).

Il file .edgelist è un file di tipo testuale che codifica sotto forma di lista (nodo, nodo, peso) la matrice di adiacenza che descrive il grafo generato.

I nodi sono generati in modo totalmente dinamico utilizzando valori interi positivi da 0 a n, con n scelto dall'utente in fase di creazione quando richiesto come parametro associato alla tipologia di grafo da generare.

Il file si presenta nella forma qui di seguito indicata (ciascuna coppia (nodo, nodo) codifica un arco al quale viene associato un perso attraverso la definizione del terzo valore della tupla, ovvero il peso intero).

nodo	**nodo**	**peso**
0	1	9
0	2	20
1	2	15
2	3	3
...	...	...

Per completezza specifichiamo che la libreria offre i seguenti 4 tipi di grafi.

- Graph (grafo senza parallelismo non-orientato)
- Directed Graph (grafo senza parallelismo orientato)
- MultiGraph (grafo con parallelismo non-orientato)
- Directed MultiGraph (grafo con parallelismo orientato)

Nel programma è stata implementata solo la tipologia Graph, cioè grafi senza parallelismo non-orientati (con pesi interi positivi associati agli archi e valori interi positivi associati ai nodi) come richiesto dal problema modellato in precedenza, ovvero il gioco della k-colorazione generalizzata.

Il programma termina a questo punto l'esecuzione permettendo all'utente di scegliere se disegnare il grafo appena creato attraverso la libreria MatPlotLib, l'utente può saltare l'esecuzione di questa operazione rispondendo negativamente alla richiesta.

3.3.2 generator.py : MULTIPLE MODE

Il funzionamento della modalità MULTIPLE MODE, accessibile selezionando l'opzione MULTIPLE durante la scelta della classe, è quasi del tutto analogo a quello descritto in precedenza, le uniche differenze sono le seguenti.

Dopo aver scelto la tipologia di grafo da generare, all'utente verrà chiesto di inserire, come parametro aggiuntivo, il numero di iterazioni relativo all'algoritmo di generazione multipla.
In sostanza l'utente dovrà scegliere quanti grafi generare (appartenenti alla tipologia scelta in precedenza).

In seguito l'utente dovrà inserire un ulteriore parametro aggiuntivo, ovvero il valore minimo e massimo relativi al range all'interno del quale dovrà oscillare randomicamente il numero di nodi dei grafi da generare.
I valori minimo e massimo dovranno ovviamente essere compresi tra $0 \geq minimo \geq massimo \geq n$, con $n \to \infty$, come specificato in precedenza.

All'utente verrà inoltre chiesto di inserire il nome della cartella nella quale salvare gli output delle generazioni multiple, come nel caso descritto nella sezione SINGLE MODE.

A questo punto l'algoritmo di generazione provvederà a creare n grafi (con n scelto dall'utente, come descritto poco sopra) appartenenti alla tipologia scelta (ad esempio complete graph).
Ciascun grafo creato avrà un valore randomico di nodi compreso tra $0 \geq minimo \geq numero\_di\_nodi \geq massimo \geq n$.

Il programma provvederà automaticamente e in modo totalmente dinamico a creare le strutture nel filesystem necessarie a salvare correttamente i file .edgelist relativi a ciascun grafo creato durante la generazione multipla.
È stato implementato un processo di assegnamento automatico del nome per ciascun output generato che utilizza il round di iterazione assieme alla tipologia e al numero di nodi del grafo creato, così da evitare conflitti, inconsistenza e perdita dei dati.

Passiamo ora a descrivere il programma reader.py, il lettore di grafi, nel quale sono includi gli algoritmi per il calcolo degli equilibri di Nash e per il calcolo degli ottimi utilizzando le 2 funzioni di benessere sociale utilitario e egalitario, che rappresentano il vero cuore dell'implementazione.

3.4 reader.py : il lettore di grafi

il programma reader.py è un lettore dinamico di grafi anche esso scritto interamente in linguaggio Python che consente la lettura e la manipolazione dei grafi creati in modo asincrono attraverso l'utilizzo del generatore generator.py.

Inoltre il programma consente la modellazione del problema del gioco della k-colorazione generalizzata attraverso un massiccio uso della Standard Library del linguaggio Python.

Nel programma reader.py sono incluse le definizioni degli algoritmi per il calcolo degli equilibri di Nash e quelli per il calcolo dell'ottimo con funzione di benessere utilitario e egalitario.
Il programma permette la lettura di un singolo o di molteplici grafi per volta e consente l'esecuzione dei suddetti algoritmi sulle istanze lette.

Per rendere meno faticoso il lavoro relativo alla sperimentazione il programma è in grado di raccogliere efficacemente e organizzare i dati derivanti dalle esecuzioni in tempo reale.

In modo totalmente dinamico il lettore è in grado di salvare i risultati delle sperimentazioni manipolando le strutture relative al filesystem.
Come per il generatore, il programma utilizza principalmente e in modo quasi esclusivo la libreria standard del linguaggio Python per eseguire queste operazioni.
Anche in questo caso le primitive e le funzioni utilizzate garantiscono il funzionamento cross-platform e dunque la portabilità del programma.
Il tutto è stato ampiamente testato su macchine con sistema POSIX e non, il programma è totalmente ottimizzato e funzionale in questo senso.

I file di output per ciascuna esecuzione sono 2, il file .init (file testuale che contiene il salvataggio delle informazioni basilari del grafo, creato dopo aver effettuato la modellazione del problema) e il file.out (file testuale che contiene i risultati finali derivanti dall'esecuzione di uno o più algoritmi).

Sia per le esecuzioni singole che per quelle multiple, il risultato prodotto sarà sempre e comunque una coppia file.init / file.out.

Anche in questo caso sono disponibili per l'utente moduli per il disegno dei grafi, in particolare pre e post modellazione e per rappresentare i risultati delle esecuzioni (ad esempio la colorazione iniziale e la colorazione stabile ottenute dal calcolo degli equilibri di Nash).
La principale libreria in questo senso è MatPlotLib (la libreria PyGraphViz è stata utilizzata solo in fase di debug come specificato in precedenza).

Per non tediare il lettore confermiamo che, anche il programma reader.py, fa uso delle medesime librerie addizionali utilizzate nel programma generator.py, il loro uso è però marginale e dunque non verranno trattate, come in precedenza.
L'utilizzo più intensivo è relativo alle librerie Pick, per fornire un'interfaccia

di selezione da console per l'utente, e alla libreria Colorama, utilizzata in fase di lettura e debug per colorare l'output su console e mettere in risalto i dati fondamentali relativi alle esecuzioni degli algoritmi e alle operazioni essenziali di gestione del programma.

Di fondamentale importanza è l'utilizzo della libreria NetworkX che consente la lettura e l'interpretazione delle informazioni relative ai file testuali di creazione .edgelist e la ricostruzione in modo asincrono dei grafi generati in precedenza, in modo da utilizzarli per la sperimentazione.

Anche in questo caso, qui di seguito, vengono riportate le caratteristiche del programma assieme ad un esempio generale di un caso d'uso in modo da descrivere il funzionamento di quest'ultimo e le operazioni disponibili per l'utente

Sono disponibili per l'utente, in modo speculare e simmetrico rispetto alla definizione del programma generator.py, 2 modalità di funzionamento per la lettura e l'esecuzione : la modalità SINGLE EXEC e la modalità MULTIPLE EXEC.
All'avvio del programma l'utente dovrà scegliere una delle 2 modalità d'uso attraverso un'interfaccia di selezione da console implementata attraverso l'utilizzo della libreria Pick.

3.4.1 reader.py : SINGLE EXEC

In modalità SINGLE EXEC il programma effettuerà una ricerca globale automatica e dinamica dei file con estensione .edgelist all'interno della cartella gen, la cartella relativa alle generazioni di grafi singoli creata dal generatore generator.py in modalità SINGLE MODE.

A questo punto l'utente dovrà selezione il grafo da modellare e sul quale eseguire una sperimentazione, utilizzando gli algoritmi forniti all'interno del programma.
La scelta è implementata attraverso un'interfaccia di selezione creata con la libreria Pick nella quale apparirà la lista di file .edgelist trovati.

Il programma leggerà e interpreterà il contenuto del file.edgelist relativo al grafo selezionato dall'utente e attraverso le funzioni della libreria NetworkX ricostruirà l'istanza dell'oggetto matematico desiderato creato in precedenza e sul quale si vuole iniziare una sperimentazione.

Ovviamente l'operazione genererà sempre e comunque un'istanza di grafo senza parallelismo non-orientato e con valori interi positivi associati ai nodi e pesi interi positivi associati agli archi (parametri scelti in precedenza, durante la fase di creazione del grafo e ricostruiti dalle funzioni di libreria).

In seguito l'utente sarà nuovamente chiamato a scegliere 2 parametri fondamentali per la modellazione del problema, ovvero il numero di colori e valore massimo per il profitto associato a questi ultimi.

L'utente potrà scegliere il numero massimo di colori presenti all'interno dell'istanza che si sta modellando.
Il numero massimo di colori non può essere superiore al numero di nodi del grafo, altrimenti la sperimentazione perderebbe di senso e in più non verrebbe rispettata la definizione del problema presentata nel Capitolo 2.
I colori, come ogni altro parametro del quale non è richiesto un range (valore minimo e massimo), saranno numerati da 0 a n (valori interi positivi), con n scelto dall'utente e sempre $n \leq numero\_di\_nodi\_del\_grafo$.

Come struttura dati, è stata scelta la lista poiché l'elenco dei colori non necessità di essere manipolato in maniera complessa, non vi è inoltre l'esigenza di effettuare query di ricerca indicizzate e altre operazioni poco efficaci sulle liste.
Semplicemente la lista dei colori andrà iterata più volte e in modo innestato durante i vari cicli degli algoritmi implementati.
Per la creazione è stata utilizzata la potente list comprehension interna al linguaggio Python.

L'utente inoltre potrà scegliere il valore massimo all'interno del quale oscilleranno randomicamente i profitti associati ai colori per ciascun giocatore.
In questo caso non vi sono restrizioni, scelto n, i valori oscilleranno in modo randomico tra 0 e n (valori interi positivi).
I profitti legati ai colori, per definizione del modello, potranno essere differenti da giocatore a giocatore, ad esempio il giocatore 0 avrà un profitto di 56 per il colore 0, il giocatore 1 avrà un profitto di 29 per il colore 0 (i valori sono totalmente casuali e servono solo da esempio).

Proprio per questo motivo, la struttura dati implementata per soddisfare le caratteristiche del modello, è un doppio dizionario innestato.
Assieme alla struttura relativa alle coppie arco-peso e quella relativa all'associazione nodo-colore, che presenteremo in seguito, quest'ultima rappresenta uno dei componenti più utilizzati del programma.
È necessaria dunque, in questo caso, un'implementazione che permetta l'utilizzo di query di ricerca asincrone a doppia chiave (l'operazione più pesante che interessa questa struttura).
È stata utilizzata a questo scopo la potente nested dictionary comprehension del linguaggio Python, in modo da eliminare la possibilità di conflitto

e inconsistenza dei dati, in modo da disambiguare efficacemente le entry del dizionario e dunque in modo da garantire l'utilizzo di interrogazioni ad accesso diretto utili nel programma.
È stata generata la seguente struttura (esempio).

nodo 0 ::	colore 0 :	profitto 24,	colore 1 :	profitto 38,	...
nodo 1 ::	colore 0 :	profitto 11,	colore 1 :	profitto 55,	...
nodo 2 ::	colore 0 :	profitto 14,	colore 1 :	profitto 65,	...
nodo 3 ::	colore 0 :	profitto 99,	colore 1 :	profitto 66,	...
...	...	...	...	...	...

A questo punto viene generata automaticamente e in modo totalmente randomico la colorazione iniziale per il grafo corrente.
Il colore è stato gestito attraverso l'implementazione di dizionari, ad ogni nodo viene associata una singola entry chiave : valore che rappresenta una label.
La chiave rappresenta il nome del parametro, ovvero la stringa "color" e il valore rappresenta il colore con il quale è colorato in nodo.
Le etichette sono gestite a livello di libreria NetworkX, di conseguenza è stato utilizzato lo standard esplicato all'interno della documentazione.
È dunque possibile accedere e manipolare, con facilità e alta velocità computazionale, i dati all'interno dei dizionari che rappresentano i parametri associati al nodo attraverso semplici query ad accesso diretto e a singola chiave, ovvero, nel nostro caso, il parametro "color".
Un esempio di colorazione iniziale è presentata qui di seguito.

nodo 0	colore 2
nodo 1	colore 1
nodo 2	colore 1
nodo 3	colore 0
nodo 4	colore 2
...	...

Tutti i dati relativi alla modellazione iniziale dell'istanza letta per ricreare le condizioni di partenza del nostro gioco sono salvate nel file.init correlato all'esecuzione corrente.

Specifichiamo nuovamente l'assoluta gestione cross-platform del filesystem e l'efficiente ottimizzazione eseguita sul lato input / output, il tutto è affiancato ovviamente da numerosissimi test su diverse tipologie di macchine.

Nel file testuale .init, relativo all'esecuzione, vengono inseriti i seguenti parametri.

- lista dei colori
- colorazione iniziale del grafo
- numero di nodi del grafo
- profitti associati ai colori per ogni giocatore
- lista degli archi del grafo con pesi associati
- numero degli archi del grafo

È importante specificare che il peso degli archi è gestito in modo del tutto identico a quello relativo al colore dei nodi.
Infatti il peso degli archi (e gli eventuali altri parametri implementabili in modo nativo grazie alla libreria NetworkX) si presenta come un dizionario chiave : valore, nel quale la chiave corrisponde al nome del parametro, ovvero la stringa "weight" e il valore rappresenta il vero e proprio peso associato all'arco.

Anche qui dunque una struttura dati di questo tipo permette un'alta velocità di calcolo e recupero delle informazioni con query ad accesso diretto a chiave singola, ovvero il nome del parametro richiesto associato all'arco che nel nostro caso è la stringa "weight".
La struttura dati si presenta come segue (esempio).

nodo 0	nodo 1	weight : 34
nodo 1	nodo 2	weight : 55
nodo 2	nodo 5	weight : 5
nodo 3	nodo 2	weight : 23
...	...	...

Terminata la creazione del file testuale con estensione .init, il programma procede e genera tutte le strutture del filesystem necessarie per salvare correttamente i risultati della sperimentazione corrente.
In particolare crea una nuova cartella all'interno della directory result (poiché siamo in SINGLE EXEC) in modo dinamico prendendo il nome del grafo corrente importato dall'utente.

Assegna il medesimo nome al file .init (e al file .out che vedremo tra poco) che viene creato all'interno del path appena generato.

A questo punto l'utente viene chiamato a compiere un'ennesima scelta, l'ultima relativa al flusso di esecuzione corrente.
All'utente viene chiesto di scegliere tra le 3 opzioni di calcolo disponibili, ciascuna corrispondete a 1 dei 3 algoritmi implementati.

- calcolo della colorazione ottima (ottimo con funzione di benessere sociale utilitaria)
- calcolo della colorazione ottima (ottimo con funzione di benessere sociale egalitaria)
- calcolo della colorazione stabile (equilibrio di Nash)
- esecuzione completa
- esci

Vi inoltre un'ulteriore opzione che consente all'utente di uscire dal programma prima di un'eventuale esecuzione che potrebbe essere pesante a livello di calcolo e dunque lenta al livello temporale, soprattutto per ciò che concerne il calcolo degli ottimi.

Per l'utente che esegue in modalità SINGLE EXEC, questo punto del programma rappresenta un hub di esecuzione.
L'utente infatti può eseguire, quante volte vuole, i vari algoritmi (può anche eseguirli tutti sull'istanza corrente o uscire e non fare nulla come specificato poco sopra).
Terminata l'esecuzione di uno degli algoritmi, all'utente verrà lasciato il pieno controllo.
Quest'ultimo potrà leggere il risultato dell'esecuzione da console, analizzare i vari cicli eseguiti e poi tornare all'hub nel quale potrà avviare una nuova esecuzione o uscire dal programma.

Vi è inoltre l'opzione "esecuzione completa" che permette di calcolare l'ottimo con funzione di benessere sociale utilitaria, l'ottimo con funzione di benessere sociale egalitario e infine la colorazione stabile, nell'ordine specificato.
Il tutto sarà eseguito in modo automatico e slegato dunque dal controllo dell'utente, il quale dovrà fornire solo i classici parametri per la modellazione dell'istanza corrente come il numero di colori o il massimo valore del range relativo ai profitti.

Prima di affrontare la descrizione degli algoritmi, il vero cuore dell'implementazione, analizziamo al modalità di funzionamento MULTIPLE EXEC relativa al programma reader.py.

3.4.2 reader.py : MULTIPLE EXEC

Per non tediare il lettore, possiamo specificare che la modalità MULTIPLE EXEC esegue le medesime operazioni di preparazione e modellazione delle varie istanze e salva il risultato di tale operazione in un file testuale estensione .init.

Una prima differenza sostanziale è relativa al path in cui vengono cercati i file .edgelist nella fase di scelta della modalità di esecuzione del lettore.
Scegliendo MULTIPLE EXEC il programma effettuerà una ricerca globale di tutte le cartelle presenti all'interno della cartella mgen, ovvero la cartella contenente tutti i risultati delle varie generazioni multiple, creata dinamicamente dal generatore di grafi in modalità MULTIPLE MODE.

A questo punto l'utente sarà nuovamente interpellato e attraverso un'interfaccia di selezione da console dovrà selezionare la cartella contenente i grafi sui quali desidera fare sperimentazione.
Una volta selezionata la cartella il programma utilizzerà tutti i file con estensione .edgelist come istanze dell'esecuzione multipla.
In particolare il programma si sviluppa all'interno di un ciclo principale che scorre tutti i file con estensione .edgelist presenti nella cartella scelta dall'utente.

Il ciclo prende un grafo alla volta, opera la fase di pre-esecuzione modellando l'istanza corrente secondo la definizione del problema e poi esegue l'algoritmo per il calcolo della colorazione stabile trovando l'equilibrio di Nash per l'istanza corrente.
Il ciclo e l'esecuzione, che potrebbe coinvolgere un numero molto grande di grafi, avvengono in maniera totalmente automatica e dunque l'utente può disinteressarsi del programma poiché slegato dal flusso di esecuzione.

Proprio a causa del fatto che le istanze potrebbero essere numerose e che la grandezza di ciascuna istanza, in termini di numero di nodi, potrebbe essere elevata, si utilizza come sperimentazione, annessa alla modalità di esecuzione multipla, il solo algoritmo per il calcolo della colorazione stabile (equilibrio di Nash) che vedremo in seguito.

L'utente in questa fase che precede l'esecuzione, solo per la prima iterazione, il valore massimo del profitto associato ai colori per ogni giocatore.

Per ciò che riguarda il profitto, possiamo affermare che il funzionamento è identico a quello della modalità SINGLE EXEC, ovvero l'utente fornirà un numero n e il profitto da associare a ciascun colore per ciascun giocatore sarà un valore randomico tra 0 e n (valori interi positivi).

Per ciò che riguarda il numero dei colori, in modalità MULTIPLE EXEC, il funzionamento è in parte differente.
Il numero di colori, infatti, viene definito in modo randomico e dinamico durante l'esecuzione su ciascuna istanza in input.
Quest'ultimo sarà un valore intero positivo k compreso tra $0 \geq k \geq n$, con $n = numero\_di\_nodi\_dell'istanza\_corrente$, assegnato randomicamente dal programma.

A questo punto il programma inizierà l'esecuzione dell'algoritmo per il calcolo della colorazione stabile per ciascun grafo appartenente all'istanza corrente di esecuzione in modalità MULTIPLE EXEC.
La modellazione e le informazioni fondamentali di ciascun grafo saranno salvate in modo congiunto all'interno di un unico file con estensione .init, accuratamente formattate.
Lo stesso vale per gli output delle varie esecuzioni, anche in questo caso il file, in formato .out, sarà unico.

3.5 Analisi e descrizione degli algoritmi

Procediamo ora a descrivere i vari algoritmi implementati e ad analizzare approfonditamente le iterazioni e le operazioni di manipolazione effettuate, in modo da definire e delineare al meglio, anche con l'utilizzo di pseudocodice annesso alla trattazione, il funzionamento di questi ultimi.

Gli algoritmi implementati sono i seguenti.

- L'algoritmo per il calcolo di una colorazione stabile (equilibrio di Nash) con annesse le funzioni di calcolo del benessere sociale utilitario ed egalitario per la colorazione finale stabile trovata.
- L'algoritmo per il calcolo della colorazione ottima utilizzando la funzione di benessere sociale utilitario
- L'algoritmo per il calcolo della colorazione ottima utilizzando la funzione di benessere sociale egalitario

3.5.1 nash_equilibrium : l'algoritmo per il calcolo degli equilibri di Nash

Gli algoritmi analizzati presentano la definizione della variabile *limit*.
Quest'ultima contiene il valore del limitatore temporale (in secondi) associato all'esecuzione corrente.
Viene impostato in modo dinamico all'inizio dell'esecuzione ed è valido per ogni tipo di calcolo.

Il seguente algoritmo viene usato per calcolare una colorazione stabile per il gioco della k-colorazione generalizzata modellato in precedenza, utilizzando la definizione di equilibrio di Nash.

Come abbiamo visto esiste sempre almeno un equilibrio di Nash per ogni istanza di gioco della k-colorazione generalizzata e il problema di trovare una di queste colorazioni stabili è PLS-Completo.

Presentiamo ora lo pseudocodice relativo al suddetto algoritmo e di seguito analizziamo in modo accurato le caratteristiche, le iterazioni e le operazioni eseguite nell'implementazione raggiunta.

Si specifica che il seguente algoritmo è stato scritto utilizzando la Standard Library del Python e la libreria di costruzione e manipolazione di grafi NetworkX e che l'implementazione è stata accuratamente ottimizzata nei cicli e nelle operazioni grazie ad un attento uso di funzioni e strutture dati e dunque la computazione è particolarmente snella è rapida.

```
start ← time.time()
limit ← X                                   ▷ X assegnato dinamicamente dall'utente
time_limit ← False

count ← 0
restart ← True
last_improved_node ← None

while restart do
    if time.time() > start + limit then
        time_limit ← True
        break
    end if
    restart ← False
    for (node, data) ∈ G.nodes(data = True) do
        color_init ← data['color']
        color_best ← data['color']
        profit_old ← profits[node][data['color']]
        if node = last_improved_node then
            continue
        end if
        neighbors ← G.neighbors(node)
        for neighbor ∈ neighbors do
            if G.node[neighbor]['color'] ≠ G.node[node]['color'] then
                edge_weight ← G[node][neighbor]['weight']
                profit_old+ = edge_weight
            else
                continue
            end if
        end for
        for current_color ∈ colors do
            if current_color ≠ color_init then
                data['color'] ← current_color
                color_new ← current_color
                profit_new ← profits[node][current_color]
                neighbors ← G.neighbors(node)
                for neighbor ∈ neighbors do
                    if G.node[neighbor]['color'] ≠ G.node[node]['color'] then
                        edge_weight ← G[node][neighbor]['weight']
                        profit_new+ = edge_weight
                    else
                        continue
                    end if
                    if profit_new > profit_old then
```

$profit\_old \leftarrow profit\_new$
$color\_best \leftarrow current\_color$
else
continue
end if
end for
else
continue
end if
end for
$data['color'] \leftarrow color\_best$
if $data['color'] \neq color\_init$ **then**
$count+ = 1$
$restart \leftarrow True$
$last\_improved\_node \leftarrow node$
break
end if
end for
end while

if $time\_limit == False$ **then**
$egalitarian\_social\_welfare \leftarrow 0$
$utilitarian\_social\_welfare \leftarrow 0$
$first\_iter\_check \leftarrow True$

for $(node, data) \in G.nodes(data = True)$ **do**
$temp\_egalitarian\_social\_welfare \leftarrow 0$
$temp\_egalitarian\_social\_welfare+ = profits[node][data['color']]$
$utilitarian\_social\_welfare+ = profits[node][data['color']]$
$neighbors \leftarrow G.neighbors(node)$
for $neighbor \in neighbors$ **do**
if $G.node[neighbor]['color'] \neq G.node[node]['color']$ **then**
$edge\_weight \leftarrow G[node][neighbor]['weight']$
$utilitarian\_social\_welfare+ = edge\_weight$
$temp\_egalitarian\_social\_welfare+ = edge\_weight$
else
continue
end if
if $first\_iter\_check$ **then**
$egalitarian\_social\_welfare \leftarrow temp\_egalitarian\_social\_welfare$
$first\_iter\_check \leftarrow False$
continue
end if
if $temp\_egalitarian\_social\_welfare < egalitarian\_social\_welfare$

then

$egalitarian\_social\_welfare \leftarrow temp\_egalitarian\_social\_welfare$

else

continue

end if

end for

end for

$nash\_utilitarian\_social\_welfare \leftarrow utilitarian\_social\_welfare$

$nash\_egalitarian\_social\_welfare \leftarrow egalitarian\_social\_welfare$

if $check\_utilitarian\_social\_welfare$ **then**

$utilitarian\_price\_of\_anarchy \leftarrow$
$opt\_utilitarian\_social\_welfare/nash\_utilitarian\_social\_welfare$

end if

if $check\_egalitarian\_social\_welfare$ **then**

$egalitarian\_price\_of\_anarchy \leftarrow$
$opt\_egalitarian\_social\_welfare/nash\_egalitarian\_social\_welfare$

end if

end if

La sintassi dello pseudocodice, in generale nell'aspetto e in particolare in alcuni punti nei quali trascende l'astrazione, è volutamente molto vicina a quella del linguaggio Python in modo tale da coinvolgere il lettore nell'implementazione e in modo da mostrare alcune soluzioni adottate per effettuare le operazioni più significative.

In ogni caso le parti più vicine all'implementazione in linguaggio Python saranno abbondantemente spiegate e rese chiare qui di seguito.

Sono state omesse nello pseudocodice tutte le porzioni di stampa su console e su file.out dei dati significativi, poiché trascurabili.

Il seguente algoritmo viene utilizzato, seguendo la definizione presentata poco sopra, solo nella modalità SINGLE EXEC, in modalità MULTIPLE EXEC non viene considerata l'ultima porzione del codice, ovvero quella relativa al prezzo dell'anarchia sperimentale utilitario e egalitario poiché, data la grandezza dei grafi utilizzati in questo tipo di sperimentazione, vengono tralasciati i calcoli dell'ottimo utilitario e egalitario.

Il prezzo dell'anarchia sperimentale rappresenta lo studio e l'analisi del prezzo dell'anarchia (benessere sociale utilitario - egalitario della colorazione ottima / benessere sociale utilitario - egalitario della colorazione stabile) ma nel caso medio, laddove il prezzo dell'anarchia rappresenta lo studio e l'analisi nel caso peggiore.
Il prezzo dell'anarchia sperimentale è uno dei risultati fondamentali prodotti dalla sperimentazione.

L'algoritmo presenta 3 importanti porzioni di ottimizzazione dei cicli che verranno presentate qui di seguito, grazie alle quali è possibile snellire la fase di calcolo e diminuire l'impiego di risorse e il tempo di esecuzione.

Come prima operazione impostiamo i valori temporali.
In particolare viene memorizzato il tempo corrente relativo all'inizio dell'esecuzione all'interno della variabile *start*, utilizzando la libreria standard del linguaggio Python per effettuare l'operazione.
In seguito viene impostato il limite temporale nella variabile *limit*, assegnato dinamicamente dall'utente all'inizio del programma.
Oltrepassato il limite temporale interrompiamo l'esecuzione e non restituiamo alcun risultato, poiché quest'ultimo non è stato trovato dall'algoritmo nel tempo prestabilito.
Tale controllo viene effettuato inizializzando al valore *False* la variabile *time_limit* ed effettuando ad ogni iterazione il controllo sulla validità del tempo complessivo trascorso, lo analizzeremo tra poco.

All'inizio del programma la variabile *count* viene inizializzata con il valore 0.
Quest'ultima serve a contare il numero di miglioramenti effettuati durante la computazione, in particolare registra il numero di mosse migliorative effettuate dalla procedura.
La variabile è restituita alla fine dell'algoritmo (è stata omessa la porzione di codice relativa alla stampa su console e su file.out nello pseudocodice) e rappresenta il numero di passi (step) effettuati dall'algoritmo.
La variabile *count* è il parametro fondamentale della sperimentazione ed è inoltre il vero e proprio metro di giudizio fra le varie esecuzioni effettuate sulle differenti istanze.

La variabile *restart* viene inizializzata con il valore booleano $True$.
Quest'ultima serve, come vedremo tra poco, a reinizializzare l'intero ciclo ogni volta che una mossa migliorativa viene trovata dall'algoritmo.

Infine un'ulteriore variabile, *last_improved_node* viene dichiarata e inizializzata con il valore $None$ (valore nullo nel linguaggio Python).
Quest'ultima viene utilizzata in una delle 3 strategie di ottimizzazione utilizzate all'interno dell'algoritmo poiché conterrà il valore dell'ultimo nodo che ha effettuato una mossa migliorativa.

Il ciclo $while(restart)$ è il ciclo fondamentale e più esterno dell'algoritmo.
Come si evince dalla condizione, l'iterazione continua fin tanto che il valore della variabile *restart* è $True$.

Viene ora istanziato il controllo sulla validità del tempo complessivo trascorso, come specificato poco sopra.
In particolare se il tempo corrente $time.time() > start + limit$ allora sono trascorsi più di X secondi e l'esecuzione non ha ancora prodotto alcun risultato, dunque modifichiamo con il valore $True$ il contenuto della variabile *time_limit* e interrompiamo l'esecuzione corrente uscendo dal ciclo principale dell'algoritmo attraverso l'istruzione $break$.

A questo punto la variabile *restart* viene inizializzata al valore $False$.
Solo quando viene trovato un miglioramento quest'ultima viene inizializzata nuovamente al valore $True$ e dunque, se nessun miglioramento viene trovato dall'algoritmo, la condizione all'interno del ciclo *while* si falsifica è l'esecuzione termina.

L'istruzione successiva è un nuovo loop innestato.
Questa volta si tratta di un ciclo for che itera scorrendo, uno per volta, tutti i nodi del grafo.
È stata utilizzata una sintassi molto vicina al codice Python in questa se-

zione per evidenziare l'utilizzo della funzione $G.nodes(data = True)$.
Quest'ultima è una funzione appartenente alla libreria NetworkX che itera tutti i nodi del grafo G.
In particolare il parametro $data = True$ viene utilizzato per prendere in input all'interno dell'iterazione le etichette associate ai nodi.
Come visto in precedenza, le etichette sono dizionari di parametri che caratterizzano il nodo, dunque in questo modo possiamo catturare il contenuto della label $'color'$ che contiene il valore del colore con il quale è colorato il nodo corrente.

Assegniamo alle variabili $color\_init$ e $color\_best$ il colore iniziale del nodo corrente attraverso l'operazione di accesso diretto a chiave singola al dizionario dei parametri associati al nodo con la stringa di codice $data['color']$.
Esso rappresenta dunque il colore iniziale assegnato al nodo ($color\_init$) e inoltre rappresenta, in questo momento, il miglior colore trovato per il nodo corrente ($color\_best$).

Nella variabile $profit\_old$ viene ora salvato il valore del profitto legato al colore associato al nodo corrente attraverso l'operazione di accesso diretto a chiave doppia al dizionario $profits$ definita come $profit\_old \leftarrow profits[node][data['color']]$.
Il dizionario profits, come specificato nelle sezioni precedenti, è stato creato in precedenza utilizzando la nested dictionary comprehension, ovvero *profits = {node:{color:random.randint(0,maxp) for color in colors} for node in G.nodes()}*.

A questo punto troviamo il primo controllo che ottimizza l'esecuzione dell'algoritmo.
In particolare viene confrontato se il valore del nodo corrente *node* è uguale al valore dell'ultimo nodo migliorato dall'algoritmo $last\_improved\_node$.
Questo poiché non è possibile migliorare ulteriormente un nodo che ha appena effettuato una mossa migliorativa e ha indotto un nuovo stato del gioco, ovvero una nuova colorazione.
Se la condizione di uguaglianza relativa al blocco if che effettua il controllo viene soddisfatta, l'algoritmo non esegue operazioni sul nodo corrente e passa alla prossima iterazione analizzando un altro nodo attraverso l'istruzione $continue$.

Nella successiva istruzione è nuovamente preservata in parte la sintassi del linguaggio Python.
Nella variabile $neighbors$ viene salvata la lista (iterabile) dei nodi adiacenti al nodo corrente attraverso la funzione appartenente alla libreria NetworkX $G.neighbors(node)$.

A questo punto viene dichiarato un ulteriore ciclo for innestato che itera tutti i nodi (adiacenti al nodo corrente) contenuti nella lista neighbors appena creata con l'istruzione precedente.

Ora, come nella definizione del modello, controlliamo se il colore del nodo corrente è diverso dal colore del nodo adiacente.
Se la condizione viene soddisfatta salviamo nella variabile $edge\_weight$ il valore del peso associato all'arco che collega i 2 nodi in oggetto e aggiorniamo il valore della variabile $profit\_old$ sommandovi il contenuto della variabile $edge\_weight$.

Se la condizione non viene soddisfatta saltiamo l'iterazione corrente con l'istruzione *continue* e, se disponibile, analizziamo uno degli ulteriori nodi adiacenti al nodo corrente.

In questo momento dunque disponiamo del valore di utilità complessivo per il nodo in oggetto colorato con il colore corrente nella colorazione che rappresenta lo stato attuale del gioco.
Con i prossimi cicli innestati tentiamo di trovare un colore che garantisca un'utilità migliore per il nodo corrente.

In particolare iteriamo con un ciclo for tutti i colori presenti nella lista $colors$.
Ora utilizziamo la seconda strategia di ottimizzazione per snellire la computazione.
Il colore selezionato dall'iterazione corrente viene confrontato con il colore del nodo corrente, ovvero vi è un controllo che verifica se la disuguaglianza $current\_color \neq color\_init$ viene soddisfatta o meno

Se la disuguaglianza non viene soddisfatta, significa che il colore corrente è esattamente quello con cui il nodo in oggetto è colorato, passiamo dunque alla prossima iterazione senza eseguire alcuna operazione attraverso l'istruzione *continue*, ciò poiché non avrebbe senso ricalcolare l'utilità del nodo in oggetto per il colore corrente, dato che ciò è stato fatto nel ciclo precedente.

Se il colore iterato è diverso dal colore corrente del nodo in oggetto, coloriamo il nodo corrente con il nuovo colore inserendolo come nuovo valore associato alla chiave $'color'$, con l'istruzione $data['color'] \leftarrow current\_color$.
In seguito inizializziamo al variabile $color\_new$ con il valore del colore corrente contenuto in $current\_color$.
Inizializziamo inoltre la variabile $profit\_new$ con il profitto associato al nuovo colore con il quale è colorato ora il nodo corrente.

Aggiorniamo ora il valore della variabile $profit\_new$ eseguendo il medesimo ciclo presentato poco sopra, sommandovi il peso degli archi tra il nodo corrente e i suoi vicini se e solo se i colori tra le coppie di nodi analizzate sono differenti.

A questo punto eseguiamo un controllo sul valore dell'utilità trovata.
In particolare, se la nuova utilità trovata colorando, con un colore differente da quello iniziale, il nodo corrente è maggiore dell'utilità iniziale (if $profit\_new > profit\_old$), aggiorniamo il contenuto della variabile $profit\_old$ con il nuovo valore della variabile $profit\_new$ e modifichiamo il valore del miglior colore trovato $color\_best$ con quello del colore corrente $current\_color$.

Descriviamo ora un'ulteriore strategia di ottimizzazione implementata all'interno dell'algoritmo.
Come vedremo nelle prossime istruzioni, l'algoritmo, preso il nodo e il colore corrente iniziale, cerca di trovare il colore che garantisce la migliore utilità all'interno della colorazione attuale per il nodo in oggetto.
Ciò significa che l'algoritmo non si ferma al primo miglioramento trovato per il nodo corrente, ma cerca di trovare il miglioramento che garantisce la massima utilità possibile per quest'ultimo.
In parole povere, itera tutti i $k-1$-colori restanti e trova il colore migliore per il nodo corrente, ovvero quello che garantisce il miglior profitto possibile per quest'ultimo all'interno della colorazione attuale.

Alla fine del precedente ciclo innestato infatti, l'algoritmo assegna al nodo corrente il miglior colore trovato attraverso l'istruzione $data['color'] \leftarrow color\_best$.
Se quest'ultimo è differente dal colore iniziale possiamo affermare che è disponibile un miglioramento per il nodo corrente.
Effettuiamo tale miglioramento e aggiorniamo i valori : $count$ (aggiungendo +1, poiché l'algoritmo ha eseguito uno step), $restart$ (assegnandole il valore booleano $True$, in modo da garantire la reinizializzazione del ciclo più esterno) e $last\_improved\_node$ (con il valore del nodo corrente, il quale, essendo stato appena migliorato, non potrà migliorare nuovamente all'iterazione successiva).
A questo punto l'iterazione corrente termina e il ciclo viene interrotto dall'uso dell'istruzione $break$.
La variabile $restart$, che ora contiene nuovamente il valore $True$ permette la reinizializzazione del ciclo poiché la condizione del $while$ più esterno viene soddisfatta.
Se non viene trovato alcun miglioramento, il valore della variabile $restart$ non viene reimpostato a $True$ e dunque, terminata l'iterazione su tutti i nodi del grafo, l'esecuzione corrente viene interrotta.

Se il calcolo dell'equilibrio viene portato a termine correttamente e nel tempo limite prestabilito ($if time\_limit == False$), viene eseguito un calcolo congiunto del benessere sociale utilitario e egalitario per la colorazione stabile trovata.

Vengono inizializzate al valore 0 le variabili $egalitarian\_social\_welfare$ e $utilitarian\_social\_welfare$.
Viene inoltre inizializzata al valore $True$ la variabile $first\_iter\_check$.

A questo punto scorriamo nuovamente tutti i nodi (con annessi i relativi parametri) del grafo corrente, utilizzando il medesimo ciclo for presentato in precedenza durante il calcolo dell'equilibrio.
Inizializziamo una variabile temporanea $temp\_egalitarian\_social\_welfare$ con il valore 0, quest'ultima servirà per il calcolo del benessere sociale egalitario relativo alla colorazione stabile trovata.

Modifichiamo il valore della variabile $temp\_egalitarian\_social\_welfare$ e quello della variabile $utilitarian\_social\_welfare$ con il valore del profitto relativo al colore associato al nodo corrente nell'attuale colorazione stabile.

Istanziamo nuovamente il loop, utilizzato in precedenza, che itera i nodi adiacenti per ciascun nodo del grafo in oggetto.

Salviamo nella variabile $edge\_weight$ il valore del peso dell'arco tra il nodo corrente e uno dei suoi nodi adiacenti se e solo se i colori con i quali sono colorati entrambi i nodi sono differenti, altrimenti non eseguiamo alcuna operazione e saltiamo l'iterazione corrente per passare alla successiva attraverso l'uso dell'istruzione $continue$.
In caso di colori differenti, aggiorniamo il valore delle variabili $utilitarian\_social\_welfare$ e $temp\_egalitarian\_social\_welfare$ sommandovi il contenuto della variabile $edge\_weight$.

Un ulteriore controllo è quello relativo al conteggio dell'iterazione corrente.
Se il valore della variabile booleana $first\_iter\_check$ è $True$, significa che ci troviamo all'interno della prima iterazione, dunque assegniamo semplicemente a $egalitarian\_social\_welfare$ il valore della variabile $temp\_egalitarian\_social\_welfare$, modifichiamo il contenuto della variabile $first\_iter\_check$ al valore $False$ e saltiamo all'iterazione successiva utilizzando l'istruzione $continue$.

Per le successive iterazioni (tutte tranne la prima), eseguiamo il seguente controllo.
Se il contenuto della variabile

$temp\_egalitarian\_social\_welfare < egalitarian\_social\_welfare$, abbiamo trovato un valore migliore per il benessere sociale egalitario relativo alla colorazione stabile.
Quindi assegniamo tale valore alla variabile $egalitarian\_social\_welfare$.

Al termine dell'iterazione più esterna le variabili
$nash\_egalitarian\_social\_welfare$ e $nash\_utilitarian\_social\_welfare$, grazie agli assegnamenti
$nash\_utilitarian\_social\_welfare \leftarrow utilitarian\_social\_welfare$ e
$nash\_egalitarian\_social\_welfare \leftarrow egalitarian\_social\_welfare$, conterranno rispettivamente i valori del benessere sociale egalitario e utilitario relativi alla colorazione stabile corrente.

La seguente procedura, relativa al calcolo del prezzo dell'anarchia utilitario sperimentale e al prezzo dell'anarchia egalitario sperimentale, è stata implementata solo all'interno della modalità SINGLE EXEC, come specificato in precedenza.

Verifichiamo a questo punto se sono stati calcolati, per l'istanza corrente, l'ottimo con funzione di benessere sociale utilitario e quello con funzione di benessere sociale egalitario.

Se la condizione $ifcheck\_utilitarian\_social\_welfare$ risulta valida, significa che abbiamo già eseguito sull'istanza corrente il calcolo dell'ottimo con funzione di benessere sociale utilitario.
La variabile $opt\_utilitarian\_social\_welfare$ conterrà il valore relativo al suddetto calcolo dell'ottimo.
Assegniamo dunque, in caso la condizione sia valida, il valore della divisione $opt\_utilitarian\_social\_welfare$ $nash\_utilitarian\_social\_welfare$ alla variabile $utilitarian\_price\_of\_anarchy$.

Se la condizione $ifcheck\_egalitarian\_social\_welfare$ risulta valida, significa che abbiamo già eseguito sull'istanza corrente il calcolo dell'ottimo con funzione di benessere sociale egalitario.
La variabile $opt\_egalitarian\_social\_welfare$ conterrà il valore relativo al suddetto calcolo dell'ottimo.
Assegniamo dunque, in caso la condizione sia valida, il valore della divisione $opt\_egalitarian\_social\_welfare$ $nash\_egalitarian\_social\_welfare$ alla variabile $egalitarian\_price\_of\_anarchy$.

3.5.2 opt_utilitarian_social_welfare : l'algoritmo per il calcolo dell'ottimo relativo alla funzione di benessere sociale utilitario

Il seguente algoritmo viene usato per calcolare la colorazione ottima per il gioco della k-colorazione generalizzata modellato in precedenza, utilizzando la definizione di funzione di benessere sociale utilitario.

Presentiamo ora lo pseudocodice relativo al suddetto algoritmo e di seguito analizziamo in modo accurato le caratteristiche, le iterazioni e le operazioni eseguite nell'implementazione raggiunta.

Si specifica che il seguente algoritmo è stato scritto utilizzando la Standard Library del Python e la libreria di costruzione e manipolazione di grafi NetworkX e che l'implementazione è stata accuratamente ottimizzata nei cicli e nelle operazioni grazie ad un attento uso di funzioni e strutture dati.

Anche in questo caso, la sintassi dello pseudocodice, in generale nell'aspetto e in particolare in alcuni punti nei quali trascende l'astrazione, è volutamente molto vicina a quella del linguaggio Python in modo tale da coinvolgere il lettore nell'implementazione e in modo da mostrare alcune soluzioni adottate per effettuare le operazioni più significative.

Inoltre, come per il caso precedente, sono state omesse nello pseudocodice tutte le porzioni di stampa su console e su file.out dei dati significativi, poiché trascurabili.

Si specifica che il calcolo dell'ottimo (sia secondo la definizione di benessere sociale utilitario che egalitario) è una procedura pesante e dispendiosa al livello temporale la cui strategia principale è modellata secondo la forza bruta.
In particolare tale procedura è incentrata sull'uso delle permutazioni, ciascuna delle quali rappresenta una possibile colorazione per l'istanza corrente.
Tale aspetto è fortemente esponenziale, infatti avendo k colori e n nodi, il numero di permutazioni da iterare corrisponde a k^n.

Per non tediare il lettore, specifichiamo che, per chiarezza, le qui presenti assunzioni e affermazioni verranno riportate anche per l'algoritmo successivo, quello per il calcolo dell'ottimo con funzione di benessere sociale egalitario.
Dunque lo invitiamo a saltare la lettura di tali assunzioni nella sezione successiva poiché saranno le medesime già lette in questa sezione.

```
start ← time.time()
limit ← X                          ▷ X assegnato dinamicamente dall'utente
time_limit ← False

utilitarian_social_welfare ← 0
permutations ← list(itertools.product(colors, repeat = G.number_of_nodes()))

for permutation ∈ permutations do
    if time.time() > start + limit then
        time_limit ← True
        break
    end if
    colouring_old = permutation
    for (node, data) ∈ G.nodes(data = True) do
        data['color'] ← permutation[node]
    end for
    temp_utilitarian_social_welfare ← 0
    for (node, data) ∈ G.nodes(data = True) do
        temp_utilitarian_social_welfare+ = profits[node][data['color']]
        neighbors ← G.neighbors(node)
        for neighbor ∈ neighbors do
            if G.node[neighbor]['color'] ≠ G.node[node]['color'] then
                edge_weight ← G[node][neighbor]['weight']
                temp_utilitarian_social_welfare+ = edge_weight
            else
                continue
            end if
        end for
    end for
    if temp_utilitarian_social_welfare > utilitarian_social_welfare then
        utilitarian_social_welfare ← temp_utilitarian_social_welfare
        colouring_best ← colouring_old
    else
        continue
    end if
end for

if time_limit == False then
    check_utilitarian_social_welfare ← True
    opt_utilitarian_social_welfare ← utilitarian_social_welfare
end if
```

All'inizio impostiamo, come nella sezione precedente, il limitatore temporale inizializzando la variabile $start$ con il valore temporale iniziale, la variabile $limit$ con il valore limite di X secondi e la variabile booleana $time\_limit$ con il valore $False$.

Dichiariamo la variabile $utilitarian\_social\_welfare$ e assegniamole il valore 0.
Generiamo a questo punto tutte le possibili permutazioni con ripetizione tra nodi e colori dell'istanza corrente.
Come specificato in precedenza, l'operazione potrebbe essere molto pesante, poiché vengono generate k^n permutazioni, con k uguale al numero di colori e n uguale al numero di nodi del grafo corrente.
La funzione per generare la seguente lista di permutazione utilizza primitive interne alla Standard Library del linguaggio Python, $permutations \leftarrow list(itertools.product(colors, repeat = G.number\_of\_nodes()))$.

A questo punto iteriamo tutte le permutazioni contenute all'interno della lista generata nell'operazione precedente.
Viene effettuato in questo punto il medesimo controllo sulla validità del tempo trascorso esplicato nella sezione precedente, le operazioni sono esattamente le stesse.
Se il tempo di esecuzione corrente è valido ($\leq X secondi$), salviamo la colorazione attuale, ovvero la permutazione corrente, nella variabile $colouring\_old$.

Iteriamo a questo punto tutti i nodi del grafo in oggetto attraverso un ciclo for identico a quello utilizzato nella sezione precedente.
Assegniamo a ciascun nodo, in sequenza, 1 colore appartenente alla permutazione corrente mediante l'istruzione $data['color'] \leftarrow permutation[node]$, ottenendo così una colorazione per il grafo corrente.

Dichiariamo ora la variabile temporanea $temp\_utilitarian\_social\_welfare$ e inizializziamola al valore 0, quest'ultima ci servirà per calcolare il valore del benessere sociale utilitario per la colorazione corrente.

Iteriamo nuovamente i nodi, con la medesima operazione descritta poco sopra (ciclo for).
Sommiamo, per ciascun nodo iterato, il valore del profitto associato al colore con il quale è colorato il nodo corrente al valore contenuto nella variabile $temp\_utilitarian\_social\_welfare$.

Generiamo ora la lista di tutti i nodi adiacenti al nodo corrente attraverso l'istruzione $neighbors \leftarrow G.neighbors(node)$ utilizzata anche nella sezione precedente.
Iteriamo a questo punto tutti i nodi adiacenti al nodo corrente attraverso

il medesimo ciclo for presentato durante la descrizione dell'algoritmo per il calcolo della colorazione stabile.

Istanziamo ora un controllo sulla validità del colore tra le coppie di nodi, lo stesso utilizzato nella sezione precedente.

Se la condizione $G.node[neighbor]['color'] \neq G.node[node]['color']$ è valida otteniamo che i colori tra il nodo corrente e uno dei suoi nodi adiacenti sono differenti, dunque salviamo all'interno della variabile $edge\_weight$ il valore del peso dell'arco che congiunge la coppia di nodi analizzata attraverso l'istruzione $edge\_weight \leftarrow G[node][neighbor]['weight']$ e aggiorniamo il contenuto della variabile $temp\_utilitarian\_social\_welfare$ sommandovi il valore della variabile $edge\_weight$.

Se la condizione esplicata in precedenza non è valida, non eseguiamo alcuna operazione e saltiamo l'iterazione corrente attraverso l'istruzione *continue*, in modo tale da iterare, se disponibile, il prossimo nodo adiacente al nodo corrente e ripetere il controllo.

A questo punto la variabile $temp\_utilitarian\_social\_welfare$ conterrà il valore totale relativo al benessere sociale utilitario correlato alla colorazione attuale (permutazione corrente).

Viene ora effettuato un controllo sul valore di benessere sociale utilitario trovato in precedenza.
Se la condizione
$temp\_utilitarian\_social\_welfare > utilitarian\_social\_welfare$ è valida, abbiamo trovato un valore migliore per il benessere sociale utilitario relativo all'istanza corrente.

Modifichiamo il valore della variabile $utilitarian\_social\_welfare$ con il valore della variabile $temp\_utilitarian\_social\_welfare$ e assegniamo alla variabile $colouring\_best$ il valore della variabile $colouring\_old$ (che contiene la tupla relativa alla colorazione attuale, ovvero la permutazione corrente).

Se la condizione
$temp\_utilitarian\_social\_welfare > utilitarian\_social\_welfare$ è valida, non viene effettuata alcuna operazione e viene saltata l'iterazione corrente attraverso l'utilizzo dell'istruzione *continue*.

A questo punto, se è stato calcolato il risultato ottimo all'interno del range temporale prestabilito, viene modificato al valore $True$ il contenuto della variabile $check\_utilitarian\_social\_welfare$ e viene assegnato alla variabile $opt\_utilitarian\_social\_welfare$ il valore della variabile *utilitarian_social_welfare* che contiene il risultato ottimo trovato.

Il valore della variabile *check_utilitarian_social_welfare*, quando uguale a *True*, ci permette di calcolare, in modalità SINGLE EXEC, il prezzo dell'anarchia sperimentale utilitario.

Al termine dell'esecuzione la variabile *colouring_best* conterrà la tupla che descrive e rappresenta la colorazione ottima e la variabile *opt_utilitarian_social_welfare* (*utilitarian_social_welfare*) conterrà il valore ottimo relativo alla funzione di benessere sociale utilitario per l'istanza corrente.

3.5.3 opt_egalitarian_social_welfare : l'algoritmo per il calcolo dell'ottimo relativo alla funzione di benessere sociale egalitario

Il seguente algoritmo viene usato per calcolare la colorazione ottima per il gioco della k-colorazione generalizzata modellato in precedenza, utilizzando la definizione di funzione di benessere sociale egalitario.

Presentiamo ora lo pseudocodice relativo al suddetto algoritmo e di seguito analizziamo in modo accurato le caratteristiche, le iterazioni e le operazioni eseguite nell'implementazione raggiunta.

Si specifica che il seguente algoritmo è stato scritto utilizzando la Standard Library del Python e la libreria di costruzione e manipolazione di grafi NetworkX e che l'implementazione è stata accuratamente ottimizzata nei cicli e nelle operazioni grazie ad un attento uso di funzioni e strutture dati.

Anche in questo caso, la sintassi dello pseudocodice, in generale nell'aspetto e in particolare in alcuni punti nei quali trascende l'astrazione, è volutamente molto vicina a quella del linguaggio Python in modo tale da coinvolgere il lettore nell'implementazione e in modo da mostrare alcune soluzioni adottate per effettuare le operazioni più significative.

Inoltre, come per il caso precedente, sono state omesse nello pseudocodice tutte le porzioni di stampa su console e su file.out dei dati significativi, poiché trascurabili.

Si specifica che il calcolo dell'ottimo (sia secondo la definizione di benessere sociale utilitario che egalitario) è una procedura pesante e dispendiosa al livello temporale la cui strategia principale è modellata secondo la forza bruta.
In particolare tale procedura è incentrata sull'uso delle permutazioni, ciascuna delle quali rappresenta una possibile colorazione per l'istanza corrente.
Tale aspetto è fortemente esponenziale, infatti avendo k colori e n nodi, il numero di permutazioni da iterare corrisponde a k^n.

$start \leftarrow time.time()$
$limit \leftarrow X$ ▷ X assegnato dinamicamente dall'utente
$time\_limit \leftarrow False$

$temp\_egalitarian\_social\_welfare \leftarrow 0$
$egalitarian\_social\_welfare \leftarrow 0$
$permutations \leftarrow list(itertools.product(colors, repeat = G.number\_of\_nodes()))$
$first\_iter\_check \leftarrow True$
$first\_iter\_opt\_check \leftarrow True$

for $permutation \in permutations$ **do**
    if $time.time() > start + limit$ **then**
        $time\_limit \leftarrow True$
        break
    end if
    $colouring\_old = permutation$
    for $(node, data) \in G.nodes(data = True)$ **do**
        $data['color'] \leftarrow permutation[node]$
    end for
    for $(node, data) \in G.nodes(data = True)$ **do**
        $local\_egalitarian\_social\_welfare \leftarrow 0$
        $local\_egalitarian\_social\_welfare \leftarrow profits[node][data['color']]$
        $neighbors \leftarrow G.neighbors(node)$
        for $neighbor \in neighbors$ **do**
            if $G.node[neighbor]['color'] \neq G.node[node]['color']$ **then**
                $edge\_weight \leftarrow G[node][neighbor]['weight']$
                $local\_egalitarian\_social\_welfare+ = edge\_weight$
            else
                continue
            end if
        end for
        if $first\_iter\_check$ **then**
            $temp\_egalitarian\_social\_welfare \leftarrow local\_egalitarian\_social\_welfare$
            $first\_iter\_check \leftarrow False$
            continue
        end if
        if $local\_egalitarian\_social\_welfare < temp\_egalitarian\_social\_welfare$
then
            $temp\_egalitarian\_social\_welfare \leftarrow local\_egalitarian\_social\_welfare$
        else
            continue
        end if
        $first\_iter\_check \leftarrow True$
        if $first\_iter\_opt\_check$ **then**

$egalitarian\_social\_welfare \leftarrow temp\_egalitarian\_social\_welfare$
$colouring\_best \leftarrow colouring\_old$
$first\_iter\_opt\_check \leftarrow False$
continue
end if
if $temp\_egalitarian\_social\_welfare > egalitarian\_social\_welfare$ **then**
$egalitarian\_social\_welfare \leftarrow temp\_egalitarian\_social\_welfare$
$colouring\_best \leftarrow colouring\_old$
else
continue
end if
end for
end for

if $time\_limit == False$ **then**
$check\_egalitarian\_social\_welfare \leftarrow True$
$opt\_egalitarian\_social\_welfare \leftarrow egalitarian\_social\_welfare$
end if

All'inizio, come per gli algoritmi descritti in precedenza, impostiamo il valore del limitatore temporale relativo al tempo di esecuzione massimo accettato in fase di sperimentazione.
Inizializziamo le variabili $start$, $limit$ e $time\_limit$ ai valori 0, X (X secondi) e $False$ rispettivamente.

Dichiariamo e inizializziamo al valore 0 le variabili $temp\_egalitarian\_social\_welfare$ e $egalitarian\_social\_welfare$ che ci serviranno per il calcolo del benessere sociale egalitario dell'istanza corrente.

Generiamo ora tutte le possibili permutazioni con ripetizione tra nodi e colori dell'istanza corrente, come effettuato in precedenza.
Anche in questo caso l'operazione potrebbe essere molto pesante, poiché vengono generate k^n permutazioni, con k uguale al numero di colori e n uguale al numero di nodi del grafo corrente.
La funzione per generare la seguente lista di permutazione utilizza primitive interne alla Standard Library del linguaggio Python, $permutations \leftarrow list(itertools.product(colors, repeat = G.number\_of\_nodes()))$.

Inizializziamo inoltre al valore $True$ le variabili $first\_iter\_check$ e $first\_iter\_opt\_check$ che ci serviranno all'interno di successivi controlli durante le iterazioni.

A questo punto iteriamo tutte le permutazioni contenute all'interno della lista generata nell'operazione precedente.
Viene effettuato in questo punto il medesimo controllo sulla validità del tempo trascorso esplicato nella sezione precedente, le operazioni sono esattamente le stesse.
Se il tempo di esecuzione corrente è valido ($\leq X secondi$), salviamo la colorazione attuale, ovvero la permutazione corrente, nella variabile $colouring\_old$.

Iteriamo a questo punto tutti i nodi del grafo in oggetto attraverso un ciclo for identico a quello utilizzato nella sezione precedente.
Assegniamo a ciascun nodo, in sequenza, 1 colore appartenente alla permutazione corrente mediante l'istruzione $data['color'] \leftarrow permutation[node]$, ottenendo così una colorazione per il grafo corrente.

Iteriamo nuovamente i nodi, con la medesima operazione descritta poco sopra (ciclo for).
Per ciascun nodo iterato, inizializziamo al valore 0 la variabile $local\_egalitarian\_social\_welfare$ e, nell'istruzione successiva, modifichiamo il contenuto di quest'ultima sommandovi il valore del profitto associato al colore con il quale è colorato il nodo corrente.

Generiamo ora la lista di tutti i nodi adiacenti al nodo corrente attraverso l'istruzione $neighbors \leftarrow G.neighbors(node)$ utilizzata anche nella sezione precedente.
Iteriamo a questo punto tutti i nodi adiacenti al nodo corrente attraverso il medesimo ciclo for presentato durante la descrizione dell'algoritmo per il calcolo della colorazione stabile.

Istanziamo ora un controllo sulla validità del colore tra le coppie di nodi, lo stesso utilizzato nella sezione precedente.
Se la condizione $G.node[neighbor]['color'] \neq G.node[node]['color']$ è valida otteniamo che i colori tra il nodo corrente e uno dei suoi nodi adiacenti sono differenti, dunque salviamo all'interno della variabile $edge\_weight$ il valore del peso dell'arco che congiunge la coppia di nodi analizzata attraverso l'istruzione $edge\_weight \leftarrow G[node][neighbor]['weight']$ e aggiorniamo il contenuto della variabile $local\_egalitarian\_social\_welfare$ sommandovi il valore della variabile $edge\_weight$.

Se la condizione esplicata in precedenza non è valida, non eseguiamo alcuna operazione e saltiamo l'iterazione corrente attraverso l'istruzione $continue$, in modo tale da iterare, se disponibile, il prossimo nodo adiacente al nodo corrente e ripetere il controllo.

A questo punto viene effettuato un controllo sull'iterazione corrente.
Se il valore contenuto nella variabile $first\_iter\_check$ è uguale a $True$, deduciamo che, quella corrente, è la prima iterazione.
Dunque assegniamo il valore contenuto nella variabile
$local\_egalitarian\_social\_welfare$ all'interno della variabile
$temp\_egalitarian\_social\_welfare$, modifichiamo il contenuto della variabile $first\_iter\_check$ con il valore $False$ e passiamo alla prossima iterazione attraverso l'uso dell'istruzione $continue$.

Qualora ci trovassimo in un'iterazione differente dalla prima, istanziamo il seguente controllo.
Se il valore della variabile
$local\_egalitarian\_social\_welfare < temp\_egalitarian\_social\_welfare$, abbiamo trovato un risultato migliore per il calcolo del benessere sociale egalitario.
Dunque assegniamo alla variabile $temp\_egalitarian\_social\_welfare$ il valore della variabile $local\_egalitarian\_social\_welfare$.
Se il valore della variabile
$local\_egalitarian\_social\_welfare \geq temp\_egalitarian\_social\_welfare$, non eseguiamo alcuna operazione e iteriamo il prossimo nodo del grafo attraverso l'utilizzo dell'istruzione $continue$.

A questo punto poniamo nuovamente a $True$ il valore della variabile $first\_iter\_check$ e istanziamo il seguente controllo sull'ottimalità del valore trovato.

Se il valore contenuto nella variabile $first\_iter\_opt\_check$ è uguale a $True$, deduciamo che, quella corrente, è la prima iterazione relativa al test sull'ottimalità del valore trovato.
Dunque assegniamo il valore contenuto nella variabile $temp\_egalitarian\_social\_welfare$ all'interno della variabile $egalitarian\_social\_welfare$, assegniamo il valore della variabile $colouring\_old$ (che contiene la tupla corrispondente alla colorazione corrente) alla variabile $colouring\_best$, modifichiamo il contenuto della variabile $first\_iter\_opt\_check$ con il valore $False$ e passiamo alla prossima iterazione attraverso l'uso dell'istruzione *continue*.

Qualora ci trovassimo in un'iterazione, relativa al test di ottimalità della soluzione trovata, differente dalla prima, istanziamo il seguente controllo.
Se il valore della variabile $temp\_egalitarian\_social\_welfare > egalitarian\_social\_welfare$, abbiamo trovato un risultato migliore per il calcolo del benessere sociale egalitario.
Dunque assegniamo alla variabile $egalitarian\_social\_welfare$ il valore della variabile $temp\_egalitarian\_social\_welfare$ e assegniamo il valore della variabile $colouring\_old$ (che contiene la tupla corrispondente alla colorazione corrente) alla variabile $colouring\_best$.
Se il valore della variabile $temp\_egalitarian\_social\_welfare \leq egalitarian\_social\_welfare$, non eseguiamo alcuna operazione e iteriamo il prossimo nodo del grafo attraverso l'utilizzo dell'istruzione *continue*.

A questo punto, se è stato calcolato il risultato ottimo all'interno del range temporale prestabilito, viene modificato al valore $True$ il contenuto della variabile $check\_egalitarian\_social\_welfare$ e viene assegnato alla variabile $opt\_egalitarian\_social\_welfare$ il valore della variabile $egalitarian\_social\_welfare$ che contiene il risultato ottimo trovato.

Il valore della variabile $check\_egalitarian\_social\_welfare$, quando uguale a $True$, ci permette di calcolare, in modalità SINGLE EXEC, il prezzo dell'anarchia sperimentale egalitario.

Al termine dell'esecuzione la variabile $colouring\_best$ conterrà la tupla che descrive e rappresenta la colorazione ottima e la variabile $opt\_egalitarian\_social\_welfare$ ($egalitarian\_social\_welfare$) conterrà il valore ottimo relativo alla funzione di benessere sociale egalitario per l'istanza corrente.

Abbiamo terminato a questo punto la trattazione approfondita riguardante i programmi e gli algoritmi implementati.
Analizziamo ora, all'interno del capitolo relativo alla sperimentazione, i risultati e le conclusioni ottenute dalle molteplici esecuzione effettuate.
In particolare descriveremo, nella parte iniziale, i moduli per la creazione e per la lettura utilizzati all'iterno della fase di sperimentazione.
Nel dettaglio specifichiamo che verranno analizzate parti specifiche del programma generator.py e del programma reader.py relative alla generazione e lettura di grafi randomici.
In seguito presenteremo sotto forma di tabelle riassuntive i molteplici risultati raccolti dalle varie sessioni di sperimentazione e trarremo le dovute conclusioni dallo studio e dall'analisi effettuata.

Parte IV

Sperimentazione

Capitolo 4

Sperimentazione

In questo capitolo esplicheremo e descriveremo gli aspetti e le caratteristiche generali relative all'attività di sperimentazione.

Inizieremo descrivendo in breve i moduli relativi alla generazione e alla lettura di grafi randomici situati all'interno dei 2 programmi implementati : generator.py e reader.py.

Procederemo delineando un elenco delle assunzioni e delle decisioni compiute che riguardano in generale la modellazione del problema e nello specifico la costruzione e la tipologia dei grafi implementati.
Queste rappresentano la base di partenza sulla quale è stata portata avanti l'attività di sperimentazione e sono fondamentali per comprendere meglio la parte concettuale e gli obiettivi dietro quest'ultima.

L'ultima fase prevede la presentazione delle conclusioni associate ai risultati ottenuti, in relazione al problema trattato (gioco della k-colorazione generalizzata) e in relazione alla trattazione matematica e ai teoremi delineati all'interno del documento alla base di questo studio (Generalized Graph k-Coloring Games).

4.1 gnp_random_graph (grafo di Erdős-Rényi o grafo binomiale)

Il principali oggetti matematici studiati durante l'attività di sperimentazione sono i grafi randomici.

In particolare è stata dedicata una porzione del codice all'interno del programma generator.py per la creazione e la manipolazione di una specifica tipologia di grafo : il gnp_random_graph.

Il gnp_random_graph, conosciuto anche come grafo di Erdős-Rényi o grafo binomiale, è l'oggetto attorno al quale ruota l'intera attività di implementazione.
Quest'ultimo appartiene alla classe Random Graphs.
Tale classe è contenuta all'interno della libreria di generazione e manipolazione grafi NetworkX utilizzata durante l'attività di implementazione, come specificato nella sezione precedente.
La tipologia di grafo gnp_random_graph, come delineato in precedenza, possiede dunque un costruttore di classe all'interno della suddetta libreria.

Il costruttore di classe, opportunamente parametrizzato in modo automatico oppure grazie all'intervento attivo dell'utente, è stato utilizzato per la creazione delle varie istanze di questa tipologia di grafo.

L'esecuzione dell'algoritmo di generazione comporta un complessità temporale pari a $O(n^2)$, dunque un valore accettabile.
Si cita in questo senso la presenza di algoritmi di generazione più rapidi correlati a differenti costruttori, come ad esempio l'algoritmo relativo alla tipologia fast_gnp_random_graph, che per valori piccoli di p (probabilità di generare archi tra coppie di nodi) riesce a generare grafi sparsi in $O(n+m)$.

Dato che la medio-alta densità dei grafi rende più interessante lo studio trattato è stato scelto di tralasciare questo tipo di costruttore alternativo e di utilizzare solo ed esclusivamente il costruttore gnp_random_graph per la generazione.

Descriviamo ora le modalità di generazione per le esecuzioni in modalità SINGLE EXEC e in modalità MULTIPLE EXEC e procediamo con la delineazione delle assunzioni formulate durante la modellazione dei grafi creati.

4.1.1 Generazione gnp_random_graph)

Le funzioni per la generazione di grafi gnp_random_graph sono raggiungibili all'interno del programma generator.py selezionando la modalità di esecuzione MULTIPLE MODE e selezionando in seguito la classe gnp_random_graph.

All'utente è richiesto, come primo parametro, l'inserimento del numero di iterazioni da compiere, che corrispondono al numero di grafi da costruire all'interno del singolo processo di generazione corrente.

In seguito l'utente dovrà inserire il numero di nodi per i grafi da genera-

re.
Tale valore, per assunzione, sarà un valore fisso per ciascun ciclo di generazione e dunque applicato a tutti i grafi appartenenti a quest'ultimo.

Gli ultimi parametri da inserire sono i valori massimo e minimo all'interno dei quali oscilleranno randomicamente i pesi associati agli archi del grafo, il tutto avviene nelle modalità specificate nella sezione relativa al generatore.

A questo punto l'utente sarà chiamato a scegliere la sotto modalità di generazione dei grafi, la scelta della modalità "single" comporterà la creazione di un path specifico all'interno della cartella gen nel quale verranno salvati i vari risultati della creazione seguendo lo schema presentato durante la descrizione della modalità SINGLE MODE relativa al generatore di grafi.
La scelta della modalità "multiple" comporterà la creazione di un path specifico all'interno della cartella mgen nel quale verranno salvati i vari risultati della creazione seguendo lo schema presentato durante la descrizione della modalità MULTIPLE MODE relativa al generatore di grafi.

L'ultimo valore, gestito in modo randomico per ciascuna istanza della generazione corrente, è p.
Il parametro p rappresenta la probabilità di generare un archi tra le varie coppie di nodi del grafo corrente, viene fatto oscillare in modo randomico tra i valori $0 \geq p \geq 1$.
La variabile p conterrà dunque un valore flottante (tipo float) la cui precisione è gestita in modo automatico e impostata a 2, ovvero sono ammesse fino a 2 cifre dopo la virgola.

Tralasciando i dettagli trascurabili descritti all'interno della sezione relativa all'implementazione, specifichiamo che la manipolazione del filesystem finalizzata al salvataggio degli output di generazione è effettuata in modo automatico e dinamico dal programma.
Quest'ultima è ottimizzata, cross-platform e presenta, come descritto in precedenza, primitive e funzioni appartenenti alla sola Standard Library del linguaggio Python.
Ovviamente presenta le medesimi caratteristiche dei moduli utilizzati e descritti in precedenza e dunque vengono accuratamente gestiti e evitati possibili conflitti tra i dati e problemi di inconsistenza.

4.1.2 Assunzioni generali

In breve, durante l'attività di progettazione e programmazione riguardanti la sperimentazione, sono state effettuate alcune decisioni relative a quest'ultima e sono state formulate le seguenti assunzioni sulla base delle quali si è svolta l'intera analisi.

- I risultati delle varie sessioni di sperimentazione sono presentate al lettore sotto forma di tabelle riassuntive correlate da semplici leggende riguardanti la definizione dei parametri al fine di rendere più chiara e semplice la lettura e la visione dei dati raccolti
- Ogni sperimentazione si è svolta fissando il numero di nodi e facendo oscillare randomicamente gli altri parametri come ad esempio il numero di color k oppure la probabilità relativa alla densità del grafo p.

- Il parametro p relativo alla densità di un grafo, è un valore flottante a precisione 2 (2 cifre dopo la virgola) che oscilla tra i valori $0 \geq p \geq 1$ e che descrive la probabilità di creare un arco tra un coppia di nodi e dunque definisce quanto il grafo in oggetto sia sparso (o denso)
- Il parametro k definisce il numero di colori utilizzati per generare le differenti colorazioni, ciascuna delle quali rappresenta uno stato del gioco in oggetto. Ciascun colore $i \in K$, con $K = 1, \ldots, k$ (il set di colori), rappresenta una strategia per un giocatore. Per convenzione (la sperimentazione effettuata perderebbe di senso altrimenti) il parametro k deve essere $k \leq n$, con n uguale al numero di nodi del grafo corrente
- Per ogni sessione di sperimentazione sono elencati 10 risultati rilevanti (quando possibile)
- Per le esecuzioni più pesati è stato concesso un limite temporale più ampio, ad esempio da 1 minuto a 2 minuti
- Il range all'interno del quale oscillano dinamicamente i pesi degli archi, per ciascuna esecuzione, è definito per convenzione $0 \geq a \geq 100$, con a uguale ai pesi degli archi
- Il range all'interno del quale oscillano i valore dei profitti associati ai colori per ciascun giocatore, per ciascuna esecuzione, è definito per convenzione $0 \geq p \geq 100$, con p uguale ai profitti associati ai colori per ogni giocatore

4.1.3 Tipologie di sperimentazione

Le tipologie di sperimentazione effettuate sono le seguenti

Tipologia I vengono eseguite in sequenza le seguenti operazioni su istanze di dimensione modesta, a causa dell'enorme dispendio temporale di alcune operazioni :

1. calcolo dell'ottimo relativo alla funzione di benessere sociale utilitario con limitatore temporale
2. calcolo dell'ottimo relativo alla funzione di benessere sociale egalitario con limitatore temporale
3. calcolo della colorazione stabile relativa alla definizione di equilibrio di Nash senza limitatore temporale
4. definizione del valore di benessere sociale utilitario relativo alla colorazione stabile
5. definizione del valore di benessere sociale egalitario relativo alla colorazione stabile
6. definizione del valore relativo al prezzo dell'anarchia sperimentale utilitario
7. definizione del valore relativo al prezzo dell'anarchia sperimentale egalitario

Tipologia II vengono eseguite in sequenza le seguenti operazioni su istanze di dimensione medio-grande :

1. calcolo della colorazione stabile relativa alla definizione di equilibrio di Nash con limitatore temporale
2. definizione del valore di benessere sociale utilitario relativo alla colorazione stabile
3. definizione del valore di benessere sociale egalitario relativo alla colorazione stabile

4.2 Risultati sperimentazione - Tipologia I

Qui di seguito vengono presentati i risultati relativi alla sperimentazione di Tipologia I.

LEGGENDA :

- Il parametro **k** descrive il numero di colori relativo all'istanza corrente
- Il parametro **p** descrive la probabilità relativa alla densità dell'istanza corrente
- Il parametro **a** descrive il numero di archi dell'istanza corrente
- Il valore **o_usw** è il risultato ottenuto dall'applicazione, sull'istanza corrente, del calcolo dell'ottimo con funzione di benessere sociale utilitario

- Il valore **o_esw** è il risultato ottenuto dall'applicazione, sull'istanza corrente, del calcolo dell'ottimo con funzione di benessere sociale egalitario
- Il valore **n_step** è il numero di step (mosse migliorative effettuate) ottenuto dall'applicazione, sull'istanza corrente, del calcolo della colorazione stabile seguendo la definizione di equilibrio di Nash
- Il valore **n_usw** è il risultato del calcolo del benessere sociale utilitario relativo alla colorazione stabile trovata per l'istanza corrente
- Il valore **n_esw** è il risultato del calcolo del benessere sociale egalitario relativo alla colorazione stabile trovata per l'istanza corrente
- Il valore **u_poa** è il risultato del calcolo del prezzo dell'anarchia utilitario sperimentale relativo all'istanza corrente
- Il valore **e_poa** è il risultato del calcolo del prezzo dell'anarchia egalitario sperimentale relativo all'istanza corrente
- Il parametro **t** è valore, assegnato dinamicamente, del limitatore temporale (1 = 1 minuto, 2 = 2 minuti, ecc...), il valore vale per ciascun calcolo (assegnando $t = 5$, l'esecuzione dovrebbe durare circa 10-15 minuti in totale)

4.2.1 Sperimentazione - Tipologia I - Random 3 nodi

k	p	a	o_usw	o_esw	n_step	n_usw	n_esw	u_poa	e_poa	t
3	0.90	2	496	126	4	496	126	1	1	1
2	0.55	3	406	120	1	406	91	1	1.3118	1
3	0.71	2	486	91	2	486	91	1	1	1
2	0.68	3	425	87	2	425	81	1	1.0740	1
2	0.46	2	468	97	2	468	97	1	1	1
3	0.82	3	550	157	2	527	157	1.0436	1	1
3	0.43	2	455	114	2	455	114	1	1	1
3	0.36	3	625	178	2	549	170	1.1384	1.0470	1
2	0.96	2	414	113	2	333	99	1.2432	1.1414	1
2	0.14	3	548	146	3	548	146	1	1	1

Tabella 4.1: Sperimentazione - Tipologia I - Random 3 nodi

4.2.2 Sperimentazione - Tipologia I - Random 5 nodi

k	p	a	o_usw	o_esw	n_step	n_usw	n_esw	u_poa	e_poa	t
5	0.90	9	314	45	5	314	41	1	1.0975	1
4	0.27	5	485	94	4	485	94	1	1	1
3	0.82	9	410	45	4	410	40	1	1.1250	1
4	0.14	4	394	40	4	394	40	1	1	1
2	0.79	7	420	37	5	420	37	1	1	1
5	0.65	9	497	86	6	497	86	1	1	2
3	0.40	6	805	120	5	766	120	1.0509	1	1
3	0.52	5	932	85	4	932	85	1	1	1
5	1	10	913	149	3	913	149	1	1	2
2	0.24	3	590	64	3	566	64	1.0402	1	1

Tabella 4.2: Sperimentazione - Tipologia I - Random 5 nodi

4.2.3 Sperimentazione - Tipologia I - Random 7 nodi

k	p	a	o_usw	o_esw	n_step	n_usw	n_esw	u_poa	e_poa	t
3	0.36	9	1359	75	4	1359	60	1	1.25	2
3	0.75	13	1922	206	5	1723	158	1.1154	1.3037	5
3	0.54	11	1819	118	5	1694	118	1.0737	1	5
2	0.89	18	1414	161	7	1361	117	1.0389	1.3760	2
3	0.27	8	1456	151	7	1323	117	1.1005	1.2905	2

Tabella 4.3: Sperimentazione - Tipologia I - Random 7 nodi

4.2.4 Sperimentazione - Tipologia I - Random 10 nodi

k	p	a	o_usw	o_esw	n_step	n_usw	n_esw	u_poa	e_poa	t
2	0.77	41	3294	260	8	3155	241	1.0440	1.0728	2
2	0.42	20	2281	127	7	2049	127	1.1132	1	2
2	0.26	13	1786	77	6	1682	65	1.0618	1.1846	2
2	0.99	45	3228	277	11	2935	247	1.0998	1.1214	2
2	0.52	19	1842	108	7	1735	82	1.0616	1.3170	2

Tabella 4.4: Sperimentazione - Tipologia I - Random 10 nodi

4.3 Risultati sperimentazione - Tipologia II

Qui di seguito vengono presentati i risultati relativi alla sperimentazione di Tipologia II.

LEGGENDA :

- Il parametro **k** descrive il numero di colori relativo all'istanza corrente
- Il parametro **p** descrive la probabilità relativa alla densità dell'istanza corrente
- Il parametro **a** descrive il numero di archi dell'istanza corrente
- Il valore **n_step** è il numero di step (mosse migliorative effettuate) ottenuto dall'applicazione, sull'istanza corrente, del calcolo della colorazione stabile seguendo la definizione di equilibrio di Nash
- Il valore **n_usw** è il risultato del calcolo del benessere sociale utilitario relativo alla colorazione stabile trovata per l'istanza corrente
- Il valore **n_esw** è il risultato del calcolo del benessere sociale egalitario relativo alla colorazione stabile trovata per l'istanza corrente
- Il parametro **t** è valore, assegnato dinamicamente, del limitatore temporale (1 = 1 minuto, 2 = 2 minuti, ecc...)

4.3.1 Sperimentazione - Tipologia II - Random 15 nodi

k	p	a	n_step	n_usw	n_esw	t
7	0.75	78	9	9398	365	1
7	0.84	83	19	9664	457	1
13	0.49	50	17	6837	176	1
5	0.30	35	7	5226	133	1
9	0.87	90	19	10747	586	1
5	0.49	41	15	5206	174	1
9	0.89	97	14	10937	451	1
10	0.79	84	19	10437	582	1
4	0.88	87	9	9295	473	1
13	0.38	34	14	4185	161	1

Tabella 4.5: Sperimentazione - Tipologia II - Random 15 nodi

4.3.2 Sperimentazione - Tipologia II - Random 30 nodi

k	p	a	n_step	n_usw	n_esw	t
29	0.09	48	30	7507	117	2
3	0.86	366	40	32140	808	2
11	0.15	72	36	9206	179	2
16	0.03	12	14	3207	106	2
8	0.15	70	37	10159	89	2
6	0.20	77	35	10730	193	2
6	0.36	151	38	15663	296	2
3	0.39	168	19	15748	286	2
28	0.56	248	34	26688	633	2
17	0.42	191	46	22487	376	2

Tabella 4.6: Sperimentazione - Tipologia II - Random 30 nodi

4.3.3 Sperimentazione - Tipologia II - Random 45 nodi

k	p	a	n_step	n_usw	n_esw	t
11	0.69	666	74	70663	74	5
19	0.82	810	56	86036	1522	5
41	0.28	284	55	32495	351	5
42	0.17	185	48	23126	225	5
23	0.62	633	70	67407	953	5
5	0.97	964	58	85815	1488	5
25	0.23	205	48	25993	231	5
29	0.58	557	57	58736	923	5
9	0.49	481	62	51387	561	5
15	0.32	338	52	37617	544	5

Tabella 4.7: Sperimentazione - Tipologia II - Random 45 nodi

4.3.4 Sperimentazione - Tipologia II - Random 60 nodi

k	p	a	n_step	n_usw	n_esw	t
30	0.27	456	67	51948	514	10
52	0.05	52	56	13850	101	10
13	0.44	783	83	85091	956	10
51	0.84	1484	87	158991	2163	10
38	0.77	1331	107	139927	1832	10
11	0.90	1584	94	161166	2144	10
32	0.66	1201	98	124281	1336	10
3	0.23	415	65	40080	374	10
14	0.39	711	92	76014	609	10
47	0.86	1507	90	158109	2236	10

Tabella 4.8: Sperimentazione - Tipologia II - Random 60 nodi

4.3.5 Sperimentazione - Tipologia II - Random 75 nodi

k	p	a	n_step	n_usw	n_esw	t
14	0.04	107	80	17362	94	10
24	0.89	2498	141	254295	2593	10
23	0.23	703	99	79253	586	10
2	0.35	960	44	64429	573	10
41	0.07	207	74	28634	174	10
10	0.33	918	95	99490	665	10
73	0.10	310	77	39623	175	10
16	0.14	400	96	46173	221	10
8	0.08	221	85	28302	92	10
6	0.29	803	115	85384	636	10

Tabella 4.9: Sperimentazione - Tipologia II - Random 75 nodi

4.3.6 Sperimentazione - Tipologia II - Random 90 nodi

k	p	a	n_step	n_usw	n_esw	t
27	0.63	2554	173	265205	2301	15
53	0.36	1468	144	158319	1059	15
25	0.36	1466	132	156640	896	15
35	0.18	722	109	81552	443	15
45	0.47	1846	122	196421	1568	15
20	0.17	652	122	72931	375	15
7	0.35	1443	135	148117	977	15
30	0.85	3433	154	357717	3387	15
45	0.18	716	105	79042	379	15
11	0.56	2230	156	227861	1881	15

Tabella 4.10: Sperimentazione - Tipologia II - Random 90 nodi

4.3.7 Sperimentazione - Tipologia II - Random 100 nodi

k	p	a	n_step	n_usw	n_esw	t
13	0.64	3252	199	330952	2706	20
3	0.56	2724	147	208293	1490	20
12	0.38	1898	173	197154	1210	20
15	0.28	1368	154	147745	873	20
14	0.05	241	104	33638	100	20
12	0.20	954	149	107103	522	20
13	0.77	3790	234	381704	2963	20
33	0.43	2119	161	225302	1483	20

Tabella 4.11: Sperimentazione - Tipologia II - Random 100 nodi

4.4 Conclusioni

La suddetta sperimentazione effettuata ha pienamente soddisfatto le aspettative e gli obiettivi prefissati in fase di progettazione.
Sia la tipologia di sperimentazione I che la tipologia di sperimentazione II hanno prodotto e portato alla luce risultati significativi che testimoniamo la validità e la correttezza dei teoremi affrontati nel Capitolo 2.
Inoltre l'assoluta accuratezza e precisione delle procedure applicate è testimoniata dalla grande precisione degli output ottenuti.

Per ciò che concerne la Tipologia I, specifichiamo che l'enorme dispendio temporale e l'ingente quantità di risorse richieste dai calcoli effettuati, hanno limitato, nella pratica, l'attività di sperimentazione.
La quantità di risultati ottenuti è però sufficiente per trarre le dovute riflessioni e conclusioni.
Per completezza specifichiamo che l'ordine delle permutazioni (colorazioni) analizzate, per ciascun calcolo dell'ottimo, è dell'ordine del milione.
Si è scelto di restare all'interno di questa soglia in modo da ottenere esecuzioni che non superino le 1-2 ore al massimo.

Analizzando gli output ottenuti relativa alla sperimentazione di tipo I, possiamo affermare con certezza che molti parametri influiscono in particolare con le operazioni di calcolo dell'ottimo sia con funzione di benessere sociale utilitario che egalitario.
In particolare possiamo tralasciare l'analisi della complessità riguardante

l'algoritmo per il calcolo della colorazione stabile poiché la grandezza dei grafi utilizzati determina un dispendio temporale trascurabile.

Concentrandoci invece sugli algoritmi per il calcolo degli ottimi con funzioni di benessere sociale utilitario e egalitario, possiamo affermare che il parametro k influisce direttamente sulla complessità temporale relativa alle varie esecuzioni poiché determina il numero di colorazioni (permutazioni) da iterare.

Nello specifico, dato come valore fisso il numero di nodi n, l'uso del parametro k in correlazione con quest'ultimo genera un numero di permutazioni pari a k^n, è dunque immediato comprendere come al crescere di k crescerà anche il numero di permutazioni k^n e di conseguenza la complessità temporale dell'algoritmo in modo esponenziale.

Un altro parametro che influenza direttamente la complessità temporale relativa agli algoritmi per il calcolo degli ottimi è p.
Il parametro p definisce il grado di densità del grafo in oggetto, dunque determina la probabilità di costruire archi tra le coppie di nodi.
La forte connessione del grafo e dunque un valore di densità elevato per quest'ultimo determinano una maggiore complessità temporale per gli algoritmi relativi al calcolo degli ottimi.
La presenza di molti archi nel grafo genera l'esistenza di un numero maggiore di nodi adiacenti per ciascun nodo del grafo e dunque aumenta in modo diretto il numero di iterazioni innestate all'interno dell'algoritmo.

Inizializzando con valori piccoli e medi i suddetti parametri k e p otteniamo esecuzioni caratterizzate da una complessità temporale minore.

Analizzando in seguito i parametri ottenuti, in particolare riguardo il prezzo dell'anarchia sperimentale utilitario e quello egalitario, possiamo notare, dai risultati ottenuti, come sia facile, in caso di grafi piccoli, ottenere colorazioni stabili con un benessere sociale utilitario e egalitario molto vicino se non pari al benessere sociale utilitario e egalitario delle colorazioni ottime.

Molto spesso infatti otteniamo il valore 1 per ciò che riguarda il prezzo dell'anarchia sperimentale utilitario o egalitario o entrambi, tale valore conferma che la colorazione stabile trovata dall'algoritmo presenta un valore di benessere sociale utilitario o egalitario o entrambi pari al valore dei rispettivi ottimi.

In generale possiamo confermare che la totalità dei risultati ottenuti rispetta pienamente la descrizione del modello matematico e le affermazioni determinate dai vari teoremi presentati all'interno del Capitolo 2.

Possiamo affermare che il gioco implementato è convergente poiché sono stati sempre ottenuti risultati stabili (equilibri di Nash) con complessità temporale polinomiale (nei casi di studio).
Abbiamo inoltre ottenuto valori per il prezzo dell'anarchia utilitario sperimentale che rispettano a pieno le asserzioni contenute nel Teorema 1, poiché abbiamo ottenuto sempre risultati ≤ 2.

Per ciò che riguarda la sperimentazione di tipo II, l'analisi è meno complessa.
Anche qui il parametro k influenza la computazione, sia al livello temporale che concettuale.
Il parametro k influenza la complessità temporale dell'algoritmo in misura del tutto minore se confrontata con quella relativa agli algoritmi per il calcolo delle 2 tipologia di ottimo implementate per il gioco in oggetto.
Nonostante ciò il parametro determina l'aumento delle iterazioni innestate che interessano la ricerca del miglioramento per ciascun nodo, tale aumento è ovviamente direttamente proporzionale al crescere di k.

Il parametro k influenza anche la complessità concettuale del gioco poiché ciascun giocatore, avendo più strategie, dovrà cercare più a lungo le mosse migliori, in modo da trovare la dinamica più adatta all'istanza corrente.

Il parametro p, anche in questa tipologia di sperimentazione, influenza direttamente la complessità temporale del calcolo poiché aumentando la presenza di connessioni e dunque la densità del grafo corrente, aumenta anche il numero di nodi adiacenti per ciascun nodo e di conseguenza il numero totale del iterazioni necessarie.

Il limitatore temporale t in questo caso specifica un'altra caratteristica fondamentale di questo tipo di analisi effettuata sul gioco in oggetto, ovvero l'aumento graduale del tempo necessario a ciascuna esecuzione all'aumentare del numero di nodi n (fissato per ogni sperimentazione).
Difficilmente il variare dei parametri k e p produce variazioni significative al livello temporale (come ad esempio un aumento netto del tempo richiesto) e infatti per la quasi totalità dei casi è corretto impostare un set di valori omogeneo per t.

Per concludere affermiamo inoltre che l'aumento del numero di nodi n, del numero dei colori k e della probabilità che descrive la densità del grafo influenzano direttamente il parametro *n_step* generando nella quasi totalità dei casi un generale aumento più o meno significativo di quest'ultimo.
Inoltre possiamo specificare che al crescere di n, per entrambe le tipologie di sperimentazione, è possibile notare un generale aumento dei valori relativi ai parametri relativi al benessere sociale utilitario, poiché abbiamo più membri

all'interno della sommatoria definita dalla funzione.
Per ciò che riguarda i parametri legati al benessere sociale egalitario il discorso è leggermente differente, questi ultimi infatti sono molto influenzati dal valore di p poiché un grafo più denso genera l'aumento generale del profitto individuale per ciascun giocatore e di conseguenza del profitto minimo individuale.

Bibliografia

[1] NetworkX Developers. *NetworkX Library*. `https://networkx.github.io/documentation/stable/`. 2004-2018.

[2] Python Software Foundation. *Python Standard Library*. `https://docs.python.org/3/library/index.html`. 2001-2018.

[3] G.Monaco R.Carosi. "Generalized Graph k-Coloring Games". In: *Gran Sasso Science Institute (GSSI)* (2018).

[4] M.Dantani G.Sukthankar E Andre S.Koenig. "Stable Outcomes in Modified Fractional Hedonic Games". In: *Internation Conference on Autonomous Agents and Multiagents Systems (AAMAS 18)* (2018).

Ringraziamenti

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.