Università degli Studi di Catania
dipartimento di matematica e informatica

Algoritmi euristici per il Problema della Dominazione Romana

Presentata in adempimento parziale
dei requisiti per l'acquisizione del titolo di
Dottorato di ricerca in Informatica

Candidato:
Salvatore Mario Nolassi

Supervisore:
Chiar.mo Prof. Vincenzo Cutello

Coordinatore:
Chiar.mo Prof. Vincenzo Cutello

Corso di Dottorato di Ricerca - XXV Ciclo

A tutti quelli che hanno un sogno da realizzare

Sommario

Una funzione di Dominazione Romana su un grafo G è una funzione di copertura $f: V \to \{0, 1, 2\}$ tale che ogni vertice con valore 0 abbia almeno un vicino con valore 2. Il numero di Dominazione Romana di un grafo G è definito come il minimo tra tutte le funzioni di dominazione romana di $\sum_{u \in V} f(u)$. Dopo un'introduzione ai concetti base che ruotano attorno alla Dominazione Romana, viene studiato il problema per due particolari classe di grafi, cioè grafi a griglia e i bipartiti. Per i grafi a griglia vengono prodotti degli schemi di copertura ottimali per griglie di qualsiasi dimensione e inoltre vengono migliorati i limiti superiori e inferiori noti del numero di Dominazione Romana. Per quanto riguarda i grafi bipartiti viene proposto un approccio che partendo da un insieme di vertex cover del grafo produce una funzione di Dominazione Romana in tempo polinomiale. Nel prosieguo della dissertazione vengono mostrati vari approcci euristici per qualsiasi classe di grafo. In particolare viene prodotto un algoritmo euristico che in tempo polinomiale calcola una copertura e un numero di Dominazione Romana che rientra nei limiti teorici noti introducendo un nuovo parametro associato ai vertici del grafo. Infine una delle varianti della stessa euristica è stata implementata su architettura CUDA permettendo la parallelizzazione del calcolo su GPU e saranno mostrate le strutture dati utilizzate e le problematiche riscontrate.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Prologo

1.1 Dominazione sui grafi

Il problema della Dominazione nei grafi è sempre stato un campo su cui si è rivolta in particolari modo l'attenzione della ricerca nel campo della teoria dei grafi. In un grafo, un insieme dominante è un sottoinsieme D dei vertici tali che ogni vertice o è in D o è adiacente ad un vertice in D. Lo studio della problematica è iniziata attorno gli anni '50 e ha avuto un'impennata attorno alla metà degli anni '70. Una testimonianza è un libro dedicato al tema della Dominazione [60] che riporta più di 1200 lavori correlati alla Dominazione sui grafi e il lavoro di approfondimento fatto in [20]. La prima formulazione del problema nasce dalla necessità per i giocatori di scacchi di conoscere il numero minimo di regine tale che ogni quadrato della scacchiera possa essere attaccato da una regina oppure contiene esso stesso una regina [9]. Con il passare del tempo, numerose problematiche sono state ricondotte al problema della Dominazione come per esempio l'ubicazione di impianti tecnologici e di servizi (server, ospedali, caserme) e la conseguente progettazione di circuiti, strade e rete telefonica [80], [74], [44].

Recentemente il problema della dominazione su grafi è stato accostato anche allo studio delle reti sociali per studiare l'influenza dei singoli vertici e tracciare la direzionalità e la velocità di propagazione di entità attraverso le connessioni

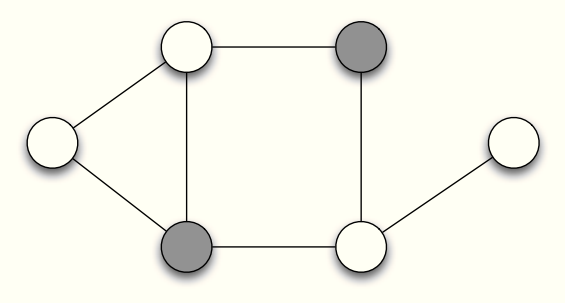


Figura 1.1: Insieme dominante di G.

presenti, come si può vedere in [30], [33] e [32]. Per fare un esempio dell'importanza dell'applicazione del problema si può fare riferimento allo scenario che vede i vertici che fanno parte dell'insieme dominante come un insieme di individui. Se consideriamo le connessioni tra i vertici come se rappresentassero un rapporto di conoscenza, allora l'insieme dominate rappresenta il numero minimo di individui che conoscono tutti nella rete di riferimento.

Dato $G = (V, E)$, il numero di dominazione $\gamma(G)$ è il numero di vertici in un insieme dominante minimo per G e in questo caso si può parlare di insieme dominante minimo. Il problema di verificare se $\gamma(G) \leq k$ per un dato grafo G e valore di input k, è un problema decisionale NP-completo classico nella teoria della complessità computazionale, come mostrato in [28]. La Figura 1.1 mostra un semplice esempio di grafo dove i vertici grigi rappresentano l'insieme dominante.

Esistono circa 80 varianti del problema della Dominazione, la maggior parte raccolte in [60], che pongono l'attenzione su particolari caratteristiche nel selezionare l'insieme D (collegato, indipendente) oppure condizioni su $V - D$ (ogni vertice dominato esattamente o almeno k volte). In questo lavoro verrà trattata la variante conosciuta come Dominazione Romana. Come sarà presentato nel prosieguo, questa variante si pone l'obbiettivo di assegnare un valore scelto tra $(0, 1, 2)$ a tutti i vertici, con l'unico vincolo che ogni vertice con il valore 0 abbia un vertice adiacente con il valore 2, con lo scopo di minimizzare la somma di tutti i valori assegnati ai vertici.

1.2 Definizioni sulla Dominazione Romana

Per convenzione, un grafo è determinato dai suoi archi e dell'insieme dei vertici, quindi si scriverà $G = (V, E)$ anche se la maggior riparte delle volte si farà riferimento ad esso con la forma abbreviata G. Per completezza, $V(G)$ e $E(G)$, o più semplicemente V e E denotano rispettivamente l'insieme dei vertici e degli archi del grafo G. Per un grafo $G = (V, E)$, con $|V|$ si definisce la cardinalità dell'insieme dei vertici presenti in G, mentre con $\{u, v\}$ si indica un generico arco di G. Verranno presi in considerazione solamente grafi non orientati cioè grafi in cui gli archi non hanno orientamento. Questo significa che l'arco (u, v) è identico all'arco (v, u). Sotto queste ipotesi il numero massimo di archi in un grafo non orientato risulta essere uguale a $|V|(|V| - 1)/2$.

In un grafo $G = (V, E)$, il grado di un vertice, denotato con $deg(v)$, indica il numero di vertici adiacenti ad esso. Il grado minimo e quello massimo di un grafo $G = (V, E)$ sono denotati rispettivamente con $\delta(G) = mindeg(v) : v \in V$ e con $\Delta(G) = maxdeg(v) : v \in V$. Un vertice si dice isolato se il suo grado è pari a zero, e un grafo si dice connesso se non possiede vertici isolati. Per ogni vertice $v \in V$, il vicinato aperto di v è l'insieme $N(v) = u \in V : \{u, v\} \in E$ e il vicinato chiuso è l'insieme $N[v] = N(V) \cup v$. Per un insieme $S \subseteq V$, il vicinato aperto è $N(S) = \bigcup_{v \in S} N(v)$ e il vicinato chiuso è $N[S] = N(S) \in S$.

Il problema della Dominazione Romana è un problema relativamente recente della teoria dei grafi introdotto in [63], ripreso in [19] e [55] poi approfondito e formalizzato dal punto di vista matematico principalmente in [18], [39], e nella tesi di dottorato [10].

Dato un grafo non orientato $G = (V, E)$, una *Funzione di Dominazione Romana* (spesso abbreviata con la sigla RDF) è definita come una funzione $f \colon V \to \{0, 1, 2\}$ che soddisfa la condizione che ogni vertice $u \colon f(u) = 0$ sia adiacente ad almeno un vertice $v \colon f(v) = 2$. Il peso di una Funzione di Dominazione Romana è la somma di tutti i valori assegnati $f(u) \colon u \in E$. Il peso minimo è conosciuto come il *Numero di Dominazione Romana* e il problema della Dominazione Romana si prefigge proprio

di trovare questo valore. Quindi per un qualsiasi grafo G, si definisce *Numero di Dominazione Romana* il valore minimo dell'insieme delle funzioni di Dominazione Romana:

$$\gamma_R(G) = \min_{f \in F} f(V) = \min_{f \in F} \sum_{u \in V} f(u).$$

dove F è l'insieme di tutte funzioni di Dominazione Romana valide per il grafo G.

Il problema prende spunto dal passaggio di una strategia militare *forward defense*, in cui le legioni romane presidiavano anche le più lontane province dell'Impero, ad una strategia *defense in depth*, ideata dall'imperatore Costantino, per fronteggiare la riduzione del numero di legioni. Con la nuova strategia, una provincia era considerata *sicura* se era presidiata da una o più legioni. D'altra parte, poteva dirsi *difendibile* se questa provincia confinava con una provincia che era presidiata da almeno due legioni, altrimenti la provincia era detta *non-sicura*.

L'idea è che i valori 1 e 2 rappresentano rispettivamente una e due legioni romane di istanza in una data località, rappresentata come un vertice v del grafo G. Una località vicina u (un vertice adiacente a v) è considerata non protetta se non possiede legioni di istanza (cioè $f(u) = 0$). Una posizione non protetta u poteva essere assicurata mediante l'invio di una legione a u da un luogo vicino v. Ma l'imperatore Costantino il Grande, nel IV secolo D.C, ha decretato che una legione non può essere spostata da una posizione v se così facendo si lascia la posizione non protetta (cioè se $f(v) = 1$). Così, due legioni devono essere di istanza in un luogo ($f(v) = 2$) prima che una possa essere inviata a un luogo vicino. Pensando di rappresentare tutte le località più importanti dell'impero romano tramite i vertici di un grafo connessi tra di loro dalle strade costruite dai romani, si ha una rappresentazione del problema e della nascita del nome della variante.

Dato un grafo $G = (V, E)$, sia $f \colon V \to \{0, 1, 2\}$ una funzione di Dominazione Romana e sia (V_0, V_1, V_2) la partizione indotta da f, dove $V_i = v \in V | f(v) = i$ con $V_i = n_i$, per $i = 0, 1, 2$. Da notare che si tratta di una corrispondenza 1-1 tra la funzioni $f \colon V \to 0, 1, 2$ e la partizione (V_0, V_1, V_2) di V e ciò permette di scrivere la funzione di Dominazione Romana anche nella forma $f = (V_0, V_1, V_2)$. Una funzione

$f = (V_0, V_1, V_2)$ è una funzione di Dominazione Romana se $V2 \succeq V0$, dove $\succeq$ sta ad indicare che l'insieme V_2 domina l'insieme V_0, cioè $V_0 \subseteq N[V2]$. Sotto queste ipotesi si ha che il peso di f è $f(V) = \sum_{v \in V} f(v) = 2n_2 + n_1$ e il numero di Dominazione Romana, denotato $\gamma_R(G)$ e a volte abbreviato con la sigla RDN, è uguale al peso minimo di una RDF su G, e si dice che una funzione $f = (V_0, V_1, V_2)$ è una funzione γ_R se risulta essere una RDF e $f(V) = \gamma_R(G)$.

Come dimostrato in [18], ricollegandosi al problema della dominazione nella sua formulazione generica essa è il relazione alla variante romana come mostrato di seguito.

Proposizione 1. *Per qualsiasi grafo G:*

$$\gamma(G) \leq \gamma_R(G) \leq 2\gamma(G).$$

In [10] si fornisce la dimostrazione del seguente teorema:

Teorema 1. *La funzione di Dominazione Romana è un problema NP-completo.*

La dimostrazione viene affrontata tramite una riduzione al problema del 3-SAT e in maniera analoga si dimostra che anche nel caso di grafi bipartiti e grafi planari il problema si presenta in forma NP-completa. Dalla Proposizione 1 si ha che $\gamma_R(G) \leq 2\gamma(G)$. Si dice che un grafo G è *Romano* se $\gamma_R(G) = 2\gamma(G)$, quindi di può dire che:

Proposizione 2. *Un grafo G è Romano se e solo se esiste una funzione di Dominazione Romana $f = (V_0, V_1, V_2)$ tale che:*

$$n_1 = |V_1| = 0.$$

Proposizione 3. *Per qualsiasi grafo G di ordine n, $\gamma(G) = \gamma_R(G)$ se e solo se $G = \overline{K_n}$.*

Sempre in [18] si fanno ulteriori considerazioni sulle caratteristiche della partizione dei vertitici di G indotta dalla funzione di Dominazione Romana. I più importanti per il lavoro svolto vengono riportati qui di seguito.

Proposizione 4. *Sia $f = (V_0, V_1, V_2)$ una funzione di Dominazione Romana. Allora*

- *$G[V_1]$, che risulta essere il sotto-grafo indotto da V_1, ha grado massimo uguale a 1.*
- *Non esistono archi di G che vanno da V_1 a V_2.*
- *Ogni vertice di V_0 è adiacente al massimo a due vertici di V_1.*
- *V_2 è un insieme dominante per il grafo $G[V_0 \cup V_2]$.*
- *Per qualsiasi grafo G privo di vertici isolati, posto che n_1 sia minimale allora $n_0 \geq 3 \cdot n/7$.*

Il limite inferiore seguente che si riferisce al numero di Dominazione Romana di qualsiasi grafo è dimostrato in [39].

Proposizione 5. *Per qualsiasi grafo G di ordine n con grado massimo $\Delta(G)$, si ha:*

$$\frac{2n}{\Delta(G)+1} \leq \gamma_R(G).$$

In [61] tramite un metodo probabilistico viene dimostrato il seguente limite superiore:

Proposizione 6. *Per un qualsiasi grafo G di ordine n con grado minimo $\delta(G)$ si ha:*

$$\gamma_R(G) \leq n \cdot \frac{2 + \ln((1+\delta(G))/2)}{1+\delta(G)}.$$

In [76] vengono dimostrati due limiti superiori per la funzione di Dominazione Romana:

Proposizione 7. *Per qualsiasi grafo G di ordine n:*

$$\gamma_R(G) \leq \frac{4n}{5}.$$

Proposizione 8. *Per qualsiasi grafo G di ordine $n \geq 9$, e grado minimo $\delta(G) \geq 2$ si ha:*

$$\gamma_R \leq \frac{8n}{11}.$$

In [58] viene fatta la seguente caratterizzazione degli insieme risultanti dalla partizione:

Proposizione 9. *Per qualsiasi grafo G di ordine $n \geq 3$ si ha che:*

- $|V_0| \geq n/5 + 1$;
- $|V_1| \leq 4n/5 - 2$;
- $|V_2| \leq 2n/5$.

Alcuni di questi limiti verranno richiamati per dimostrare la bontà delle risultati forniti dagli algoritmi euristici, specificando anche quale limite risulta essere più stretto in base alle caratteristiche dei grafi.

Come detto precedentemente il problema della Dominazione Romana nella sua versione decisionale, risulta essere NP-completo e le ricerche recenti di approfondimento rivolgono la propria attenzione nel caratterizzare specifiche classi di grafi.

In [77] viene mostrato un problema di locazione di risorse che viene risolto tramite un'applicazione della Dominazione Romana trattando il problema di posizionare un numero minimo di server in modo tale che due richieste provenienti da vertici differenti possano essere servite con due server diversi. In [24] viene mostrata un'applicazione su grafi a disco (grafi di intersezione di dischi di dimensioni uguali nel piano) ampiamente usati come modello matematico per la progettazione di reti wireless, per i quali viene descritto un algoritmo di approssimazione in tempo lineare.

In [21] viene fatta una caratterizzazione della Dominazione Romana sugli alberi, in [50] si dimostra che il numero di Dominazione Romana di un grafo intervallo (grafo che rappresenta le intersezioni di una insieme di intervalli su una retta reale) può essere calcolato in tempo lineare così come nel caso di co-grafi (grafi generati a partire da un singolo vertice K_1 procedendo per completamento e unione disgiunta)

oltre a mostrare l'esistenza di algoritmi capaci di calcolare il numero di Dominazione Romana in tempo polinomiale per grafi con un d-octopus (un d-octopus di un grafo $G = (V, E)$ è un sottografo $T = (W, F)$ di G tale che W è un insieme dominante di G, e T è l'unione di d percorsi minimi di G che hanno un estremo in comune [38]). L'obbiettivo che si propone questo dissertazione è fornire degli algoritmi capaci di produrre un'approssimazione in tempo polinomiale per qualsiasi grafo.

1.3 Classi di grafi presi in esame

Avendo l'obbiettivo di produrre e validare degli algoritmi euristici che abbiano un buon comportamento su qualsiasi classe di grafi, sono state implementate delle procedure per la generazione di un vasto insieme eterogeneo di grafi su cui sono poi state provate le varie implementazioni capaci di generare una copertura romana per tali grafi. Le famiglie di grafi generate sono state grafi random tramite il modello di Edgar Gilbert, grafi random tramite il modello di Barabási-Albert, grafi bipartiti, grafi planari, grafi a griglia e grafi del Re.

Si definisce random un grafo generato tramite un qualche processo casuale e non riproducibile. La teoria che ruota attorno ai grafi random è un'intersezione tra teoria dei grafi e teoria probabilistica a cui sono stati dedicati parecchi lavori tra cui [3]. Un grafo random è ottenuto partendo da un insieme di n vertici isolati, quindi non collegati tra di loro, procedendo successivamente all'aggiunta di archi che li colleghino in maniera casuale. Se in un grafo pur essendo casuale, è desiderabile che siano presenti determinate proprietà, allora la procedura che lo genera può avere dei parametri di ingresso che guidano la creazione del grafo stesso. La scelta dei parametri da utilizzare durante la generazione del grafo, genera un modello differente. Ogni modello produce una distribuzione di probabilità degli archi del grafo diversa come ampiamente trattato in [3].

Il primo modello che si è scelto di implementare in questo lavoro è stato quello di Edgar Gilbert [16], comunemente denotato con $G(n, p)$, nel quale ogni possibile arco, è presente con probabilità $0 < p < 1$. Le probabilità utilizzate variano da 0.01

a 0.9 e il numero di vertici presenti per ogni grafo varia tra i 50 e i 10000. Con una leggera variante del modello si sono generati anche dei grafi random con un grado massimo fissato quindi un $\Delta(G) = k$ con k dato in input.

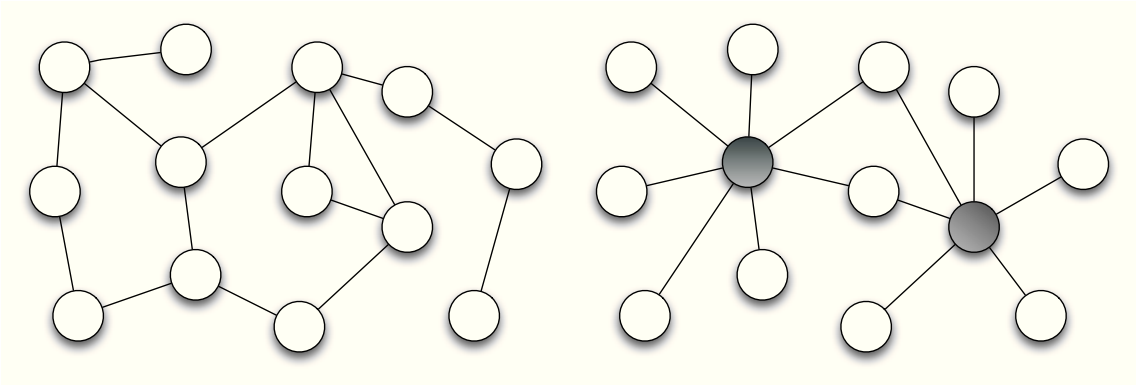
Un secondo modello di grafi random implementato è stato quello di Barabási-Albert [1]. Questo modello incorpora un nuovo concetto nei grafi random, cioè la formalizzazione di una rete a invarianza di scala. In un grafo che gode di questa proprietà, la distribuzione dei gradi dei vertici che compongono il grafo seguono una legge di potenza. Questo significa che si può individuare una partizione di vertici $P(k)$ che hanno k connessioni verso altri vertici e per k abbastanza grande si ha:

$$P(k) \sim k^{-\rho}.$$

dove ρ è un parametro che solitamente oscilla nell'intervallo $2 < \rho < 3$ e nel caso specifico del modello di Barabási-Albert è uguale a 3.

Una caratteristica che scaturisce da questa proprietà è che nei grafi con invarianza di scala sono presenti dei particolari vertici detti *hub* che hanno un grado di connessione molto alto rispetto alla media. Questa tipologia di grafi sono stati osservati in svariati contesti naturali e sociali, dai social network alla diffusione delle malattie fino ad arrivare alla struttura di Internet.

Il modello riprende e implementa due caratteristiche osservabili anche nelle reti appena nominate: la crescita continua e la giunzione preferenziale. La crescita continua implica che il grafo può continuare a crescere nel tempo. La giunzione preferenziale indica la tendenza dei nuovi vertici a collegarsi a vertici che hanno già un alto grado di connessione. Quindi più il vertice è connesso all'interno della sua rete, più è probabile che riceva altre connessioni in fase di crescita. Intuitivamente, come trattato anche in [11] e in [64] questo fenomeno è osservato per esempio nei collegamenti che si creano su Internet in cui si ha la tendenza di un nuovo sito si incorporare i collegamenti a siti molto popolari sul web. Così come un nuovo utente di un social network tende a collegarsi con utenti popolari sul social piuttosto che con utenti sconosciuti. Un'altra interessante proprietà del modello Barabási-Albert, è



(a) Grafo random (b) Grafo con invarianza di scala

Figura 1.2: Esempio di grafo random (a) e grafo con invarianza di scala (b).

che la lunghezza del cammino medio tra una qualsiasi coppia di vertici risulta essere più breve rispetto a un grafo random classico. Questa grandezza cresce in maniera approssimativamente logaritmica rispetto alla grandezza del grafo stesso. La formula che regola tale grandezza risulta essere:

$$l \sim \frac{lnN}{lnlnN}.$$

La procedura di generazione di un grafo così formato parte da una configurazione iniziale di $m_0$vertici. I vertici iniziali sono connessi tra di loro tali da formare un $\overline{K_{m_0}}$. I nuovi vertici vengono aggiunti al grafo uno per volta. Ogni nuovo vertice viene connesso ai vertici esistenti con una probabilità proporzionale ai collegamenti del vertice già esistenti. Più formalmente la probabilità p_i che un nuovo vertice sia connesso ad un vertice i è:

$$p_i = \frac{k_i}{\sum_j k_j}$$

dove k_i rappresenta il grado del vertice i e la sommatoria racchiude tutti gli archi già presenti nel grafo [2]. Con questa procedura, i vertici fortemente connessi tendono a raccogliere sempre più connessioni mentre è più raro che i nuovi vertici si connettano a vertici esistenti con un basso grado di connessioni. Una configurazione tipica dei due modelli di grafi random è mostrata in Figura 1.2.

Alcune prove sono state fatte su una leggera variante del modello. La variante prevede la possibilità di decidere in fase di inizializzazione il numero di archi che collegheranno ogni nuovo vertice al grafo esistente. Detto M questo valore, risulta un prerequisito il fatto che sia $M \leq m_0$.

Tramite una personalizzazione del modello esposto in [16], usato già per la generazione di grafi random, è stata generato un insieme di grafi bipartiti. Si ricorda che un grafo si dice bipartito se è possibile suddividere i vertici presenti in due sottoinsiemi disgiunti in maniera che non siano presenti archi che collegato due vertici che fanno parte dello stesso insieme. Più formalmente dato un grafo $G = (V, E)$ si dice che G è bipartito se l'insieme dei vertici V può essere suddiviso in due sottoinsiemi disgiunti $V = V_1 \cup V_2$ tale che $\forall(e) \in E \Rightarrow e = (v_1, v_2) : v_1 \in V_1, v_2 \in V_2$. Anche per questa classe di grafi è stata usata una probabilità di connessione tra vertici che varia tra 0.01 e 0.9 e un numero variabile tra i 50 e i 10000 vertici. In questo caso quindi la probabilità di connessione esprime la percentuale di connessioni tra un vertice $v_1 \in V_1$ e l'insieme dei vertici di V_2.

Altra classe di grafi generati sono stati i grafi planari. Si ricorda che si definisce grafo planare un grafo che può essere raffigurato in un piano in modo che non si abbiano archi che si intersecano tra di loro. Nella fase di generazione di questi particolari grafi, si sono associate delle coordinate geometriche ad ogni vertice e si è fatto in modo di collegare i vertici tra loro prediligendo giunzioni di vertici che posti sul piano abbiano una distanza euclidea ridotta tra di loro. Alla fine del processo di creazione e popolazione del grafo, comunque mantenuto casuale, si è verificata l'effettiva planarità di esso tramite il test di planarità esposto in [67].

Una classe di grafi che sono serviti sia in fase di calcolo che in fase di studio teorico sono i grafi a griglia. Un grafo a griglia è definito come un grafo $n * m$ risultato di un prodotto cartesiano $P_n \square P_m$, dove P_n e P_m sono due grafi lineari rispettivamente di n e m vertici. Un grafo così definito ha $n \cdot m$ vertici e $2m \cdot n - m - n$ archi.

Un'ultima tipologia di grafi utilizzati è stata quella dei cosiddetti grafi del Re. Questo particolare tipo di grafi rappresentano tutte le possibili mosse della pedina del Re nel gioco degli scacchi. In un grafo simile ogni vertice rappresenta un quadrato

della scacchiera e ogni arco è un movimento della pedina del Re. Per un grafo di questa forma delle dimensioni di $n * m$ si ha un totale di $n \cdot m$ vertici e un totale di $4n \cdot m - 3(n + m) + 2$ archi.

Nella maggior parte dei casi si è scelto di rappresentare i grafi tramite liste di adiacenza e di memorizzarli e leggerli tramite file di testo opportunamente formattati. Nello specifico è stato scelto il formato denominato *gdl* che permette anche la visualizzazione a schermo tramite un software dedicato che minimizza le intersezioni in fase di disegno del grafo [70], [73]. Sotto stati utilizzati anche altri formati come il *comma-separated values*(CSV) e il *graphML*, formato basato su *XML* che rappresenta un buon tentativo di creare un formato standard per lo scambio di dati rappresentati mediante grafi.

1.4 Panoramica degli argomenti trattati

Nella prima parte del secondo capitolo, viene presentato un metodo per individuare una copertura per il problema della Dominazione Romana su grafi a griglia. In particolare, viene dimostrato che questa nuova copertura permette un miglioramento del limite superiore del numero di Dominazione Romana sui grafi a griglia, rispetto ai valori conosciuti. Inoltre, viene dimostrato che la differenza tra il limite superiore e quello inferiore risulta essere $(m + n + 2)/5 = \Theta(m + n)$, che è sub-lineare, dato il numero di vertici, $m \cdot n$, del grafo a griglia .

Poiché le metodologie di copertura proposte godono della proprietà di non ridondanza, cioè ogni vertice con valore romano 0 è coperto da esattamente un vertice con valore 2 e data la struttura regolare dei grafi a griglia, si ipotizza che, dato un grafo a griglia $G_{m,n}$, con $m, n \geq 5$, il limite superiore trovato è molto vicino al valore esatto del numero di Dominazione Romana del grafo, tanto da poter ipotizzare che:

$$\gamma_2^*(G_{m,n}) - 1 \leq \gamma_R(G_{m,n}) \leq \gamma_2^*(G_{m,n}).$$

Nella seconda parte del secondo capitolo è presentato un approccio al calcolo della Dominazione Romana su grafi bipartiti, che si basa sull'equivalenza del problema del vertex cover e il problema del matching, quest'ultimo computabile in tempo polinomiale nel caso di grafi bipartiti come mostrato in [31]. Si è utilizzato questo risultato per calcolare una Dominazione Romana consistente impiegando un tempo computazionale polinomiale per la specifica classi dei grafi bipartiti. L'approccio prevede di partire da un insieme di vertici che è uguale a quello del vertex cover, riducendolo ad una funzione di Dominazione Romana procedendo poi per raffinamenti successivi.

Nel terzo capitolo ci si pone l'obbiettivo di trovare un approccio generico che permetta di calcolare una buona approssimazione del Numero di Dominazione Romana su qualsiasi tipologia di grafo, senza fare nessuna supposizione sulla struttura del grafo che quindi non sarà usata come parametro durante la computazione. Viene così prodotta un'euristica che in tempo polinomiale produce una copertura romana e un numero di Dominazione Romana che rispetta i limiti teorici noti. In conclusione vengono proposte alcune varianti che migliorano il risultato ottenuto dall'euristica precedente a discapito della complessità computazionale.

Nel quarto capitolo viene esposta l'implementazione tramite piattaforma distribuita CUDA dell'approccio euristico che prevede punti iniziali di etichettatura multipli, trattata nell'ultima parte del capitolo precedente. Saranno esplorate diverse ottimizzazioni per le diverse caratteristiche di grafi in esame e per adattare l'approccio all'architettura stessa. Saranno analizzati i pro e i contro dell'architettura e esposti i risultati ottenuti correlati dai dati di utilizzo di memoria e spazio su disco.

Capitolo 2

Dominazione Romana su classi di grafi

Come si è visto nei paragrafi precedenti, il problema della Dominazione Romana è un problema NP-completo [10], quindi l'obbiettivo che ci si pone in questo capitolo è quello di trovare delle caratterizzazioni della funzione di Dominazione Romana per delle specifiche classi di grafi. In letteratura vengono esposti parecchi approcci che partendo dal presupposto di conoscere la struttura del grafo, riescono a calcolare un numero di Dominazione Romana buono per alcune classi di grafi specifici. Un esempio è mostrato in [50] dove si producono degli algoritmi efficienti per il calcolo del numero di Dominazione Romana su grafi intervallo e co-grafi.

Nella prima parte di questo capitolo, viene presentato un metodo per individuare una copertura per il problema della Dominazione Romana su grafi a griglia. In particolare, viene dimostrato che questa copertura permette un miglioramento del limite superiore sul numero della Dominazione Romana su grafi a griglia, rispetto ai valori conosciuti.

Inoltre, viene dimostrato che la differenza tra il limite superiore e quello inferiore risulta essere $(m + n + 2)/5 = \Theta(m + n)$, che è sub-lineare, dato il numero di vertici, $m \cdot n$, del grafo a griglia .

Poiché le metodologie di copertura proposte godono della proprietà di non

ridondanza, cioè ogni vertice con valore romano 0 è coperto da esattamente un vertice con valore 2 e data la struttura regolare dei grafi a griglia, si ipotizza che, dato un grafo a griglia $G_{m,n}$, con $m, n \geq 5$, il limite superiore trovato è molto vicino al valore esatto del numero di Dominazione Romana del grafo. Quest'argomento è oggetto dell'articolo *The Roman Domination Problem on Grid Graphs* presentato alla conferenza internazionale *Middle-European Conference on Applied Theoretical Computer Science* (MATCOS-13) [46].

Nella seconda parte del capitolo è presentato un approccio al calcolo della Dominazione Romana su grafi bipartiti, che si basa sull'equivalenza del problema del *vertex cover* e del *matching*, quest'ultimo computabile in tempo polinomiale nel caso dei grafi bipartiti. Si è provato a utilizzare questo risultato per calcolare una Dominazione Romana che basandosi su un insieme di vertici di partenza uguale a quello del vertex cover produca una copertura romana consistente impiegando un tempo computazionale polinomiale su grafi bipartiti.

2.1 Caratteristiche di un grafo a griglia

Una classe speciale di grafi non orientati è individuata dalla classe di grafi a griglia. Si denota un grafo a griglia come un grafo formato da m righe e n colonne e viene indicato con $G_{m,n}$. In [81] un grafo a griglia è definito come un grafo risultato di un prodotto cartesiano $P_n \Box P_m$, dove P_n e P_m sono due grafi lineari rispettivamente di n e m vertici.

In altre parole un grafo a griglia è un grafo i cui vertici corrispondono ai punti di un piano con coordinate intere nella forma (i, j), con $0 \leq i \leq n-1$ e $0 \leq j \leq m-1$, dove due vertici sono connessi tra di loro da un arco se i corrispondenti punti che li identificano sono a distanza 1. La Figura 2.1 mostra un grafo a griglia del tipo $G_{5,5}$.

Ogni vertice di un grafo a griglia è connesso al massimo a 4 vertici, cioè i vertici di coordinate $(i, j-1)$, $(i, j+1)$, $(i-1, j)$ e $(i+1, j)$. Ovviamente i vertici che stanno agli angoli del grafo sono connessi con solo 2 vertici mentre i vertici ai bordi (ma non nell'angolo) sono connessi a 3 vertici.

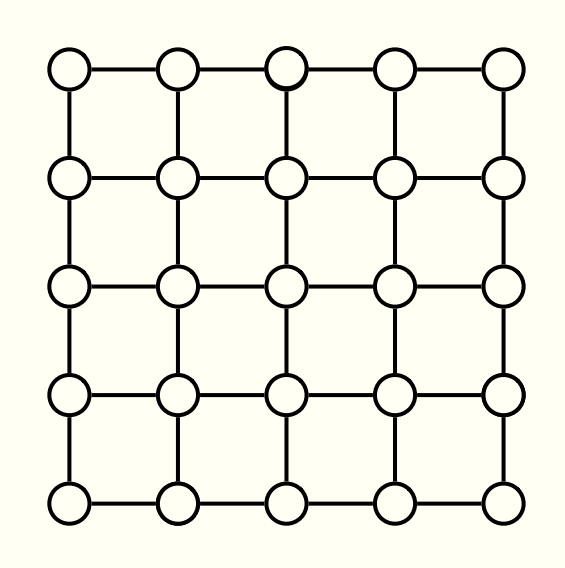


Figura 2.1: Grafo a griglia $G_{5,5}$.

Per convenzione, le coordinate del vertice all'angolo in alto a sinistra sono $(0, 0)$ e quelle del vertice all'angolo in basso a destra del grafo $G_{m,n}$ sono $(m-1, n-1)$.

Il problema di trovare una Dominazione Romana per grafi a griglia è stato formalizzato per la prima volta in [10] e poi in [39] e recentemente ripreso in [81]. In [18] è mostrata una caratterizzazione del problema della Dominazione Romana per grafi a griglia$G_{2,n}$ riportando quanto segue:

$$\gamma_R(G_{2,n}) = n + 1. \tag{2.1}$$

Viene lasciato anche un problema aperto, cioè la caratterizzazione del problema per grafi a griglia $G_{m,n}$, con $m, n \geq 3$. Particolare rilevanza ha il lavoro [10] dove gli autori propongono un metodo di copertura per i grafi a griglia del tipo $G_{3,n}$ e $G_{4,n}$. In [81] viene mostrato un metodo di copertura per i grafi a griglia del tipo $G_{k,n}$ con $5 \leq k \leq 8$ e $n \geq 5$ diversificando il metodo di copertura proposto a secondo di k. In [39], per grafi a griglia del tipo $G_{m,n}$ con $m, n \geq 5$, viene proposto il seguente limite superiore:

$$\gamma_R(G_{m,n}) \leq 2 \cdot \left(\left\lceil \frac{m \cdot n}{5} \right\rceil + \left\lceil \frac{m}{5} \right\rceil + \left\lceil \frac{n}{5} \right\rceil \right) .$$

Sempre in [39] viene dimostrato che per ogni grafo $G \neq \overline{K_n}$ con $|V| = n$, si ha il seguente limite inferiore:

$$\gamma_R(G) \geq \left\lceil \frac{2 \cdot n}{\Delta(G) + 1} \right\rceil .$$

Se si applica questo limite inferiore ad un grafo a griglia $G_{m,n}$ con $m, n \geq 3$, si ottiene $|V| = m \cdot n$ e $\Delta(G_{m,n}) = 4$ e risulta essere:

$$\gamma_R(G_{m,n}) \geq \left\lceil \frac{2 \cdot (m \cdot n)}{4 + 1} \right\rceil = \left\lceil \frac{2 \cdot (m \cdot n)}{5} \right\rceil .$$

2.2 Schemi di copertura per un grafo a griglia

Per i grafi a griglia sono stati individuati 5 diversi schemi di copertura che, come si vedrà nella sezione successiva, portano ad un numero di Dominazione Romana migliore del limite superiore noto. I diversi schemi hanno lo stesso meccanismo di applicazione e si diversificano principalmente per la posizione iniziale del primo valore romano 2 posizionato. Gli schemi, indicati con S_i per $i = 0, \ldots, 4$, avranno ognuno un valore romano 2 nella posizione $(0, i)$ e si applicano nella seguente maniera:

- Si sceglie il vertice del grafo dove posizionare il valore romano 2.
- Si assegna il valore romano 0 a tutti i vertici connessi al vertice con valore romano 2.
- Muovendosi sulla griglia in senso orario ma come se fosse una scacchiera ed eseguendo i movimenti legali della pedina del Cavallo nel gioco degli Scacchi, si individuano gli altri vertici a cui assegnare il valore romano 2.
- Si assegna il valore romano 0 ai vicini e si ripete fino a quando è possibile il punto precedente.
- Si assegna il valore romano 1 ai vertici rimanenti.

$i \mod 5$	S_0 $j \mod 5$	S_1 $j \mod 5$	S_2 $j \mod 5$	S_3 $j \mod 5$	S_4 $j \mod 5$
0	0	1	2	3	4
1	3	4	0	1	2
2	1	2	3	4	0
3	4	0	1	2	3
4	2	3	4	0	1

Tabella 2.1: Regole di posizionamento del valore romano 2 su grafo a griglia.

La procedura appena presentata si prende cura solamente di fissare le posizioni dei vertici a cui assegnare il valore romano 2, in quando una volta noto l'insieme V_2 è banale determinare V_0 e V_1 di conseguenza. La Tabella 2.1 mostra come vengono posizionati i valori romani 2 dal punto di vista matematico in base allo schema adottato.

Il primo schema, denominato S_0 è stato introdotto in [39], dove per griglie del tipo $G_{m,n}$, con $m, n \geq 5$ viene provato il seguente risultato:

Teorema 2. *Dato un grafo a griglia $G_{m,n}$, con $m, n \geq 5$, si ha:*

$$\gamma_R(G_{m,n}) \leq 2 \cdot \left(\left\lceil \frac{m \cdot n}{5} \right\rceil + \left\lceil \frac{m}{5} \right\rceil + \left\lceil \frac{n}{5} \right\rceil \right) .$$

Il modello proposto viene ripetuto su tutto il grafo a griglia, con la condizione che esso abbia un numero di righe o colonne maggiore di 5. Gli altri quattro schemi vengono introdotti in [46] e sono in grado di restituire risultati migliori rispetto a S_0. Questi nuovi schemi possono essere ricavati a partire da S_0. Per ottenere lo schema generico S_i partendo proprio da S_0, si spostino in ogni riga i valori romani 2 di i posizioni, per $i = 1, 2, 3, 4$.

Tale modalità di copertura garantisce che a nessun vertice interno sia assegnato un valore romano 1 e quindi si può affermare il fatto che segue.

Proposizione 10. *Sia $G_{m,n}$ un grafo a griglia. Se la funzione di Dominazione Romana per $G_{m,n}$ è ottenuta usando uno degli schemi $S_0, \cdots, S_4$, allora ogni vertice interno appartiene a $V_0 \cup V_2$.*

2.3 Miglioramento dei limiti

La struttura regolare di un a grafo a griglia rende possibile la seguente uguaglianza tra il grafo $G_{m,n}$ e il suo trasposto $G_{n,m}$:

$$\gamma_R(G_{m,n}) = \gamma_R(G_{n,m}).$$

in ogni caso per chiarezza, si definisce $\gamma_2^*(G_{m,n})$ come:

$$\gamma_2^*(G_{m,n}) = \min\{\gamma_2(G_{m,n}), \gamma_2(G_{n,m})\}.$$

Per quanto riguarda il metodo di copertura per grafi del tipo $G_{5,n}$ in [81] si dice che:

$$\gamma_R(G_{5,n}) = \begin{cases} 8 & n = 3 \\ \\ \left\lfloor \frac{12n}{5} \right\rfloor + 2 & \text{altrimenti} \end{cases} \tag{2.2}$$

Dagli argomenti trattati sempre in [81] si può dedurre che:

Lemma 3. *Sia $G = C_5 \square P_n$ con $n \geq 5$ il prodotto Cartesiano tra un grafo a ciclo con 5 archi e un grafo lineare con $n-1$ archi. Allora G può essere coperto da un qualsiasi schema S_i, e quindi si ha:*

$$\gamma_i(C_5 \square P_n) = \gamma_i(P_n \square C_5) = \gamma_R(C_5 \square P_n) = 2 \cdot n + 2$$

Il seguente Teorema, oggetto del lavoro di ricerca e argomento della pubblicazione [46], migliora il limite superiore mostrato nel Teorema 2.

Teorema 4 (Teorema delle Griglie)**.** *Sia $G_{m,n}$ un grafo a griglia, con $m, n \geq 5$.*

Grafi a Griglia	RDN γ_R	S_2 γ_2	S_2 γ_2^*	Teorema delle griglie	Teorema 2
5x5	**14**	**14**	**14**	**14**	14
5x6	**16**	**16**	**16**	**16**	18
5x7	**18**	**18**	**18**	**18**	20
5x8	**21**	**21**	**21**	**21**	22
5x9	**23**	**23**	**23**	**23**	24
6x5	**16**	**16**	**16**	**16**	18
6x6	**19**	**19**	**19**	**19**	24
6x7	**22**	**22**	**22**	**22**	26
6x8	**24**	25	**24**	**24**	28
6x9	**27**	**27**	**27**	**27**	30
7x5	**18**	19	**18**	**18**	20
7x6	**22**	**22**	**22**	**22**	26
7x7	**24**	25	25	25	28
7x8	**28**	**28**	**28**	**28**	32
7x9	**31**	**31**	**31**	**31**	34
8x5	**21**	**21**	**21**	**21**	22
8x6	**24**	**24**	**24**	**24**	28
8x7	**28**	**28**	**28**	**28**	32
8x8	**32**	**32**	**32**	**32**	34
8x9	**35**	**35**	**35**	**35**	38
9x5	**23**	**23**	**23**	**23**	24
9x6	**27**	**27**	**27**	**27**	30
9x7	**31**	**31**	**31**	**31**	34
9x8	**35**	**35**	**35**	**35**	38
9x9	n.d.	**38**	**38**	**38**	42

Tabella 2.2: Confronto tra il valore ottimo di γ_R, dato dall'equazione di pagina 19, lo schema S_2, il Teorema delle Griglie e il Teorema 2.

Allora usando lo schema S_2 si ha:

$$\gamma_R(G_{m,n}) \leq \gamma_2^*(G_{m,n}) = \begin{cases} \left\lfloor \frac{2 \cdot (m \cdot n + m + n)}{5} \right\rfloor - 1 & \textit{se} \quad m, n \mod 5 = 4 \\ \\ \left\lfloor \frac{2 \cdot (m \cdot n + m + n)}{5} \right\rfloor & \textit{altrimenti.} \end{cases} \tag{2.3}$$

Ponendo la formula sotto una forma più semplice si ha:

$$\gamma_2^*(G_{m,n}) \leq \left\lfloor \frac{2 \cdot (m \cdot n + m + n)}{5} \right\rfloor \leq 2 \cdot \left(\left\lceil \frac{m \cdot n}{5} \right\rceil + \left\lceil \frac{m}{5} \right\rceil + \left\lceil \frac{n}{5} \right\rceil \right) .$$

A sostegno di questa tesi nella tesi di Dottorato *The Roman Domination Problem on Grid Graphs* del Dott. V. Currò [7] vengono dimostrati i seguenti fatti:

Lemma 5. *Il Teorema delle Griglie è vero per grafi a griglia $G_{m,n}$ con $5 \leq m \leq 9$ e $n \geq 10$.*

Lemma 6. *Per un grafo a griglia $G_{m,n}$ con $5 \leq m \leq 8$ e $n \geq 5$, con l'eccezione del grafo $G_{7,7}$, il Teorema delle Griglie restituisce un valore di $\gamma_R(G_{m,n})$ uguale a quello ottenuto usando l'equazione di pagina 19.*

Inoltre dall'osservazione sperimentale si può dedurre che il Teorema delle Griglie è vero per grafi $G_{m,n}$ con $5 \leq m, n \leq 9$ e la dimostrazione risulta essere immediata osservando la Tabella 2.2.

Al fine di migliorare il limite inferiore conosciuto, si introduce un nuovo concetto associato ad ogni singolo vertice. Si parlerà quindi di *costo* di copertura di un vertice indicandolo con il simbolo $\chi(v)$. In un dato grafo a griglia, per ogni vertice tale grandezza rappresenta quanto si spende per proteggere quel vertice data una funzione di Dominazione Romana.

Per esempio, banalmente se un vertice v è etichettato con valore romano 1, allora il suo costo risulta essere 1.

$$\chi(v) = 1.$$

Se invece un vertice v è etichettato con valore romano 2, allora il suo costo è ugualmente condiviso e ammortizzato da se stesso e dai suoi vicini. In questo modo il costo condiviso, indicato con σ, è definito come il valore:

$$\sigma(v, w) = \frac{2}{N[v]} \qquad \forall w \in N[v].$$

Infine se un vertice v è etichettato con valore romano 0, allora si ha che:

$$\sigma(v, w) = 0 \qquad \forall w \in N[v].$$

Da queste affermazioni si può dedurre che se un vertice v è etichettato con valore romano 0 o 2, allora il suo costo è la somma di tutte le condivisione dei vertici del vicinato, incluso se stesso, cioè è uguale al valore calcolato dalla seguente formula:

$$\chi(v) = \sum_{w \in N[v]} \sigma(w, v).$$

Chiaramente, la somma di tutti i costi sarà uguale al numero di Dominazione Romana, cioè:

$$\sum_{v \in V} \chi(v) = \sum_{v \in V} f_{RD}(v).$$

Nel caso specifico dei grafi a griglia, si vogliono suddividere i vertici in due sottoinsiemi in base alla loro posizione sulla griglia stessa: A è l'insieme dei vertici interni e B è l'insieme dei vertici sui bordi o frontiera. I vertici in A hanno un costo di almeno 2/5. Infatti, nel migliore dei casi, si ha un vertice v con valore romano assegnato pari a 2 e i suoi quattro vicini con valore romano 0 e coperti solo da v. Per i vertici in B, il problema di trovare il costo è un po' più complicato. Ogni vertice sui bordi potrebbe anch'esso avere un costo di almeno 2/5, ed è il caso nel quale il vertice esterno è coperto solo da un vertice interno etichettato con il valore romano 2.

Quindi considerando questi fatti il limite inferiore per un grafo a griglia al momento risulta essere:

$$\gamma_R \geq \frac{2}{5} \cdot |A| + \frac{2}{5} \cdot |B| = \frac{2}{5} \cdot |V| = \left\lceil \frac{2 \cdot (m \cdot n)}{5} \right\rceil.$$

Come si può osservare dall'analisi delle configurazioni che si possono venire a creare mostrate in 2.2, non tutti i vertici in B hanno un costo di copertura pari a 2/5. Partendo da questa analisi, in [7] viene definito B^* come l'insieme di vertici che hanno un costo di almeno 2/4:

$$B^* = \{v \in B \colon \chi(v) < 2/4\}.$$

E supponendo di usare una funzione di Dominazione Romana che generi un insieme

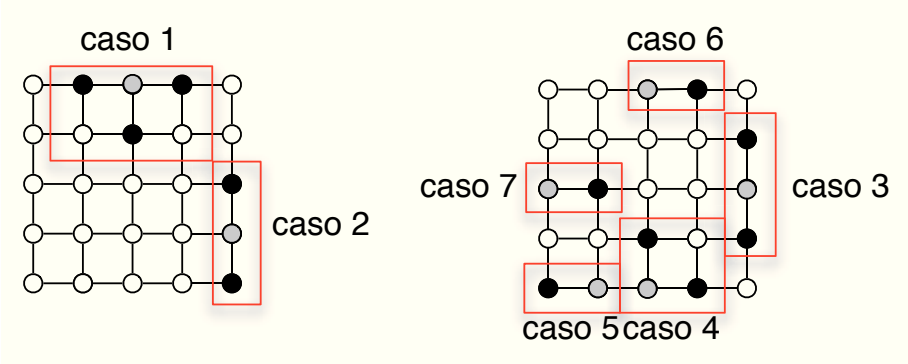


Figura 2.2: Distribuzione del costo di copertura di un vertice sul bordo.

V_1 minimale, cioè dove nessun vertice in V_1 abbia un vicino in V_2 si dimostra il seguente Lemma.

Lemma 7. *Sia $G_{m,n}$ un grafo a griglia con copertura romana dove V_1 sia minimale, allora esiste una funzione iniettiva $f\colon B^* \to A \cup B$, tale che $\forall v \in B^*$:*

$$\chi(v) + \chi(f(v)) \geq \begin{cases} 2/5 + 2/4 & se \quad f(v) \in A \\ 2/4 + 2/4 & se \quad f(v) \in B \end{cases} . \tag{2.4}$$

Riassumendo i risultati ottenuti si ha:

$$\gamma_R(G_{m,n}) \geq \frac{|A| \cdot 2}{5} + \frac{|B| \cdot 2}{4} = \left\lceil \frac{(2 \cdot m \cdot n + m + n - 2)}{5} \right\rceil > \left\lceil \frac{2 \cdot (m \cdot n)}{5} \right\rceil .$$

Cioè il limite inferiore calcolato tenendo conto dei costi di ogni singolo vertice nella griglia risulta essere maggiore, e quindi più accurato, del limite inferiore noto. In conclusione ricordando il Teorema 4 sulle Griglie si ha che:

$$\gamma_R(G_{m,n}) \leq \left\lfloor \frac{2 \cdot (m \cdot n + m + n)}{5} \right\rfloor .$$

Quindi unendo i risultati ottenuti si ha che il numero di Dominazione Romana per un grafo a griglia è limitato come segue:

$$\left\lceil \frac{(2 \cdot m \cdot n + m + n - 2)}{5} \right\rceil \leq \gamma_R(G_{m,n}) \leq \left\lfloor \frac{2 \cdot (m \cdot n + m + n)}{5} \right\rfloor . \tag{2.5}$$

La distanza risulta essere $(m+n+2)/5 = \Theta(m+n)$, la quale risulta essere sub-lineare, dati il numero dei vertici $m \cdot n$ del grafo a griglia.

2.4 Equivalenza tra vertex cover e il problema del matching per grafi bipartiti

Anche per quanto riguarda i grafi bipartiti il problema della Dominazione Romana risulta essere NP-completo. Se ne da un accenno in [10] e in più esistono molti lavori che riportano la caratteristica di NP-completezza anche per il problema generico della Dominazione come per esempio in [35]. L'idea è quella di aggirare l'ostacolo sfruttando l'equivalenza tra particolari problemi su questa specifica classe di grafi.

Nella teoria dei grafi, un *vertex cover* di un grafo è rappresentato da un insieme di vertici tali che ogni arco del grafo sia incidente ad almeno un vertice dell'insieme. Più formalmente si definisce C un vertex cover per il grafo $G = (V, E)$ se per ogni arco $\{u, v\} \in E$ è vera una delle due seguenti affermazioni:

- $u \in C$;
- $v \in C$.

Un vertex cover è detto minimo se non è possibile determinarne un altro di cardinalità minore e il problema del Vertex Cover Minimo consiste nel trovare un vertex cover per G di cardinalità minima. Si può vedere un esempio di vertex cover per un grafo G in Figura 2.3

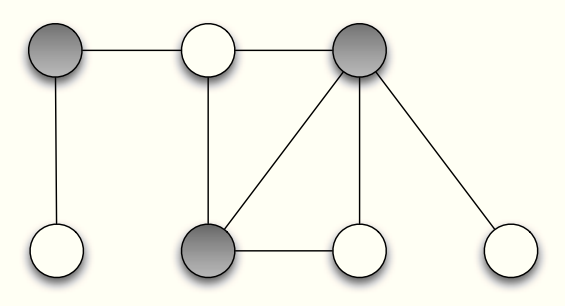


Figura 2.3: Vertex cover di un grafo.

Il problema di trovare il vertex cover minimo è un problema classico dell'ottimizzazione combinatoria in quanto si tratta di un problema NP-completo ben studiato nella teoria della complessità ripreso per esempio in [59]. Il problema di trovare un vertex cover è strettamente correlato al problema di trovare un insieme indipendente massimale. Infatti risulta essere che un insieme di vertici è un vertex cover se e solo se il suo complemento $\bar{C} = V \setminus C$ rappresenta un insieme indipendente. Da ciò si deduce che un grafo con n vertici avrà un vertex cover di cardinalità k se e solo se il grafo ha in insieme indipendente di cardinalità $n - k$. Quindi i due problemi risultano essere duali.

Nonostante il vertex cover sia un problema NP-completo, un algoritmo *brute force* (cioè un algoritmo capace di provare tutte le possibili soluzioni) può risolverlo in tempo $2^{O(K)} * n^{O(1)}$. Ciò significa che è un problema trattabile a parametro fisso, e se si è interessati solo a piccoli valori di k, si può risolvere il problema in tempo polinomiale. La strategia dell'algoritmo brute force è di scegliere un vertice e fare salti ricorsivi, con due casi ad ogni passo: inserire nell'insieme che andrà a formare il vertex cover il vertice corrente o tutti i vertici ad esso adiacenti ed è esposto in [79].

Nella teoria dei grafi, il *matching* di un grafo è un insieme di archi tali che essi non condividono nessun vertice tra di loro. In altre parole si tratta di un insieme di archi in cui è impossibile trovarne due che condividono uno stesso endpoint (punto di incidenza in un vertice). Un matching massimale è un insieme M di un grafo G tale che se un arco che non è già in M viene aggiunto ad M, allora M non sarà più un matching. In altre parole, un matching M di un grafo G è massimale se ogni arco di G ha un intersezione non vuota con almeno un arco in M. Un matching massimo (cioè un insieme di matching con la cardinalità massima possibile) è l'insieme che contiene il maggiore numero possibile di archi. Possono esserci diversi insiemi di matching con la stessa cardinalità e questa cardinalità viene indicata come il *matching number* di un grafo. Da sottolineare che ogni matching massimo è massimale ma non è vero il viceversa. Ma se un insieme risulta essere sia massimale che massimo allora lo si usa denotare come matching perfetto [15].

Il teorema di König, fornito da Dénes König in [31] e rivisitato recentemente anche

in [12] , è conosciuto per aver presentato l'equivalenza tra il problema del matching e quello del vertex cover per quanto riguarda i grafi bipartiti. Per i grafi che non sono bipartiti, la complessità della risoluzione dei due problemi è molto diversa. Infatti il problema del matching è risolvibile in tempo polinomiale per qualsiasi classe di grafi mentre il vertex cover nella sua formalizzazione generale è un problema NP-completo.

Quindi l'equivalenza dimostrata da König permette di risolvere il problema del vertex cover su dei grafi bipartiti in tempo polinomiale nonostante la NP-completezza del problema generale se applicato ad altre classi di grafi [40]. Il teorema si basa sull'asserzione che in un qualsiasi grafo bipartito, il numero di archi presente in un matching massimo è uguale al numero di vertici in un vertex cover minimo. Da sottolineare, come viene fatto in [66] che sebbene i due problemi siano equivalenti, non esiste la stessa equivalenza per gli algoritmi di approssimazione. Infatti l'algoritmo per trovare il matching massimo di un grafo bipartito può essere approssimato in tempo costante da un algoritmo distribuito. A differenza del calcolo approssimato di un vertex cover minimo di un grafo bipartito che richiede almeno tempo logaritmico.

2.4.1 Algoritmo e Analisi

La dimostrazione del teorema fornisce anche la struttura e la possibilità di costruire un algoritmo che permetta il calcolo effettivo del matching e quindi del vertex cover nel caso di grafi bipartiti. Tuttavia l'algoritmo di Hopcroft-Karp [29] è usato più frequentemente per risolvere il problema in maniera efficiente, in quanto è meno difficile da comprendere e implementare e in questo lavoro è stato utilizzato proprio quest'ultimo. Tale algoritmo, dato $G(V, E)$ tale che G sia bipartito, è in grado di produrre in tempo $O(|E|\sqrt{|V|})$ un matching di cardinalità massima.

Vediamone nel dettaglio l'approccio utilizzato, dando in fase preliminare alcune utili definizioni. Un vertice che non è un *endpoint* per nessun matching parziale M è detto *vertice libero*. Si definisce *augmenting path* un cammino che parte e finisce in un vertice libero, e si alterna durante il percorso tra un vertice che fa parte del matching e un vertice che non ne fa parte. Se M è un insieme di matching di cardinalità $|V|$ e P

è un augmenting path associato a M, allora la differenza simmetrica dei due insieme di archi, $M \bigoplus P$, costituisce un matching di dimensioni $|V| + 1$. Quindi ad ogni passo la cardinalità dell'insieme di matching aumenta fino a trovare la cardinalità massima possibile. Supponendo per assurdo che $|M|$ non sia ottimale, si assegni a P la differenza simmetrica $M \bigoplus M^*$ dove M^* è un matching ottimale. Allora P deve formare una collezione di augmenting path disgiunti cioè cicli e percorsi nei quali sono presenti un numero uguale di archi che hanno un matching e quelli che non ce l'hanno. La differenza di dimensioni tra M e M^* rappresenta il numero di augmenting path in P. In questo modo, se nessun augmenting path può essere trovato, l'algoritmo finisce dato che in questo caso M deve essere già ottimale.

Siano U e V i due insiemi di un grafo bipartito e M il matching da U a V. L'algoritmo si sviluppa in diverse fasi e per ogni fase vengono eseguite le seguenti operazioni:

- Una visita in ampiezza partiziona i vertici del grafo in livelli. I vertici liberi in U sono utilizzati come punto di partenza e formano il primo livello della partizione. Al primo livello della ricerca, solo gli archi senza matching possono essere attraversati. Nei livelli successivi della ricerca, per attraversare qualsiasi arco è richiesto l'alternarsi tra archi con matching e archi senza. Quindi quando si cerca un successore partendo da un vertice in U, verranno attraversati solo archi senza matching, mentre nel caso di un vertice in V, saranno attraversati solo archi con matching. La ricerca si interrompe ad un qualsiasi livello k in cui uno o più vertici in V vengono raggiunti.

- Tutti i vertici liberi in V al livello k sono posizionati in un insieme F. In pratica, un vertice v è posizionato in F se e solo se è l'estremità finale di un augmenting path.

- Arrivati a questa fase l'algoritmo ha a disposizione un insieme massimale di vertici disgiunti che formano un augmenting path di lunghezza k. Questo insieme può essere elaborato da una ricerca in profondità da F ai vertici liberi di U, usando il livello generato al primo passo dalla visita in ampiezza per farsi

guidare nella ricerca. La ricerca in profondità è permessa solo se si seguono archi che portano ad un vertice non utilizzato nel livello precedente. Inoltre i percorsi nell'albero generato dalla ricerca in profondità devono alternarsi tra archi che hanno un matching e archi che non ce l'hanno. Nel momento in cui un augmenting path che coinvolge un vertice in F è stato trovato, la ricerca in profondità prosegue con il vertice successivo.

- Ogni percorso trovato in questo modo viene utilizzato per espandere M.

L'algoritmo termina quando nella fase di ricerca in ampiezza del primo passo, non viene trovato nessun nuovo augmenting path.

Ogni singola fase consiste in una ricerca in profondità o in ampiezza ed entrambe possono essere implementate in tempo lineare. Questo vuol dire che le prime $\sqrt{|V|}$ fasi impiegano tempo $O(|E|\sqrt{|V|})$ in un grafo con $|V|$ vertici e $|E|$ archi. E' possibile dimostrare come viene fatto in [29], che ogni fase incrementa la lunghezza del più corto augmenting path di almeno un'unità. Infatti ogni fase trova un insieme massimale di augmenting path di una data lunghezza e questo implica che non può esisterne uno di lunghezza maggiore. In più, una volta che sono completate $\sqrt{|V|}$ fasi, gli augmenting path rimanenti devono essere composti da almeno $\sqrt{|V|}$ archi. Tuttavia la differenza simmetrica di un eventuale matching ottimale rispetto ad un matching parziale di M trovato nella prima fase, forma una collezione di augmenting path e cicli alternati di vertici disgiunti. Se ogni percorso della collezione ha lunghezza almeno $\sqrt{|V|}$, possono essere presenti al massimo $\sqrt{|V|}$ percorsi nella collezione e la dimensione del matching ottimale può differire dalla dimensione di $|M|$ di al massimo $\sqrt{|V|}$ archi. Siccome ogni fase incrementa la dimensione del matching di almeno un'unità, allora ci possono essere al massimo $\sqrt{|V|}$ fasi aggiuntive prima che l'algoritmo termini correttamente. Visto che l'algoritmo esegue un totale di al massimo $2\sqrt{|V|}$ passi, impiega un tempo massimo di $O(|E|\sqrt{|V|})$ nel caso peggiore. In verità pare che altri studi hanno dimostrato che l'algoritmo non si comporta così bene nella realtà come nella teoria ma continua ad essere comunemente utilizzato per la sua facilità di implementazione e comprensione.

2.5 Riduzione del Vertex Cover al problema della Dominazione

Come visto nella fase introduttiva, dato un grafo non orientato $G = (V, E)$, si definisce l'insieme $D \subseteq V$ come un insieme dominante se ogni $v \in V$ o fa parte di D oppure è adiacente ad almeno un elemento di D.

Dato un grafo $G = (V, E)$ che contiene un'istanza di vertex cover di dimensione k, si vuole convertire tale istanza in un'istanza per il problema dell'insieme dominante. La procedura di conversione prevede che per ogni arco $e = (u, v)$ del grafo G, si aggiungano un vertice S_{uv} e gli archi (u, s_{uv}) e (v, s_{uv}). In pratica si viene a creare un triangolo per ogni arco in G e si denota con $G^{'} = (V^{'}, E^{'})$ il grafo così ottenuto.

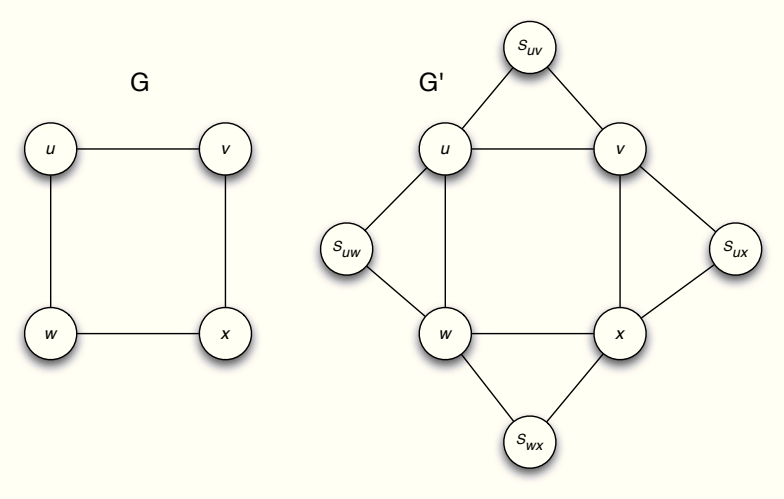


Figura 2.4: Vertex cover in G e insieme dominante in G'.

Dal grafo della Figura 2.4 si deduce che il suo vertex cover è l'insieme formato dai vertici (u, x), il quale è anche l'insieme dominante del grafo appena costruito $G^{'}$. Tuttavia è importante sottolineare, anche per il prosieguo dello svolgimento, che l'insieme $\{u, v\}$ in G rappresenta un insieme dominante ma non un vertex cover.

Proposizione 11. *Il vertex cover è un caso speciale dell'insieme dominante. Più formalmente, vertex cover* $\leq_p$ *insieme dominante. G ha un vertex cover di dimensione al massimo k se e solo se* $G^{'}$ *ha un insieme dominante di dimensione al massimo k.*

Dimostrazione. $\Rightarrow$ G ha un vertex cover di dimensioni al massimo $k \Rightarrow G'$ ha un insieme dominante di dimensione al massimo k.

Sia C un vertex cover di G di dimensione k. Siccome ogni arco in G è incidente in almeno un vertice in C, allora i vertici originali di G, i quali sono anche in G', sono dominati dagli stessi vertici in C. Per ogni arco $e = \{u, v\}$ uno tra v e w si trova in C. Questo significa che il vertice s_{uv} in G' è dominato da C. Questo comporta che tutti i nuovi vertici aggiunti in G' sono dominati e C è l'insieme dominante per G' che è esattamente quello che si voleva dimostrare.

$\Leftarrow$ G' ha un insieme dominante di dimensione al massimo $k \Rightarrow G$ ha un vertex cover di dimensioni al massimo k.

Sia D un insieme dominante di dimensioni k in G'. Possono presentarsi due casi distinti:

- D contiene solo vertici che provengono dal grafo originale G. In questo caso, tutti i nuovi vertici hanno un arco verso un vertice di D. Questo significa che D copre tutti gli archi ed è un vertex cover valido per G. Tuttavia, se esistono dei vertici nuovi in D, è necessario rimpiazzarli con un vertice di G. Quindi bisogna rimpiazzare un nuovo vertice s_{uv} presente in D con un vertice tra u e v.

- D contiene alcuni vertici che stanno anche in G'. In questo caso l'arco $e = \{u, v\}$ sarà coperto da D. Se u o v fanno già parte di D non è necessario aggiungerli. Infatti quando tutti i nuovi archi verranno rimossi, D sarà un vertex cover valido per G.

□

2.6 Dominazione romana su grafi bipartiti

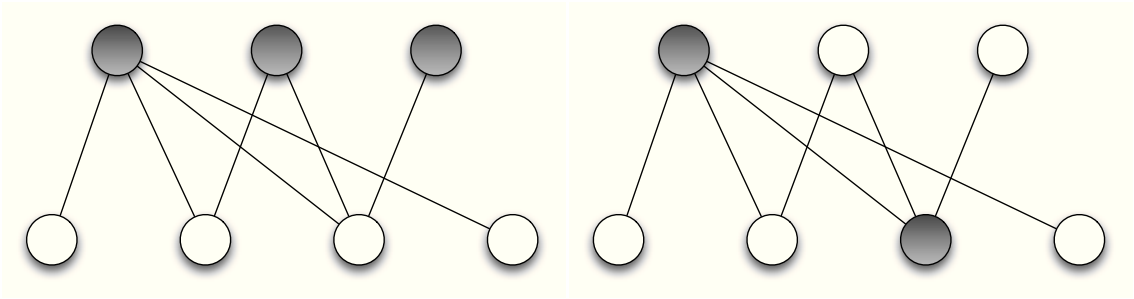
Sfruttando il fatto che il problema di trovare il matching di cardinalità massima su un grafo bipartito può essere calcolato in tempo polinomiale tramite l'algoritmo esposto nella sezione precedente, e considerando il teorema di König [31] che dimostra l'equivalenza tra questo problema e quello del vertex cover, si è pensato di applicarlo come base di partenza per produrre una funzione di Dominazione Romana per grafi bipartiti. Ricordando che il calcolo sia dell'insieme dominante che della Dominazione Romana è un problema NP-completo anche nel caso di grafi bipartiti, usando questo approccio si ha un insieme dominante che assicura una Dominazione Romana valida per il grafo impiegando tempo polinomiale. Quindi sia $G = (V, E)$ un grafo non diretto e sia C il vertex cover del grafo calcolato con l'algoritmo di König e quindi ottenuto in tempo polinomiale. L'idea di base è quella di assegnare inizialmente $C = V_2$ dove V_2 rappresenta l'insieme dei vertici del grafo a cui assegnare il valore romano 2 come risultato di una funzione di Dominazione Romana. Per costruzione C rappresenta una dominazione valida per G.

$$\gamma_R = 2|C|$$

Procedendo in questo modo si ha una configurazione valida ma non ottimale. Si può in ogni caso procedere a dei raffinamenti successivi dal costo computazionale polinomiale.

- Se esiste un vertice $v \in V_2$ tale che il vicinato di v è composto da $N(v) = \forall u, u \in V_2 : \{u, v\} \in E$ allora si assegni a v il valore romano 0 spostandolo così di fatto in V_0.
- Se esiste un vertice $v \in V_2$ tale che il suo vicinato $N(v) = u \in V_0 \cup V_2 : \{u, v\} \in E$ è formato solo da vertici u tali che $u \in V_0$ ed a sua volta il vicinato di u è composto da $N(u) = x \in V_2 : \{u, x\} \in E$ allora v può essere spostato in V_0 se il suo vicinato presenta almeno un vertice in V_2 altrimenti lo si può sposta in V_1.

Pur ottenendo una funzione di Dominazione Romana valida e consistente può accadere (come accennato durante la dimostrazione di riduzione del problema del vertex cover all'insieme dominante) che un insieme dominante non sia anche un vertex cover e in più si può verificare che la cardinalità minima dell'insieme domante sia più bassa di quella del vertex cover. Infatti dato un insieme dominante minimo D, in generale non esiste un vertex cover minimale C tale che $D \subseteq C$. La situazione è mostrata in Figura 2.5 dove viene mostrato un caso in cui $C \neq V_1 \cup V_2$. L'approccio porta comunque ad avere un numero di Dominazione Romana calcolato in tempo polinomiale che approssima il valore reale.



(a) Vertex Cover (b) Insieme Dominante

Figura 2.5: Confronto tra vertex cover e copertura romana di un grafo.

Capitolo 3

Algoritmi euristici per grafi generici

In questo capitolo ci si pone l'obbiettivo di trovare un approccio *greedy* generico capace di calcolare una buona approssimazione del numero di Dominazione Romana su qualsiasi tipologia di grafo, senza fare nessuna supposizione sulla struttura del grafo che quindi non sarà usata come parametro durante la computazione. Vengono esposte varie procedure, dalla più semplice che si basa sul grado dei vertici presenti nel grafo, all'introduzione di un nuovo parametro dinamico che permette di trovare un numero di Dominazione Romana migliore. Quest'ultima euristica produce un numero di Dominazione Romana coerente con i limiti teorici noti impiegando tempo polinomiale. Infine vengono proposte delle ottimizzazioni che rappresentano la base di partenza per l'implementazione tramite calcolo distribuito presentata nel capitolo successivo.

L'argomento dell'euristica greedy che fa uso di un parametro dinamico per calcolare il numero di Dominazione Romana è oggetto di un lavoro dal titolo *Heuristics for Roman Domination Problem* che è in fase di sottomissione [45].

3.1 Approccio greedy per la Dominazione romana

Un algoritmo *greedy* è un algoritmo euristico che costruisce la soluzione di un determinato problema adottando la soluzione migliore a livello locale nella speranza che questa scelta porti ad una buona soluzione a livello globale. Quindi solitamente una

strategia greedy non porta ad una soluzione ottima ma ad una buona approssimazione in un tempo computazionalmente accettabile. Gli algoritmi euristici descritti di seguito adottano questa strategia per il calcolo del numero di Dominazione Romana su grafi di qualsiasi classe.

Il primo approccio greedy usato per il problema su un generico grafo è stato quello di prendere in considerazione l'assegnazione del valore romano 2 per i vertici che hanno un grado maggiore rispetto agli altri. Quindi si prevede di assegnare di volta in volta il valore 2 ai vertici più connessi, aggiornando di conseguenza il valore romano dei vertici vicini a 0. Si tratta di una classica strategia greedy perché non fa nessuna considerazione sulla struttura globale del grafo e si limita a fare la scelta migliore nello specifico momento.

La procedura tiene conto della struttura locale di ogni singolo vertice ma non fa nessuna considerazione o supposizione sulla struttura globale del grafo. Questo anche perché si vuole sviluppare un approccio indipendente dalla struttura del grafo esaminato di volta in volta.

L'algoritmo euristico procede come segue. Come primo passo si esegue una procedura di inizializzazione che assegna un valore fittizio -1 ad ogni vertice. Questo valore, che come si può notare non è un valore consentito in una Dominazione Romana valida, serve ad indicare che il vertice non è stato ancora analizzato e quindi potrebbe risultare anche non coperto.

Il passo successivo consiste nel produrre un ordinamento decrescente dei vertici usando come parametro di ordinamento il loro grado. Questo parametro associato ai vertici non cambia durante la computazione e quindi può essere calcolato una sola volta e rimane invariato per tutta l'esecuzione.

Dopo questa fase di inizializzazione ha inizio una procedura di assegnazione che si occupa di prendere in esame ogni vertice v con valore -1 in ordine decrescente rispetto al grado, eseguendo su di esso le seguenti operazioni:

- Se v ha ancora nella sua lista di adiacenza almeno un vertice con valore -1, si assegna a v il valore romano 2. Inoltre si assegna il valore 0 a tutti i vertici

presenti nella sua lista di adiacenza che hanno valore -1.

- Se v non ha vertici adiacenti con valore assegnato -1 allora l'unica azione da fare è assegnare a v il valore romano 1.

Questa prima fase si chiude quando tutti i vertici del grafo vengono analizzati. Alla fine quindi, nessun vertice avrà valore assegnato -1 e si avrà una configurazione di Dominazione Romana valida. Una volta finita questa fase, si eseguono delle procedure di raffinamento che hanno l'obbiettivo di decrementare il numero di Dominazione Romana trovato con la sola procedura di assegnazione appena eseguita.

Questa procedura ha l'obbiettivo di decrementare la cardinalità dell'insieme V_1, quindi ridurre il numero di vertici a cui è stato assegnato il valore romano 1. La procedura di raffinamento consiste nel trovare eventuali vertici v a cui è stato assegnato il valore romano 0 e verificare se nella loro lista di adiacenza sono presenti due o più vertici etichettati con il valore romano 1. In questo caso, si etichetta con il valore romano 2 il vertice v e con il valore 0 tutti i vertici della sua lista di adiacenza che avevano un valore romano assegnato pari a 1.

La complessità di questa euristica è molto semplice da calcolare e risulta essere:

- la procedura di inizializzazione dei vertici prende tempo $O(V)$.
- l'ordinamento dei vertici secondo il loro grado prende tempo $O(V \cdot \log V)$.
- l'assegnamento e l'aggiornamento dei valori romani prende tempo $O(V \cdot E)$.
- la procedura di raffinamento prende tempo $O(V \cdot E)$.

Si può quindi concludere che l'intera algoritmo impiega un tempo di computazione pari a $O(V \cdot E)$, quindi polinomiale rispetto alle dimensioni del grafo. Come si vedrà nella prossima sezione, l'approccio porta a dei risultati largamente migliorabili tramite una procedura che utilizza un nuovo parametro per determinare l'ordine con cui estrarre i vertici da coprire.

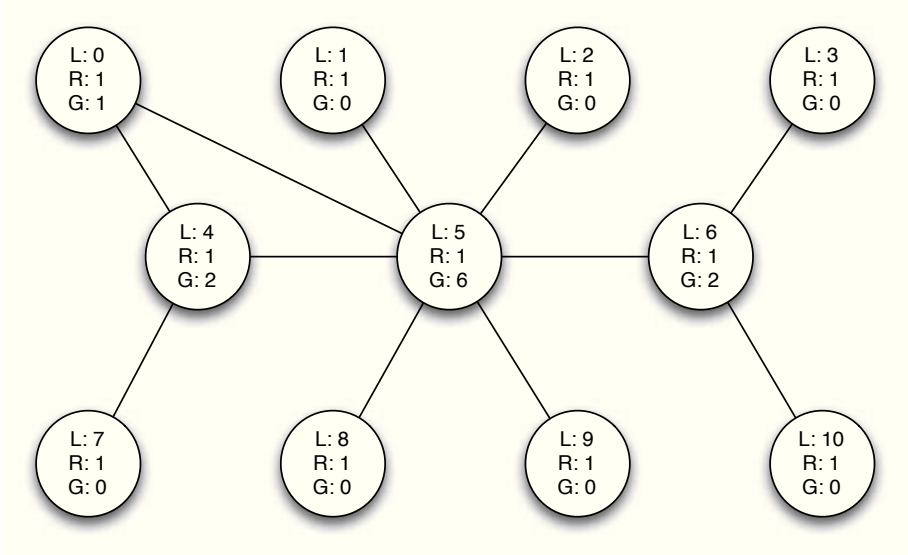


Figura 3.1: Esempio di GainFactor dei vertici di un grafo.

3.2 Utilizzo di un parametro dinamico: il GainFactor

Si vuole trovare un parametro che aiuti a migliorare i risultati ottenuti. A questo scopo si è definito un nuovo parametro dinamico per ogni vertice, denominato *GainFactor*, che rappresenta il guadagno che si ha supponendo di assegnare il valore romano 2 al vertice e aggiornando di conseguenza il vicinato del vertice in questione.

3.2.1 Caratteristiche del GainFactor

Formalmente si definisce GainFactor per un vertice v il valore così calcolato:

$$GF(v) = R(v) - 2 + \sum adj_1(v)$$

dove $R(v)$ è il valore romano corrente assegnato al vertice v e $\sum adj_1(v)$ è il numero totale dei vertici adiacenti a v che hanno assegnato il valore romano 1.

Nella Figura 3.1 per ogni vertice sono indicate le varie grandezze che vengono prese in considerazione in questo caso:

- L è l'etichetta che viene assegnata al vertice in fase di creazione e ne indica il nome e la posizione.
- R indica il valore romano attualmente assegnato al vertice.
- G rappresenta il valore attuale del GainFactor del vertice.

La configurazione mostrata in Figura 3.1 è quella che si presenta subito dopo la fase di inizializzazione e prima di iniziare la vera e propria computazione. Questo nuovo parametro, a differenza del grado dei vertici, cambia durante la computazione considerando che variano i valori romani assegnati ai vertici. A seguire alcune caratteristiche fondamentali del nuovo parametro.

Teorema 1. *Il GainFactor è una funzione decrescente quindi il GainFactor di un vertice può solo diminuire.*

Dimostrazione. Sia $G = (V, E)$ un grafo e v un vertice del grafo. La dimostrazione si riduce a verificare che il numero di vicini di tale vertice che hanno valore romano 1 può solamente diminuire. Infatti poiché il GainFactor di un vertice è dato dalla somma del numero dei suoi vicini con valore romano uguale a 1, dal proprio valore romano e da un valore costante, ciò implica che il primo componente, per costruzione, può solo diminuire e l'unico parametro che potrebbe far aumentare il valore di GainFactor di un vertice è la variazione del valore romano del vertice stesso. Tuttavia gli unici cambiamenti di valore romano verso uno più grande sono $0 \to 2$ e $1 \to 2$, ma tali cambiamenti avvengono solo quando un vertice viene selezionato come il vertice che in quel momento ha il GainFactor più alto. Ciò implica che il GainFactor di quel vertice deve essere positivo e che il numero dei suoi vicini posti ad 1 sarà azzerato, per cui anche il GainFactor sarà ricalcolato e sarà sicuramente minore o uguale al valore di partenza. Analizzando i due casi esposti si osserva che:

- $0 \to 2$: il vertice che ha valore romano 0 per avere un GainFactor positivo deve avere almeno 3 vicini con valore romano 1.
- $1 \to 2$: il vertice che ha valore romano 1 deve avere almeno 2 vicini con valore romano 1.

In entrambi i casi porre il valore romano del vertice a 2 comporta l'azzeramento del numero dei sui vicini con valore romano ad 1 e ciò comporta che ad ogni variazione il GainFactor diminuisce di almeno una unità, quindi possiamo affermare definitivamente che il GainFactor può solo diminuire o rimanere invariato. □

3.2.2 Conseguenze della variazione del GainFactor

Sia $G = (V, E)$ un grafo e sia v un vertice appartenente a tale grafo. Assegnare a v il valore romano 2 richiede l'aggiornamento del GainFactor di al massimo dei vertici contenuti nel vicinato di primo livello di v e dei vertici del vicinato di secondo livello di v più v stesso. Infatti come già detto, la variazione del GainFactor di un vertice può esser causata da due motivazioni, cioè la variazione dei vertici adiacenti che hanno valore romano 1 e il cambio di valore romano del vertice stesso.

Si definisce con $N_1(v)$ l'insieme dei vicini del vertice v, più formalmente $N_1(v) = e \in E : e = \{v, x\}$, mentre indichiamo con $N_2(v)$ l'insieme dei vicini dei vicini, oppure vicini di secondo grado, più formalmente $N_2(v) = e \in E : e = \{u, x\}, u \in N_1(v)$. Nel caso peggiore v ed i vertici appartenenti a $N_1(v)$ hanno tutti valore romano 1. Assegnare a v il valore romano 2 e porre il valore romano di tutti i vertici contenuti in $N_1(v)$ a 0 implica la necessità di aggiornare il GainFactor di v e di tutti i vertici presenti in $N_1(v)$ per il fatto di aver cambiato il valore romano ed i vertici facenti parte dell'insieme $N_2(v)$ perché per tali vertici il numero di vicini che avevano valore romano 1 è diminuito di almeno una unità. Le conseguenze di un'assegnazione di un valore romano 2 si ripercuotono sul GainFactor e sono mostrate nella Figura 3.2 dove il valore romano è riportato all'interno dei vertici e il GainFactor è il valore rappresentato fuori dal vertice.

Quindi riassumendo, l'aggiornamento del GainFactor dei vertici successivamente ad una fase di assegnazione del valore romano a 2 ad un vertice riguarda il vicinato di primo e di secondo livello. Ma questa affermazione può risultare ingannevole, infatti anche nel caso di una connettività di 0.1, l'insieme dei vertici da aggiornare potrebbe corrispondere ad una cospicua porzione dei vertici del grafo. Ciò significa

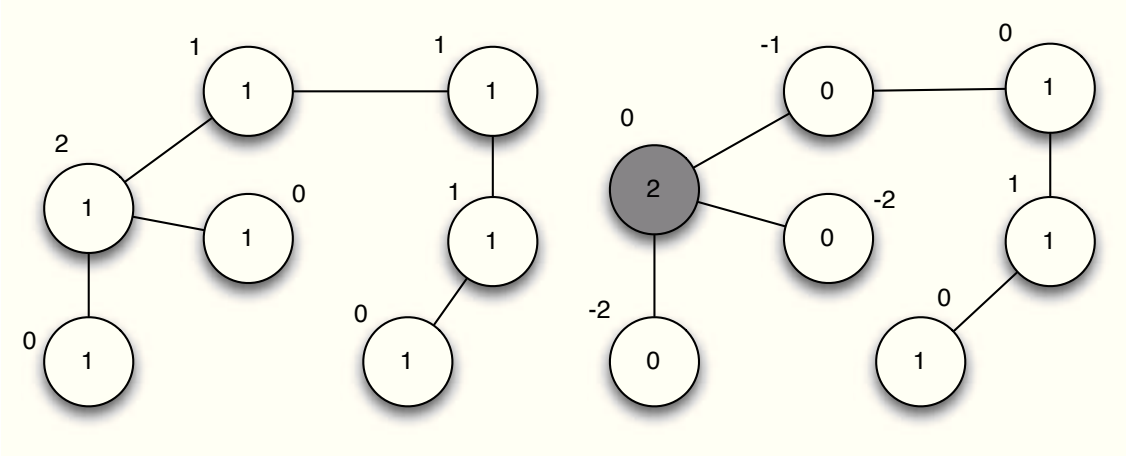


Figura 3.2: Variazione del GainFactor e conseguenze sul vicinato.

che per grafi con connettività maggiore o uguale a 0.1, investire risorse per la ricerca dei vertici per cui è necessario aggiornare il GainFactor in modo da rifare i calcoli solo su questo insieme di fatto non porta un guadagno ma può risultare addirittura controproducente per via del fatto che vanno individuati i due insiemi $N_1(v)$ e $N_2(v)$ che possono anche essere non disgiunti. Quindi nella maggior parte dei casi, una volta assegnato un nuovo valore romano ad un vertice v si preferisce ricalcolate il GainFactor di tutti i vertici del grafo nel caso la connettività risulti essere più alta di una certa soglia. Più precisamente si ricalcola il GainFactor di tutti quei vertici che avevano un valore maggiore di zero al passaggio precedente. Questo ricordando che il GainFactor non può aumentare e che una volta arrivato a zero, ci rimarrà per sempre.

3.3 Nuova euristica con il GainFactor

Si vuole utilizzare il parametro appena formalizzato per il calcolo del numero di Dominazione Romana di un qualsiasi grafo. A differenza della precedente euristica che utilizza il grado di ogni vertice e procede tenendo il considerazione un ordinamento di tale valore, questa nuova euristica vuole sfruttare il valore di GainFactor associato ad ogni vertice per determinare a quali vertici del grafo associare il valore romano 2. L'idea è quella di utilizzare il parametro per guidare la procedura di copertura

del grafo, selezionando di volta in volta i vertici con GainFactor maggiore. Prima di iniziare, si esegue una procedura di inizializzazione che assegna il valore romano 1 ad ogni singolo vertice. Questa operazione permette di avere una Dominazione Romana valida in ogni momento dell'esecuzione dell'euristica.

Come secondo passo, una volta calcolato il GainFactor per ogni vertice, con la modalità vista nel paragrafo precedente, viene creata in memoria una lista ausiliaria L_G che contiene tutti i vertici con GainFactor maggiore di zero. La lista conterrà in ogni momento tutti i vertici ancora da analizzare.

Ad ogni iterazione, si estrae da L_G il vertice v con il GainFactor massimo, si assegna ad esso il valore romano 2 ed ai suoi vicini che hanno valore romano 1 (i vertici presenti nella sua lista di adiacenza) si assegna il valore romano 0.

Per ogni iterazione vengono cancellati da L_G il vertice appena estratto e tutti gli eventuali vertici che risultano avere un GainFactor uguale a 0. Dopo questa operazione, è necessario aggiornare il valore di GainFactor di ogni vertice coinvolto (o di tutti i vertici, a secondo della scelta implementativa che si è fatta). Le iterazioni continuano fino a quando è presente almeno un vertice con GainFactor maggiore di 0, cioè fino a quando L_G non è vuota.

La fase centrale dell'euristica è così completata e si esegue una procedura raffinamento del risultato che si prefigge l'obbiettivo di ridurre al minimo possibile il numero di vertici con valore romano 1, cioè ridurre la cardinalità di V_1. Si posso presentare due casi diversi:

- Nel primo, si ha un vertice con il valore 1 adiacente ad un'altro (o anche più di uno) sempre con il valore romano 1. In questo caso, si può arbitrariamente assegnare il valore romano 2 ad uno di esso ed assegnare ai rimanenti il valore romano 0.

- Nella secondo caso, si ha un vertice v con il valore romano pari a 0 e due vertici u e t nella propria lista di adiacenza con valore romano 1. In questo caso si assegna il valore romano 2 al vertice v mentre ai vertici u e t viene assegnato il valore romano 0.

Algoritmo 1 Pseudo-codice dell'euristica che fa uso del GainFactor

```
1: GainFactorHeuristic(G)
2: R(V, 1)
3: GF(V)
4: LG(v.gainfactor ≥ 0)
5: while L_G ≠ do
6:    max ← 0
7:    pos ← nil
8:    for all v ∈ L_G do
9:       if v.gainfactor ≥ 0 then
10:         max ← v.gainfactor
11:         pos ← v
12:      else if v.gainFactor ≤ 0 then
13:         REMOVE(v)
14:      end if
15:      if max ≥ 0 then
16:         UpdateGF(v)
17:      end if
18:   end for
19: end while
20: Minimize(G)
21: return SUM(R(V))
```

3.3.1 Considerazioni sulla complessità

Lo pseudo-codice dell'algoritmo appena esposto è mostrato nel Listato 1. Riassumendo i passi salienti dell'algoritmo presentato si ha:

- L'inizializzazione dei vertici ad 1 impiega tempo $O(V)$.
- Il calcolo del GainFactor per ogni vertice impiega tempo $O(V + E)$.
- La creazione di L_G impiega tempo $O(V)$.
- La ricerca del massimo GainFactor all'interno di L_G viene eseguita in tempo $|V|$, mentre la cancellazione di un vertice che ha un GainFactor non positivo avviene in tempo $O(1)$.
- Una volta individuato il prossimo vertice da elaborare, l'aggiornamento dei vicini viene eseguito in tempo $O(V + E)$ mentre la rimozione del vertice stesso da L_G avviene in tempo costante $O(1)$.

Detto questo si può concludere che l'euristica appena esposta presenta una complessità di $O(V) * O(V + E)$, quindi polinomiale alla grandezza del grafo dato in input. Il primo fattore che compone la complessità dedotta rappresenta il numero dei vertici del grafo a cui si assegna un valore romano pari a 2, quindi tale valore è legato al coefficiente di connessione oltre che alla struttura stessa del grafo. Il secondo fattore è rappresentato dall'aggiornamento del GainFactor dei vertici dei nodi. Si ricorda che si preferisce per motivi computazionali di calcolare ad ogni passo il GainFactor dei vertici presenti nella lista L_G che rappresentano i vertici che hanno ancora un GainFactor maggiore di zero e quindi hanno ancora possibilità di vedersi assegnato il valore romano 2. Per grafi densi si ha $E \approx |V|(|V| - 1)/2$ ma per tali grafi l'insieme V_2 indotta dalla funzione di Dominazione Romana risulta essere di cardinalità molto bassa per cui il ciclo principale viene eseguito un numero contenuto di volte. Quindi le tipologie di grafi che rappresentano un punto debole dell'algoritmo sono i grafi sparsi per cui una funzione di Dominazione Romana genera un insieme V_2

numericamente importante e ciò comporta l'aggiornamento ripetuto del GainFactor dei vertici presenti in L_G.

L'euristica ritorna buoni risultati ma ovviamente non è in grado di trovare la soluzione ottima. In Figura 3.3 viene mostrato un grafo su cui è stata calcolata una funzione di Dominazione Romana grazie alla procedura appena esposta mentre in Figura 3.4 viene mostrata la soluzione ottima per lo stesso grafo. Come si può facilmente notare l'euristica sbaglia ad assegnare al vertice centrale il valore romano 2. Questo errore sarebbe sicuramente correggibile tramite una procedura di raffinamento successiva ma in ogni cosa fornisce un contro-esempio sul funzionamento ottimale dell'euristica.

Nella Tabella 3.1 è mostrato un confronto tra i risultati dell'euristica che fa uso soltanto del grado dei vertici (indicata con la sigla *SH*) e quella che utilizza il nuovo parametro dinamico del GainFactor (indicata con la sigla *gFH*). Si nota facilmente che *SH* restituisce valori molto più alti e la differenza aumenta con l'aumentare del numero dei vertici. Questo ci dice che la prima euristica presentata non è all'altezza delle aspettative e soprattuto che l'euristica che fa uso del GainFactor guadagna molto nell'introduzione di questo nuovo parametro. Grazie ad esso infatti, si riesce ad abbassare notevolmente il numero di Dominazione Romana calcolato. Inoltre l'euristica soddisfa pienamente tutto i limiti superiori teorici conosciuti e già esposti in [18], [76] e [58] e il confronto con i risultati ottenuti sarà fatto nella sezione dedicata.

3.4 Possibili varianti

Nell'approccio appena descritto si è scelto di procedere alla copertura dei vertici con GainFactor corrente maggiore. Non si è trattato il problema che si viene a presentare quando due o più vertici hanno lo stesso valore di GainFactor. Quindi l'algoritmo precedentemente illustrato, a parità di GainFactor si limita a selezionare il primo vertice in base all'ordine dato dall'etichetta del vertice stesso nel grafo.

Grafo	$\gamma_{SH}(G)$	$\gamma_{gFH}(G)$
Random, $\|V\| = 150$, cf 0.1	41	29
Random, $\|V\| = 150$, cf 0.2	23	17
Random, $\|V\| = 150$, cf 0.3	18	13
Bipartito, $\|V\| = 200$, cf 0.1	69	52
Bipartito, $\|V\| = 200$, cf 0.2	51	33
Bipartito, $\|V\| = 200$, cf 0.3	34	22
Random, $\|V\| = 2000$, $\Delta(G) = 6$	1109	785
Random, $\|V\| = 2000$, $\Delta(G) = 11$	829	532
Random, $\|V\| = 2000$, $\Delta(G) = 21$	570	335
Random, $\|V\| = 5000$, $\Delta(G) = 6$	2761	1954
Random, $\|V\| = 10000$, $\Delta(G) = 6$	5540	3913
Random, $\|V\| = 10000$, $\Delta(G) = 11$	4156	2622
Random, $\|V\| = 10000$, $\Delta(G) = 21$	2948	1690
Random, $\|V\| = 1000$, $\Delta(G) = 21$	2948	1690
Scale Free Model, $\|V\| = 1000$	603	455
Scale Free Model, $\|V\| = 5000$	3048	2366
Scale Free Model, $\|V\| = 10000$	7365	5898

Tabella 3.1: Confronto tra l'euristica greedy *SH* e euristica con GainFactor *gFH*.

Sono state eseguite altre prove e studiate numerose varianti. Per esempio si può pensare di prendere il vertice più vicino (nel senso di lunghezza di cammino) all'ultimo a cui è stato assegnato un valore di Dominazione Romana pari a 2, oppure il più lontano. Si potrebbe pensare anche di forzare il posizionamento dei valori romani 2 in maniera che non siano adiacenti tra di loro, ma anche questa scelta non ha evidenziato una tendenza positiva, alternando miglioramenti a peggioramenti dei risultati finali. Queste prove sono state eseguite per diversificare la sequenza di copertura dei vertici all'interno del grafo ma questi approcci sebbene portino a risultati del RDN leggermente diversi non è stato possibile definire un andamento generico per cui si è giunti alla conclusione che non è possibile stabilire prima quale di questi approcci porti ad un risultato migliore rispetto ad un altro. Inoltre bisogna considerare anche il fatto che per prendere i vertici ordinati in maniera diversa, bisognerebbe eseguire ulteriori operazioni ad ogni iterazione che porterebbero a un degrado delle prestazioni. Nella prossima sezione viene studiata nel dettaglio una variante che sebbene appesantisca notevolmente il calcolo, è in grado di trovare

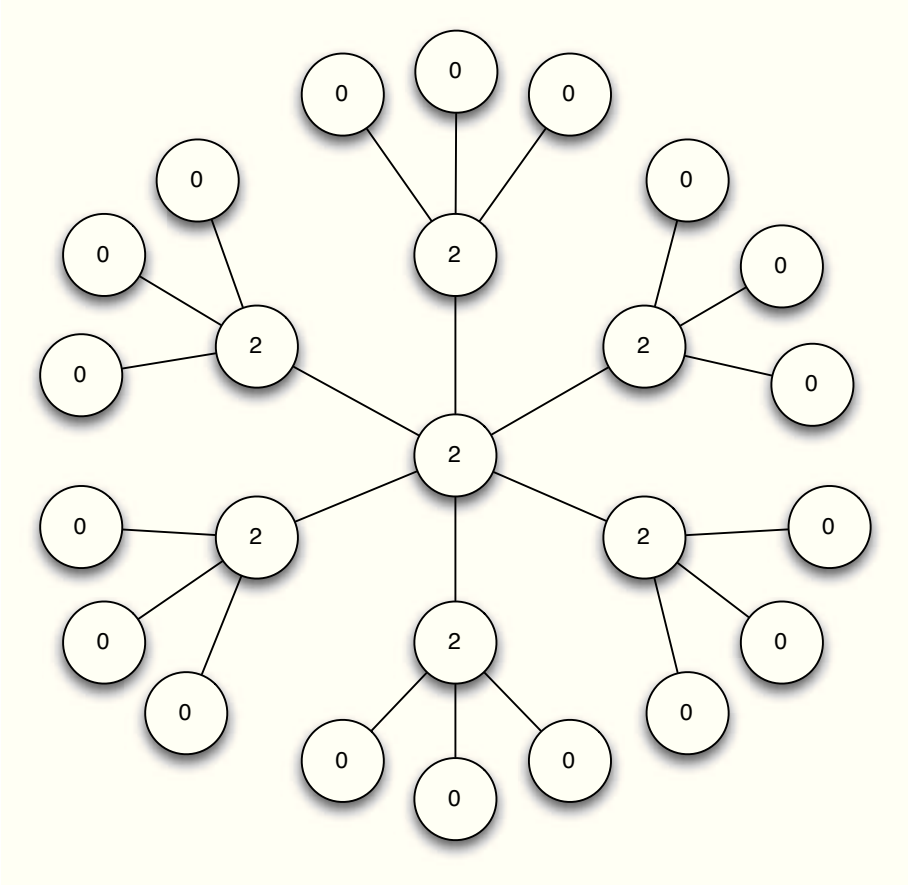


Figura 3.3: Dominazione romana con l'utilizzo del GainFactor.

nella maggior parte dei casi un numero di Dominazione Romana migliore di quello ottenuto nella sezione precedente.

3.4.1 Differenti configurazioni iniziali e percorsi alternativi

Pur ottenendo valori molto buoni, scegliere di iniziare con un vertice con GainFactor alto porta si ad un risultato buono, ma migliorabile in vario modo. Un approccio studiato e implementato è quello di eseguire la stessa euristica sul grafo un numero di volte pari al numero dei vertici del grafo stesso.

Quindi in questo approccio, se n è il numero dei vertici, si procede ad eseguire n volte l'euristica base descritta nella sezione precedente. All'iterazione i, si forza l'assegnazione del vertice i al valore romano 2, in modo da avere in pratica una nuova configurazione iniziare per ogni iterazione. In questo caso il numero di Dominazione

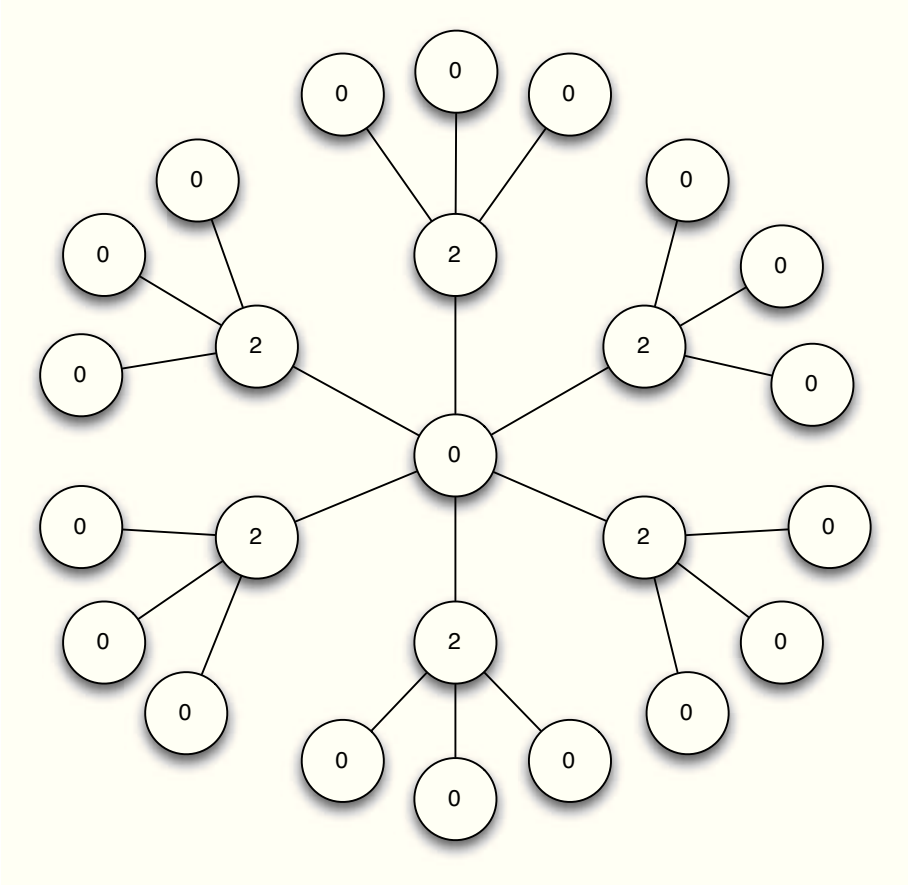


Figura 3.4: Dominazione romana ottima.

Romana sarà uguale al peso della migliore funzione di Dominazione Romana prodotta dalle varie iterazioni.

Quest'approccio soffre sicuramente di alcuni problemi. Per esempio si è notato che eseguendo più iterazioni sullo stesso grafo, pur avendo configurazioni iniziali diverse, molte iterazioni possono eseguire la copertura selezionando lo stesso insieme di vertici con la conseguenza che il numero di Dominazione Romana trovato sarà lo stesso. Per ovviare a questo problema, durante l'esecuzione dell'euristica, ad ogni iterazione che corrisponde ad una configurazione iniziale, viene salvata la sequenza con cui sono stati estratti ed elaborati i vertici dalla lista ausiliaria indicata sempre con L_G. In questo modo, ad ogni iterazione successiva, si verifica se lo specifico percorso è stato già seguito e in caso affermativo, si cerca di sceglierne uno diverso.

Per tenere traccia anche di questa informazione si sono costruite delle ulteriori

liste ausiliarie. Ognuno di queste liste indicate con P_i saranno lunghe $|V|$ e nel momento iniziale ogni posizione sarà uguale a 0. Quando al vertice v corrispondente sarà assegnato il valore romano 2 la posizione corrispondente di v sarà cambiata in 1. Alla fine di ogni iterazione la lista P_i conterrà il valore 1 in corrispondenza dei vertici che sono stati assegnati a V_2 durante la procedura di copertura.

Alla i-esima iterazione, al momento di scegliere il vertice j- esimo da valutare, viene selezionato il vertice v con il GainFactor massimo dalla lista L_G, viene confrontato attraverso un'operazione di XOR alle altre sequenze di copertura, cioè alle liste con indici da 0 a $i-1$ prendendo in esame i primi j valori. Se la sequenza risulta essere differente, allora si assegna il valore romano 2 al vertice v e si procede all'individuazione del prossimo vertice. Altrimenti si sceglie il prossimo vertice dalla lista L_G e si ritorna a verificare se la sequenza così generata è stata già coperta oppure no. Lo pseudo-codice dell'euristica che implementa la variante con i diversi punti iniziali e che cerca di differenziare i percorsi fatti è mostrato nel Listato 2.

Questa euristica è in grado di trovare risultati migliori rispetto all'euristica che esegue una sola procedura di copertura ma ovviamente risulta essere molto più pesante dal punto di vista computazionale. In ogni caso si può anche pensare di eseguire la procedura per un numero di volte k, con $1 < k < n$ e come si è osservato, anche questo approccio porta ad un miglioramento del risultato ottenuto. Oppure, come vedremo nel capitolo successivo, si può provare a utilizzare il calcolo distribuito per computare le varie esecuzioni.

La Tabella 3.2 mostra i risultati ottenuti dalla variante che prevede le configurazioni iniziali multiple e procede selezionando il percorso da seguire durante l'esecuzione. Per comodità viene indicata con la sigla *MIPH* e i risultati vengono confrontati con l'euristica *gFH*. Come si può vedere in tutti i casi mostrati, l'euristica con le diverse configurazioni iniziali riesce a produrre risultati leggermente migliori.

Algoritmo 2 Pseudo-codice dell'euristica con punti iniziali multipli

```
1: MIP-GainFactorHeuristicInit(G)
2: R(V, 1)
3: for all v ∈ V do
4:    UpdateGF(v)
5:    MIP-GainFactorHeuristic(G,v)
6: end for
7: return MAX(RDN)
```

```
1: MIP-GainFactorHeuristic(G,v)
2: while stop do
3:    GF(V)
4:    v ← GetNextNode(G)
5:    if v.gainfactor ≥ 0 then
6:       UpdateGF(v)
7:    else
8:       stop
9:    end if
10: end while
```

```
1: GetNextNode(G)
2: for all v ∈ V do
3:    if v ∉ P_i then
4:       SavePath(P_i, v)
5:       return v
6:    end if
7: end for
```

	$\gamma_{gFH}(G)$	$\gamma_{MIPH}(G)$
Random, $\|V\| = 450$, cf 0.05	70	66
Random, $\|V\| = 350$, cf 0.2	22	20
Bipartito, $\|V\| = 400$, cf 0.05	109	103
Bipartito, $\|V\| = 500$, cf 0.2	37	37
Planare, $\|V\| = 200$	18	17
Planare, $\|V\| = 400$	26	24
Planare, $\|V\| = 500$	31	28

Tabella 3.2: Risultati dell'euristica con GainFactor *gFH* con configurazioni iniziali multiple e percorso selettivo.

3.5 Risultati e Considerazioni

Le euristiche precedenti sono state provate sull'insieme di grafi generato appositamente. Si ricorda che i grafi in considerazioni sono connessi e non orientati, di dimensione variabile tra i 50 e i 10.000 vertici e con coefficiente di connessione variabile (indicato con *cf*) tra 0.01% e 0.9% per quanto riguarda i grafi random generati con il modello di Gilbert [16]. Si ricordi che secondo il modello di Gilbert [16] il fattore di connessione esprime la probabilità che presi due vertici a caso, esiste un arco che li collega.

Dai limiti teorici visti nella fase introduttiva, presentati tra gli altri in [76] e [61] si deduce che dato un grafo G, se $\delta(G) \leq 1$ allora il limite più stretto risulta essere $\gamma_R(G) \leq 4n/5$, se $\delta(G) = 2$ e $|V| \geq 9$ allora il limite più stretto è $\gamma_R \leq 8n/11$ mentre se $\delta(G) \geq 3$ allora si ha che il limite più stretto è dato dall'equazione 3.1. Nella Tabella 3.3 è mostrato come per grafi con $\delta(G) \geq 3$, i risultati dell'algoritmo *gFH* migliorarono tutti ampiamente i limiti teorici noti mostrati nella fase introduttiva mentre nella Tabella 3.4 è mostrato come la stessa tendenza positiva è mantenuta nel caso di grafi con $\delta(G) \leq 1$.

$$\gamma_R(G) \leq n \cdot \frac{2 + \ln((1 + \delta(G))/2)}{1 + \delta(G)}. \tag{3.1}$$

Nella Tabella 3.5 vengono riportati i risultati dell'euristica *gFH* applicata ad un insieme di grafi random con un fattore di connessione fissato pari a 0.1 e una cardinalità variabile di V. Sebbene i grafi presi in esame abbiano un basso coefficiente di connessione, l'insieme V_1 risulta essere molto contenuto al crescere di $|V|$ mentre

Grafo	eq. (3.1)	$\gamma_{gFH}(G)$
Scale Free, $\|V\| = 1000$, $\delta(G) = 4$	583.26	209
Scale Free, $\|V\| = 1000$, $\delta(G) = 5$	516.44	180
Scale Free, $\|V\| = 5000$, $\delta(G) = 5$	2582.18	848
Random, $\|V\| = 100$, $\delta(G) = 4$	58.33	28
Random, $\|V\| = 200$, $\delta(G) = 8$	77.87	33
Random, $\|V\| = 450$, $\delta(G) = 12$	134.02	70
Random, $\|V\| = 1000$, $\delta(G) = 3$	673.29	262
Random, $\|V\| = 5000$, $\delta(G) = 4$	2916.29	838
Random, $\|V\| = 10000$, $\delta(G) = 5$	5164.35	1676

Tabella 3.3: Confronto tra l'algoritmo *gFH* per grafi con $\delta(G) \geq 3$ e i limiti teorici.

Grafo	$4n/5$	$\gamma_{gFH}(G)$
Random, $\|V\| = 100$, $\Delta(G) = 5$	80	61
Random, $\|V\| = 200$, $\Delta(G) = 8$	160	98
Random, $\|V\| = 5000$, $\Delta(G) = 6$	4000	1949
Random, $\|V\| = 10000$, $\Delta(G) = 6$	8000	3913
Bipartito, $\|V\| = 200$, $\Delta(G) = 11$	84	160
Bipartito, $\|V\| = 700$, $\Delta(G) = 10$	560	343
Scale Free, $\|V\| = 50$, $\Delta(G) = 8$	40	32
Scale Free, $\|V\| = 100$, $\Delta(G) = 13$	80	57
Scale Free, $\|V\| = 500$, $\Delta(G) = 78$	400	241
Scale Free, $\|V\| = 1000$, $\Delta(G) = 77$	800	455
Scale Free, $\|V\| = 5000$, $\delta(G) = 5$	2582.18	848

Tabella 3.4: Confronto tra l'algoritmo *gFH* per grafi con $\delta(G) = 1$ e i limiti teorici.

V_2 cresce in maniera costante.

Nella Tabella 3.6 vengono esposti i risultati ottenuti applicando l'euristica *gFH* ad un insieme di grafi random fissando il numero di vertici al valore $|V| = 600$ e variando il fattore di connessione. Si può facilmente osservare come il numero di Dominazione Romana diminuisca velocemente all'aumentare del fattore di connessione tra i vertici. Da sottolineare che la cardinalità di V_2, cioè il numero di vertici a cui è stato assegnato il valore romano 2, è un valore che indica quante volte l'euristica ha eseguito un'estrazione dalla lista L_G e quindi quanti cicli principali sono stati eseguiti.

Nella Figura 3.5 è rappresentato l'andamento della cardinalità degli insiemi V_1 e V_2 che scaturisce dalla funzione di Dominazione romana calcolata con l'euristica *gFH*. Come possiamo osservare sia V_1 che V_2 hanno un comportamento inversamente proporzionale rispetto al fattore di connessione. Gli stessi dati vengono ripresi nella Figura 3.6 riferendosi a grafi con coefficiente di connessione molto bassi.

$\|V\|$	$\|V_2\|$	$\|V_1\|$	$\gamma_{gFH}(G)$
50	8	7	23
100	11	6	27
150	12	6	29
200	15	3	33
250	15	3	33
300	17	3	37
350	17	4	38
400	19	2	40
450	19	2	40
500	19	2	40
550	19	3	41
600	21	1	43
650	21	2	44
700	21	3	45
750	22	0	44
800	22	1	45
850	22	3	47
900	24	1	49
950	23	1	47
1000	23	2	48

Tabella 3.5: Risultati per grafi con fattore di connessione 0.1 usando l'euristica *gFH*.

La Figura 3.7 riporta gli stessi tipi di dati provenienti stavolta da una funzione di Dominazione Romana su grafi a invarianza di scala, generati tramite una leggera variazione del modello di Barabási-Albert [1]. La modifica dal modello originale consiste nell'impostare in fase di creazione del grafo, un valore M che indica il numero di vertici a cui si collega ogni nuovo vertice appena inserito. In questo caso la distribuzione di partizione è meno regolare ma per quanto riguarda l'insieme V_2 mantiene la caratteristica di proporzionalità inversa al crescere della densità del grafo stesso. In questo caso però la partizione V_1 rimane di cardinalità non trascurabile e addirittura per certe densità maggiore di quella di V_2.

Nella Tabella 3.7 vengono riportati i risultati ottenuti applicando l'euristica ad un insieme di grafi con lo stesso fattore di connessione $cf = 0.05$ e lo stesso numero di vertici $|V| = 600$. Come ci si poteva aspettare, il numero di Dominazione Romano rimane pressoché costante.

Fattore di connessione	$\|V_2\|$	$\|V_1\|$	$\gamma_{gFH}(G)$
1	81	59	221
2	59	25	143
3	46	11	103
4	39	7	85
5	33	7	73
6	31	2	64
7	27	4	58
8	24	2	50
9	22	2	46
10	21	1	43
20	12	2	26
30	8	1	17
40	6	1	13
50	5	1	11
60	4	0	8
70	3	2	8
80	3	0	6
90	2	0	4

Tabella 3.6: Risultati per grafi random con 600 vertici tramite algoritmo *gFH*.

Nel grafico di Figura 3.8 è mostrata la distribuzione dei vertici indotta dalla partizione dei vertici a seguito del calcolo della funzione di Dominazione Romana. A differenza dei grafi random generati con il modello di Edgar Gilbert [16] in cui all'aumentare del fattore di connessione l'insieme V_1 individuato dalla funzione romana tende a diminuire rapidamente, nel caso del modello di Barabási-Albert [1] continua a crescere insieme a V_2.

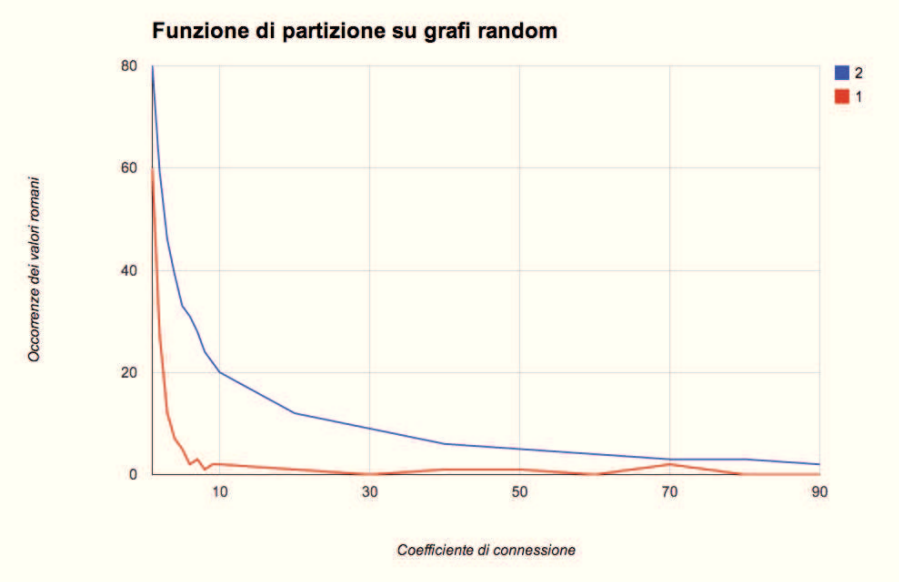


Figura 3.5: Distribuzione della partizione indotta dalla funzione romana su grafi random.

$\lvert V_2\rvert$	$\lvert V_1\rvert$	$\gamma_{gFH}(G)$
33	4	70
34	3	71
34	4	72
35	5	75
35	2	72
35	3	73
33	5	71
33	5	71
34	4	72
33	6	72

Tabella 3.7: Risultati per grafi con $cf = 0.05$ e $|V| = 600$ usando l'algoritmo *gFH*.

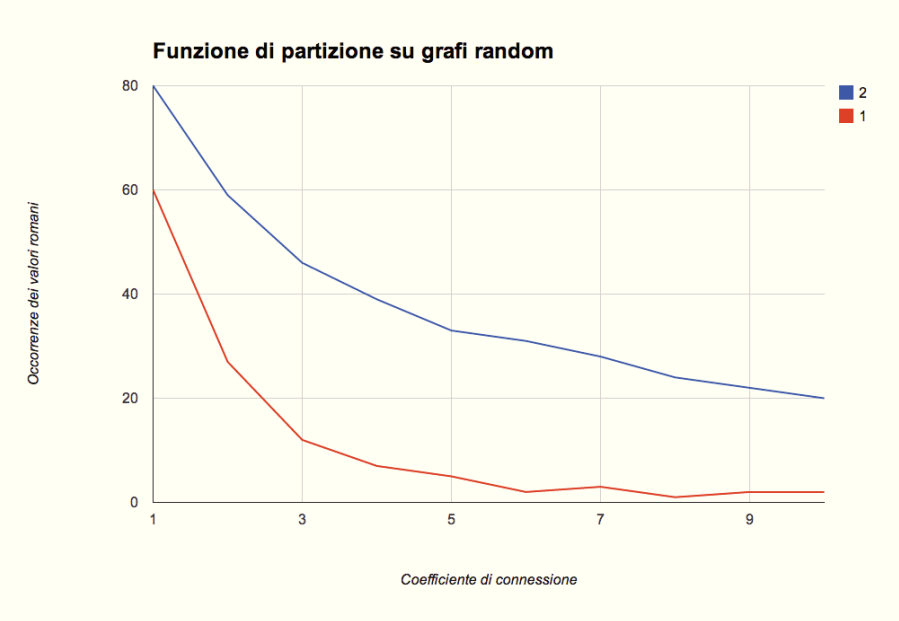


Figura 3.6: Distribuzione della partizione indotta dalla funzione romana su grafi sparsi.

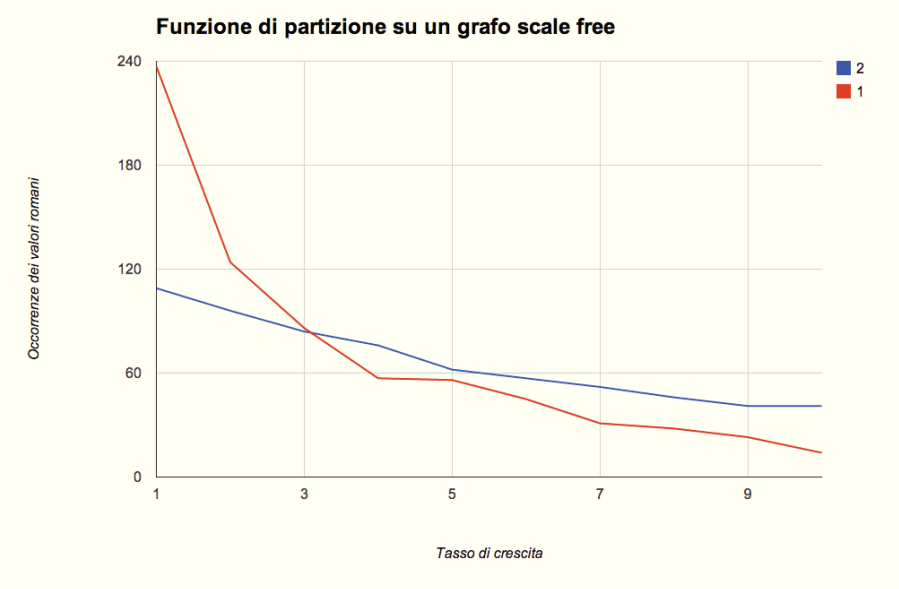


Figura 3.7: Distribuzione della partizione indotta dalla funzione romana su grafi a invarianza di scala.

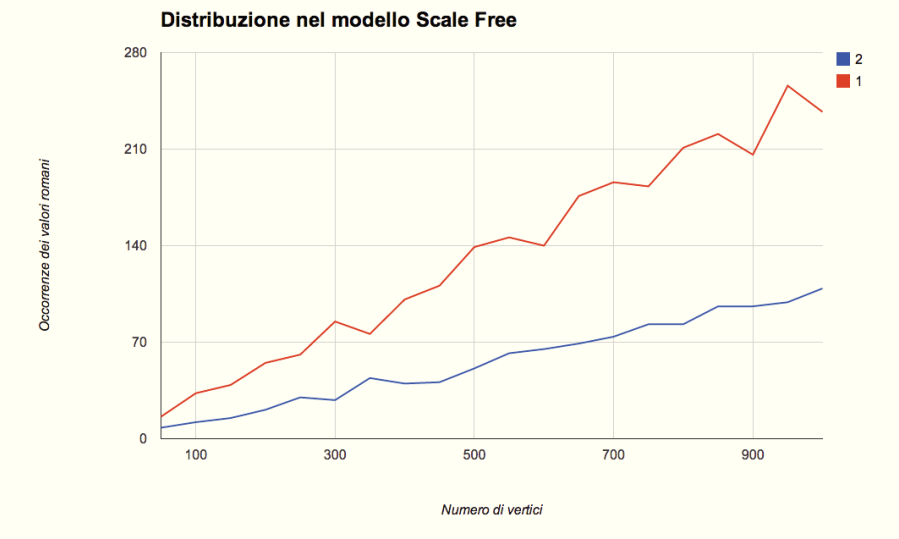


Figura 3.8: Distribuzione della partizione indotta dalla funzione romana su grafi scale free con numero di vertici variabile.

Capitolo 4

Calcolo distribuito della Dominazione Romana

CUDA (acronimo di *Compute Unified Device Architecture*) è un'architettura hardware per l'elaborazione parallela creata da NVIDIA. Tramite l'ambiente di sviluppo per CUDA è possibile scrivere applicazioni capaci di eseguire calcolo parallelo sulle GPU delle schede video NVIDIA. In questo capitolo sarà implementato tramite questa piattaforma l'algoritmo euristico che prevede punti iniziali multipli di copertura. Saranno studiate e implementate varie ottimizzazioni per le diverse caratteristiche di grafi in esame e per adattare lo stile di programmazione al paradigma tipico della piattaforma CUDA.

4.1 Ambiente di sviluppo CUDA

La piattaforma di elaborazione in parallelo CUDA consente di esprimere il parallelismo dei task in linguaggi ad alto livello come C, C++, Fortran o in standard aperti come le direttive OpenACC. La piattaforma di elaborazione in parallelo è ampiamente usata in migliaia di applicazioni accelerate dalle GPU e oggetto di numerose dissertazioni e pubblicazioni scientifiche. Degli esempi sono i lavori [42], [47], [25], [8] e [41] in cui vengono ottimizzati proprio problemi sulla teoria dei grafi, sebbene i grafi non si

prestano perfettamente alle esecuzione in parallelo come si vedrà più avanti in questo capitolo. Nella maggior parte dei casi viene utilizzata la potenza di calcolo della piattaforma per calcolare ottimizzazioni combinatorie di problemi su grafi cercando di adattare le rappresentazioni dei dati in memoria all'infrastruttura ospitante.

Usando CUDA, le ultime GPU Nvidia diventano quindi architetture aperte come le CPU. Diversamente dalle CPU, le GPU hanno un'architettura parallela con diversi core, ognuno capace di eseguire centinaia di processi simultaneamente, e supponendo che l'applicazione in esame sia adatta o adattata per questo tipo di architettura, la GPU può offrire grandi performance e benefici. Questo approccio di risoluzione dei problemi è noto come GPGPU.

Solitamente un'algoritmo necessita di diversi sotto-processi per portare a termine l'obbiettivo finale. Con il termine *thread* si indica un singolo sotto-processo, che può essere eseguito in maniera concorrenziale rispetto agli altri thread in esecuzione. Il funzionamento concorrenziale dei thread può essere implementato sia su sistemi di elaborazione mono-processore (e in questo caso si parla di sistemi multi-threading) oppure su sistemi multi-processore come il caso di CUDA. Le porzioni dell'intero processo vengono eseguite sulla piattaforma sono forma di *kernel*. Può essere eseguito un kernel alla volta e più thread possono eseguire contemporaneamente lo stesso kernel.

Il modello di memoria a disposizione dell'architettura è suddiviso in porzioni in base alla posizione della memoria stessa e a chi può accedere ad essa. Si distinguono i seguenti tipi di memoria che vengono tutti utilizzati in questo lavoro:

- **Registri**: associati ad ogni singolo thread, accessibili solo da esso e con lo stesso ciclo di vita.
- **Memoria Locale**: associata ad ogni singolo thread, posizionata fisicamente sulla DRAM, anche in questo caso ha lo stesso ciclo di vita del thread di riferimento.
- **Memoria Condivisa**: associata ad un blocco di thread ne condivide il ciclo di vita. Essa si trova fisicamente sul chip della GPU.

- **Memoria Globale**: Accessibile da tutti i thread come dalla CPU. Il ciclo di vita è gestito da un sistema di allocazioni e rilasci di porzioni di memoria.
- **Memoria Host**: Rappresenta la memoria della CPU ospitante e non è accessibile dai thread.

CUDA ha parecchi vantaggi rispetto alle tradizionali tecniche di computazione sulle GPU che usano le API grafiche. Le più importanti sono riportate qui di seguito:

- Letture temporali: il codice viene letto attraverso indirizzi arbitrari di memoria.
- Memoria condivisa: veloce condivisione di memoria (una regione di 16 kb di grandezza) che può essere condivisa fra i thread. Questa può essere usata come una cache gestita dall'utente.
- Riletture veloci verso e dalla GPU.
- Supporto completo per divisioni intere e operazioni bit-a-bit.

Come già accennato, l'esecuzione su piattaforma CUDA necessità di un adattamento non solo dei codice ma anche dell'intero paradigma utilizzato per il calcolo. In fase di progettazione si devono tenere in conto anche le varie limitazioni presenti:

- CUDA è composto da un sottoinsieme del linguaggio C privo di ricorsione e puntatori a funzione, con l'aggiunta di alcune semplici estensioni. Comunque, un singolo processo deve essere eseguito attraverso multiple disgiunzioni di spazi di memoria, diversamente da altri ambienti di runtime C.
- Sono presenti sono diverse variazioni dallo standard IEEE 754 per quanto riguarda la doppia precisione (supportata nelle nuove GPU come la GTX 260).
- La larghezza di banda e la latenza tra CPU e GPU può rappresentare un collo di bottiglia.
- I thread devono essere eseguiti in multipli di 32 per ottenere prestazioni migliori, con un numero totale di thread nell'ordine delle migliaia. Le ramificazioni del

codice non influiscono nelle prestazioni, a condizione che ciascuno dei 32 thread prenda lo stesso cammino di esecuzione.

Ci sono quindi degli aspetti che occorre tenere in considerazione durante la progettazione e lo sviluppo di algoritmi che mirano a sfruttare l'architettura CUDA. L'architettura obbliga lo sviluppatore ad utilizzare determinate strategie per poter ottenere il massimo rendimento dalla parallelizzazione. Tuttavia riuscire a raggiungere questo intento non è affatto semplice e in alcuni casi non è neanche possibile e in ogni caso la struttura del codice che rispetta queste regole può risultare molto complessa.

L'architettura CUDA è di tipo single SIMD ovvero *Single Instruction, Multiple Data*, ciò significa che il parallelismo si ottiene quando una singola istruzione è eseguita su un insieme di dati abbastanza grande. Oltre a questo CUDA impone che i dati siano allineati secondo regole precise e che i thread accedano ad aree di memoria contigue e che gli accessi avvengano linearmente. Ad esempio dato un vettore di n elementi e lanciati n thread di un kernel che manipolino questo vettore, si deve fare in modo che il thread i-esimo acceda all'elemento i-esimo dell'array.

Altri fattori di cui bisogna tener conto sono gli accessi in memoria *unchached* (cioè accessi a porzioni di memoria non presenti nella memoria condivisa) che provocano latenze variabili dai 400 ai 600 cicli di clock. Inoltre lanciare un numero basso di thread o utilizzare configurazioni di lancio particolari non permette di sfruttare in pieno le potenzialità dell'hardware con la conseguenza che il software eseguito su CPU potrebbe risultare più veloce di quello eseguito tramite GPGPU in quanto la frequenza di clock sulle CPU è molto più alta.

Tenendo in considerazione l'ambiente in cui si sta lavorando, occorre far in modo di utilizzare il meno possibile ogni istruzione di controllo del flusso di esecuzione come i costrutti *if* e allineare le letture per renderle coalescenti (fare in modo che possano esser soddisfatte più letture contemporanee tramite la lettura di un unica area contigua di memoria). Se non si tiene conto di questi accorgimenti può presentarsi il caso in cui l'esecuzione del codice non avviene in parallelo ma ad esempio ogni ramo

di esecuzione ed ogni lettura non coalescente vengono eseguite in seriale (aumentando il tempo necessario per eseguire tutto il processo).

Ovviamente esistono classi di problemi che si adattano meglio a tali architetture (come alcune elaborazioni sulle immagini di cui [69] rappresenta uno dei tanti esempi) ed altre che invece presentano non pochi problemi come appunto la gestione dei grafi.

Infatti la natura dei grafi comporta l'accesso sparso alle loro strutture, questo è stato il grosso ostacolo da aggirare, in parte utilizzando le tecniche messe a disposizione dalla memoria condivisa. Questo approccio permette di condividere dati di input tra i vari thread. Questo però pone dei limiti sulla cardinalità massima del grafo con cui si vuole operare, limitazioni legate alle risorse disponibili.

Si è scelto di implementare tramite CUDA l'euristica con le diverse configurazioni iniziali. Non è stato possibile utilizzare la tecnica dei percorsi alternativi per le varie esecuzioni in quanto l'approccio risulta non compatibile con l'architettura CUDA e quindi porterebbe a un grande dispendio di memoria e a lunghi tempi di esecuzione. Due risultano essere le possibili strade da poter percorrere per parallelizzare l'euristica in questione:

- Parallelizzare le esecuzioni facendo in modo che ogni singolo thread si occupi di un'unica simulazione (quindi fare in modo di eseguire simultaneamente più simulazioni).

- Effettuare una parallelizzazione sulla singola esecuzione, ovvero ogni thread concorre al completamento di una singola esecuzione (le varie simulazioni vengono serializzate).

Tenendo in considerazione il grande numero di strutture dati da mantenere in memoria prevista dal primo approccio, si è scelto di percorrere la seconda strada e tentare in alcuni casi una soluzione ibrida.

4.2 Implementazione dell'euristica in CUDA

L'obbiettivo iniziale è stato quello di produrre una versione funzionante dell'euristica implementata nel capitolo precedente in modo tale da avere una base su cui poter costruire le versioni successive ed un metro per paragonare i risultati ottenuti man mano. In questa versione si è mantenuta la rappresentazione di un grafo tramite lista di adiacenza, il tutto implementato tramite due vettori *zero-indexed*:

- **Start** è composto da n elementi. Al vertice i-esimo corrisponde quindi l'i-esimo elemento del vettore.
- **EdgesHeap** è un vettore che contiene informazioni sui vicini che ogni vertice del grafo possiede.

Con questa impostazione, tutti i vertici sono indicizzati a partire da 0, inoltre per ogni vertice i-esimo si può facilmente risalire ai suoi adiacenti e il primo vicino del vertice di indice i è il vertice con indice *EdgesHeap[Start[i]]*.
Ad ogni vertice quindi corrisponde una porzione del vettore *EdgesHeap*. Per chiarire meglio la rappresentazione dei dati si prenda in esame il grafo mostrato in Figura 4.1.

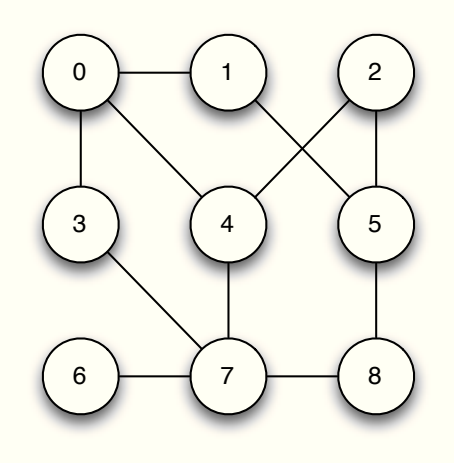


Figura 4.1: Grafo d'esempio per la rappresentazione in memoria su CUDA.

Il grafo possiede 9 vertici indicizzati da 0 ad 8. Di seguito è mostrato il vettore *Start* che contiene la rappresentazione dell'elenco dei vertici:

[0, 3, 5, 7, 9, 12, 15, 16, 20]

Il vettore *EdgesHeap* contiene invece l'elenco dei vicini di ogni vertice:

[1, 3, 4, 0, 5, 4, 5, 0, 7, 0, 2, 7, 1, 2, 8, 7, 3, 4, 6, 8, 5, 7]

I diversi colori son stati usati per sottolineare l'alternanza dei vicini dei diversi vertici. Tramite l'indice recuperato dal vettore *Start* è possibile posizionarsi sui vicini che si trovano in *EdgesHeap*. Dato quindi il vertice di indice j è sempre possibile sapere dove inizia e dove finisce la porzione del vettore *EdgesHeap* che contiene i suoi vertici adiacenti. L'indice d'inizio è recuperabile da *Start[j]*, mentre l'indice del suo ultimo elemento è reperibile dall'indice *Start[j+1]-1* tranne che per il vertice di indice $n-1$ la cui porzione di *EdgesHeap* termina con la fine del vettore stesso. Anche per questo motivo sono passati come parametri di input oltre che le strutture appena esposte anche il valore di $|V|$ e $|E|$, cioè la cardinalità degli insieme di vertici e archi presenti in G.

Oltre alla struttura del grafo occorrono anche dei vettori ausiliari, necessari per memorizzare le informazioni appartenenti ai vertici come il GainFactor e il valore romano corrente. Tali strutture sono rappresentate rispettivamente da due vettori di interi:

- **NodeValue** è un vettore di dimensione n contenente il valore romano corrente dei relativi vertici.
- **GainFactors** è un vettore anch'esso di dimensione n in cui di volta in volta vanno memorizzati i valori di GainFactor dei vari vertici.

Considerando che si vuole implementare l'algoritmo con diversi punti iniziali di copertura, ad ogni singola esecuzione viene passato l'indice del vertice da cui far partire la simulazione e le varie strutture dati ausiliarie necessarie. Il primo passo di una singola esecuzione è sempre quello di inizializzare il grafo assegnando il valore romano 1 a tutti i vertici.

A questo punto viene fatto un primo aggiornamento del valore romano dei vertici e quindi alla simulazione i-esima viene richiamato la funziona di aggiornamento passando tra i vari parametri il valore i che rappresenta il vertice di partenza da cui far iniziare la procedura di copertura. In questa fase vengono lanciati almeno $nAdj$ thread, dove $nAdj$ equivale al numero di vicini posseduti dal vertice i. Ad ogni thread viene assegnato quindi uno dei vicini di i, ed ognuno di essi deve leggere il proprio valore dal vettore *NodeValue*. In caso questo equivalga al valore 1, deve aggiornarlo ponendolo a 0. Il thread di indice 0 inoltre ha l'onere di porre il valore romano del vertice su cui è richiamato a 2.

Questo approccio evita di lanciare un solo thread che si occupi di impostare il vertice di riferimento e tutti i suoi vicini in modo sequenziale.

L'ultimo passo di inizializzazione consiste nel calcolare il GainFactor per tutti i vertici e ciò viene fatto lanciando n thread divisi in più blocchi. Tutti i thread di un blocco provvedono a caricare tutto il vettore *NodeValue* sulla memoria condivisa in modo tale che le letture siano coalescenti. L'architettura CUDA permette di leggere intere porzioni di memoria presenti nella memoria globale quindi se tutte le letture avvengono all'interno di un blocco, possono essere soddisfatte contemporaneamente a patto che le letture avvengano allineate.

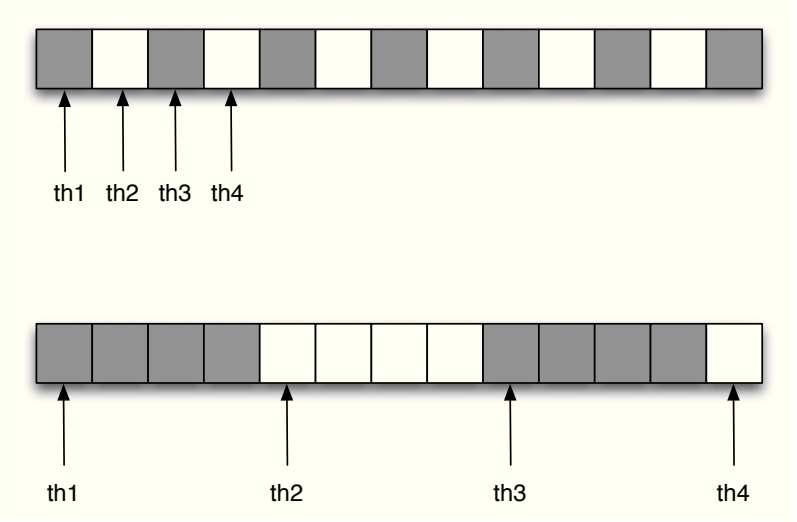


Figura 4.2: Esempio di accessi coalescenti alla memoria.

La Figura 4.2 mostra un esempio semplificato sul come avvengono le letture del vettore *NodeValue*. Negli esempi è mostrato parte di un vettore *NodeValue* di 12 elementi. Supponendo di avere 4 thread e che le letture avvengano a blocchi di 4, leggendo con il primo schema, tutte le letture possono essere soddisfatte contemporaneamente, invece utilizzando il secondo schema, già per una sola lettura occorre caricare 4 blocchi di memoria.

Nell'implementazione si è fatto in modo che le letture avvengano come mostrato nel primo esempio, quindi inizialmente tutti i thread si occupano di leggere una prima parte contigua del vettore, poi si accede alla seconda parte contigua successiva e così via, fin quando tutto il vettore è stato letto.

Il vettore *NodeValue* è un vettore di interi (e in effetti è più efficiente leggere interi dalla memoria globale), ma i valori contenuti possono essere rappresentati anche con un *char* e considerando che la dimensione della memoria condivisa di un blocco è limitata, tali valori vengono memorizzati proprio come *char* sulla memoria condivisa. La fase successiva prevede che ogni thread si occupi di aggiornare il valore del GainFactor del proprio vertice e per far ciò legge i valori dalla memoria condivisa (implementare questi accessi sulla memoria globale risulterebbe molto inefficiente).

Con questa operazione termina la fase di inizializzazione del grafo ed iniziano le varie iterazioni dell'algoritmo. Per effettuare la prima iterazione tuttavia è necessario identificare il vertice con GainFactor massimo e poiché non basta solamente sapere qual'è il valore massimo ma anche la sua posizione non è possibile ricorrere ad una delle primitive fornite dall'ambiente CUDA per la ricerca del massimo, quindi la ricerca viene fatta richiamando una funzione scritta appositamente..

A questo punto finché il valore ritornato dalla funzione è positivo si richiamano le varie iterazioni e alla fine si calcola il numero di Dominazione Romana della corrente simulazione, sommando tutti i valori romani assegnati.

4.3 Ottimizzazione degli accessi in memoria

Le prestazioni ottenute con la prima versione sviluppata sono influenzate negativamente dagli accessi sparsi in memoria. Dalle osservazioni effettuate emerge che la maggior parte del tempo di calcolo (oltre il 90%) viene occupato, come ci si poteva aspettare, dal calcolo del GainFactor per ogni vertice.

Già l'accesso al vettore *EdgesHeap* risulta essere sparso, perché ogni thread si occuperà di un vertice diverso, dunque esaminerà una porzione differente del vettore. Inoltre, dopo aver recuperato l'identificativo del vertice adiacente i-esimo è necessario accedere al vettore *NodeValue* per recuperarne il valore romano corrente, rendendo ancora più sparso l'accesso in memoria globale. Oltre a questo, la lunghezza del ciclo *for* dipende dal grado del vertice processato; ciò comporta un aumento della frammentazione dell'esecuzione e l'impossibilità da parte del compilatore di effettuare *loop unroll*, cioè trascrivere ed eseguire il codice necessario all'interno di un ciclo, come una sequenza di istruzioni.

Una seconda versione sviluppata tenta di rendere più regolari gli accessi in memoria modificando la struttura dati che rappresenta il grafo. Per sfruttare al meglio la coalescenza, infatti, lo schema degli accessi è stato cambiato e corrisponde a quello in Figura 4.3.

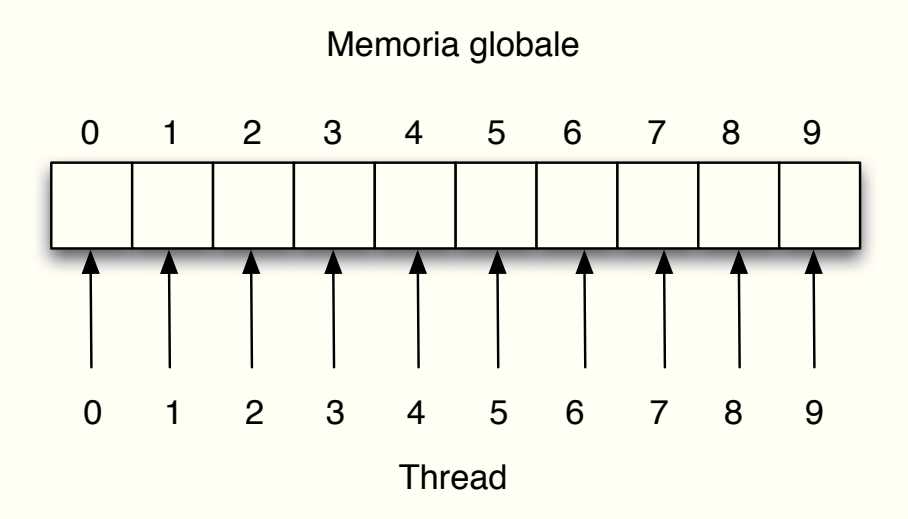


Figura 4.3: Schema degli accessi ottimale per la coalescenza.

Classe di adiacenza	Grado	Vertici
0	1	6
1	2	1, 2, 3, 8
2	3	0, 4, 5
3	4	7

Tabella 4.1: Classi di adiacenza dei vertici.

Ogni thread si occupa di un vertice, e legge un valore adiacente alla volta. Per aderire allo schema di accessi ottimale, la prima cella di memoria dovrebbe contenere il primo vertice adiacente al vertice processato dal thread 0. La seconda cella di memoria dovrebbe contenere il primo vertice adiacente al vertice processato dal thread 1 e così via. In questo modo la totalità degli accessi risulta equivalente allo schema ottimale.

Sorge a questo punto una problematica da risolvere: il GainFactor viene aggiornato per ogni vertice, quindi com'è possibile organizzare la struttura dati in maniera tale da avere sempre in sequenza i vertici adiacenti necessari? La soluzione consiste nel suddividere i vertici in base al loro grado. Considerando nuovamente il grafo di Figura 4.1 e suddividendo i vertici in base al grado, otteniamo le classi di adiacenza mostrate nella Tabella 4.1.

Invece di avere un unico vettore per gli adiacenti di tutti i vertici (*EdgesHeap*) si ha un numero di vettori uguale al numero di classi di adiacenza. In questo esempio vi è una corrispondenza $1-1$ tra le classi di adiacenza e i gradi, ma in generale potrebbe non essere così; infatti nel grafo potrebbero esserci vertici di grado 1, 3, 7, ma nessuno di grado 2, 4 e 5. In tal caso si avranno solo 3 classi di adiacenza, i cui indici non corrispondono ai gradi dei vertici che appartengono alla classe.

A questo punto, per ogni classe di adiacenza, bisogna aggregare in un singolo vettore tutti i vertici adiacenti a quelli appartenenti alla classe. Considerando ad esempio la classe di adiacenza 2, i vertici appartenenti a tale classe sono 0, 4 e 5. I vertici adiacenti risultano essere quelli riportati nella Tabella 4.2.

Vertici	Adiacenti
0	1, 3, 4
4	0, 2, 7
5	1, 2, 8

Tabella 4.2: Vertici e Vicinato proprio.

Aggregando ed intrecciando tutti i vertici adiacenti in un unico vettore si ottiene:

$$[\ 1,\ \ 0,\ \ 1,\ \ 3,\ \ 2,\ \ 2,\ \ 4,\ \ 7,\ \ 8\ ]$$

Questo elenco può essere visto come l'unione di 3 sottogruppi (ogni vertice ha infatti grado 3). Infatti $1, 0, 1$ è il sottogruppo dei primi vertici adiacenti, $3, 2, 2$ è il sottogruppo dei secondi vertici adiacenti e così via. Applicando lo stesso criterio per tutte le classi di adiacenza, si ottiene la seguente matrice frastagliata:

$$\begin{bmatrix} 7. & & & & & & & & \\ 0, & 4, & 0, & 5, & 5, & 5, & 7, & 7. & \\ 1, & 0, & 1, & 3, & 2, & 2, & 4, & 7, & 8. \\ 3, & 4, & 6, & 8. & & & & & \end{bmatrix}$$

Questa matrice viene calcolata prima di effettuare la simulazione insieme ai seguenti vettori ausiliari:

- **nAdjs**: Vettore indicizzato per classe di adiacenza che contiene l'informazione di quanti vertici appartengono a tale classe. Per il caso in esame si ha:

$$nAdjs = 1, 4, 3, 1.$$

- **nodeStride**: Vettore indicizzato per classe di adiacenza che contiene il grado dei vertici appartenenti a tale classe. Per il caso in esame si ha:

$$nodeStride = 1, 2, 3, 4.$$

ID Vertice	Classe di adiacenza	Indice sottogruppo
0	2	0
1	1	0
2	1	1
3	1	2
4	2	1
5	2	2
6	0	0
7	3	0
8	1	3

Tabella 4.3: Classi di adiacenza e relativi indici.

- **nodeDegreeClassAndIndex**: Vettore che associa ad ogni vertice del grafo la classe di adiacenza e l'indice dei vertici adiacenti relativo a ciascun sottogruppo. Si ha quindi una coppia di interi per vertice (e quindi l'indicizzazione è del tipo 2 * id).

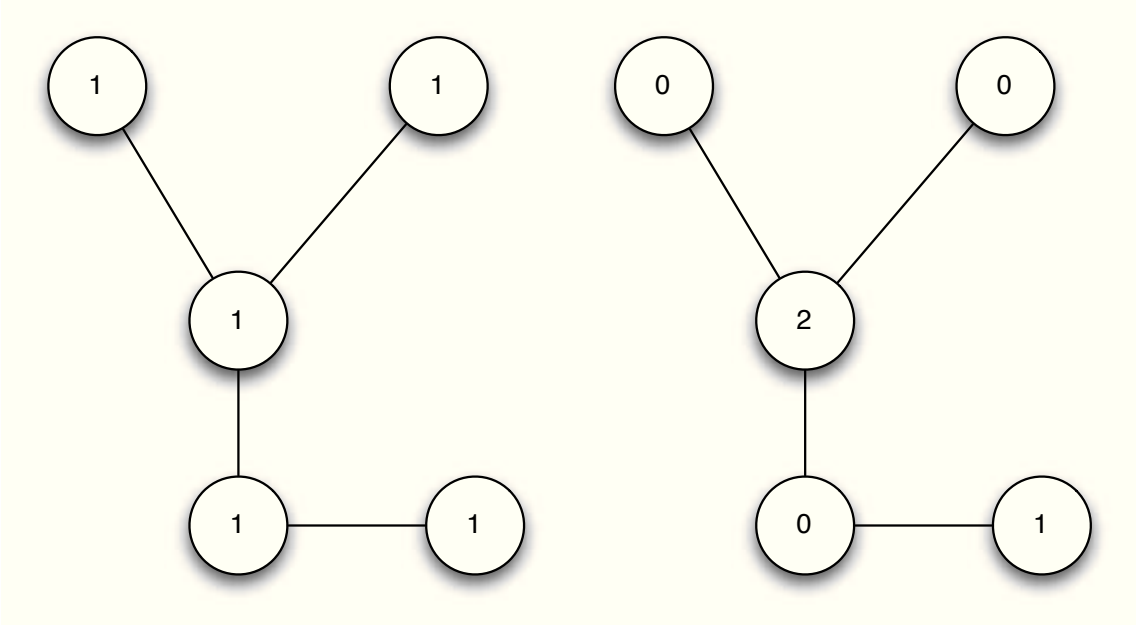
$$\{\{2,0\},\{1,0\},\{1,1\},\{1,2\},\{2,1\},\{2,2\},\{0,0\},\{3,0\},\{1,3\}\}.$$

- **nVertex**: numero di vertici del grafo (in questo caso 9).
- **count**: numero di classi di adiacenza (in questo caso 4).

La configurazione appena descritta è riassunta nella Tabella 4.3. A questo punto si ponga l'attenzione su come avviene la fase di etichettatura e aggiornamento di un singolo vertice, che dato un vertice centrale v pone il suo valore romano a 2 mentre pone a 0 il valore romano di tutti i vertici adiacenti che hanno in quel momento un valore romano pari a 1.

Innanzitutto, dato l'indice del vertice centrale bisogna determinare a quale classe di adiacenza appartiene. Ciò si ottiene facilmente tramite il vettore *nodeDegreeClassAndIndex*, in particolare la posizione risulta essere uguale a:

*nodeDegreeClassAndIndex[2 * nodeID].*



(a) Prima (b) Dopo

Figura 4.4: Variazione del GainFactor.

Oltre a questo serve sapere in quale posizione si trovano i vertici adiacenti all'interno di ciascun sottogruppo. Anche per questo scopo si utilizza ancora una volta *nodeDegreeClassAndIndex* e in particolare si fa riferimento all'indice:

*nodeDegreeClassAndIndex[2 * nodeID + 1].*

A questo punto vengono lanciati tanti thread quanti sono i vertici adiacenti di *nodeID*, ripartendoli eventualmente in più blocchi. Ogni thread dunque esaminerà un solo vertice adiacente di *nodeID*, ed in particolare il primo thread si occuperà anche di impostare a 2 il valore romano del vertice centrale. Supponendo di impostare il valore romano 2 sul vertice 4, si ha la configurazione mostrata in Figura 4.4.

Il vertice 4 ha grado 3 ed appartiene alla classe di adiacenza 2. Lo schema seguente chiarisce la struttura della lista.

$$\underbrace{1, 0, 1}_{\textit{Primi adiacenti}} \quad , \quad \underbrace{3, 2, 2}_{\textit{Secondi adiacenti}} \quad , \quad \underbrace{4, 7, 8}_{\textit{Terzi adiacenti}} \quad .$$

Poiché *nodeDegreeClassAndIndex[2 * 4 + 1] = 1*, l'interno di ciascun sottogruppo di vertici adiacenti occupano la posizione di indice 1. Nella Figura 4.5 questi indici sono evidenziati in rosso.

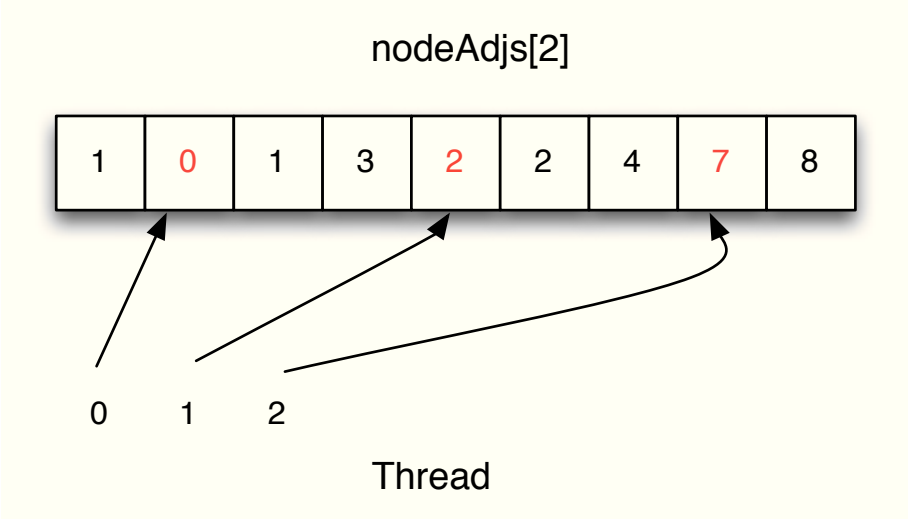


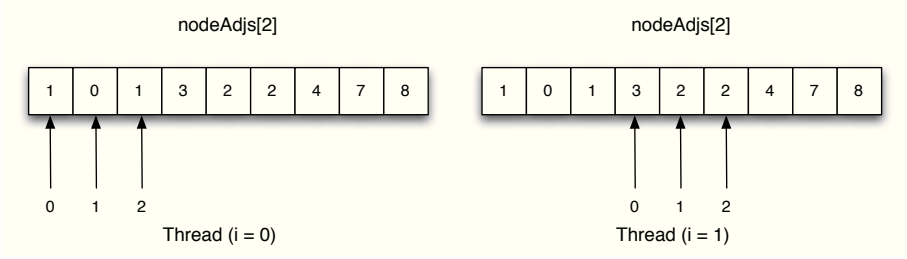
Figura 4.5: Accessi sparsi per l'aggiornamento del GainFactor.

Come si evince dalla Figura 4.5, l'aver adottato questa particolare struttura dati penalizza la coalescenza per l'operazione di assegnazione di un valore romano pari a 2 e conseguente aggiornamento del vicinato di un vertice, in quanto gli accessi sono inevitabilmente sparsi.

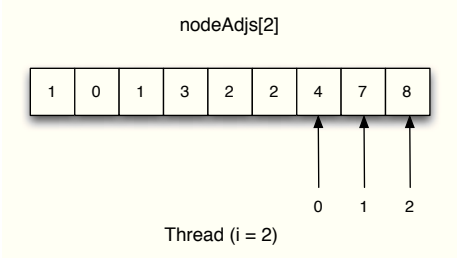
L'altra operazione fondamentale è il calcolo del GainFactor. In questa versione, si hanno due funzioni:

- **calcGF()**: Kernel CUDA che si occupa di aggiornare il GainFactor di tutti i vertici aventi lo stesso grado.
- **updateGainFactors()**: Routine host che aggiorna il GainFactor per tutto il grafo richiamando più volte il kernel *calcGF()*.

In particolare il kernel *calcGF()* verrà richiamato tante volte quante sono le classi di adiacenza. Vengono usati tanti thread quanti sono i vertici che appartengono a quella classe, ed ognuno di essi si occuperà di scandire tutti gli adiacenti del vertice associato. Aggiornando il GainFactor per tutti i vertici di grado 3 (classe di adiacenza



(a) I iterazione (b) II iterazione



(c) III iterazione

Figura 4.6: Calcolo del GainFactor per una classe di adiacenza.

2), ogni thread mantiene un accumulatore per il GainFactor. Gli accessi effettuati sono mostrati nella Figura 4.6. L'intero processo va ripetuto per ogni classe di adiacenza.

La particolare struttura dati permette di ridurre (anche se non del tutto) gli accessi non coalescenti alla memoria. Nel complesso però, tale soluzione è più lenta rispetto alla prima versione; ciò è dovuto principalmente a due motivi:

- Ogni classe di adiacenza può contenere molti vertici, ognuno con molti vertici adiacenti. Lanciare tanti thread quanti sono i vertici spesso risulta inadeguato; d'altra parte far calcolare lo stesso GainFactor a più thread risulterebbe problematico perché bisognerebbe trovare un modo per fondere i risultati successivamente.

- Ci possono essere molte classi di adiacenza, e ricordando che è necessario lanciare tante volte il kernel *calcGF()* quante sono le classi di adiacenza, l'operazione risulta essere computazionalmente dispendiosa..

4.3.1 Ottimizzazione delle prestazioni

L'implementazione appena presentata, pur riorganizzando la struttura dati, non riesce a sfruttarla bene per aumentare la velocità di esecuzione dell'intero processo. La versione esposta in questa sezione mantiene la stessa struttura dati, ma tenta di calcolare il GainFactor in maniera differente per ottimizzare le prestazioni. Vengono introdotti i due nuovi kernel elencati di seguito:

- **findIndexedSum()** che si occuperà di volta in volta di recuperare i vertici coinvolti nel calcolo del GainFactor.
- **sumValuesToGainFactor()** che si occuperò di sommare i valori che contribuiscono alla somma del GainFactor stesso.

L'idea scaturisce come risposta alle principali problematiche della versione precedente:

- **Problema 1**: Un thread deve occuparsi sequenzialmente del calcolo di un singolo GainFactor. **Soluzione**: Calcolare la somma mediante riduzione parallela.
- **Problema 2**: Il kernel *calcGF()* viene lanciato troppe volte. **Soluzione**: Effettuare tutte le somme necessarie in parallelo quando possibile. Non è possibile fare una sola riduzione parallela per tutte le classi di adiacenza in quanto sia il numero di elementi sia l'indice associato all'output varia in modo non uniforme.

Per ovviare a questi problemi, innanzitutto bisogna stabilire un parametro *parallelSums* che indica quante riduzioni parallele effettuare contemporaneamente. Per grafi non troppo grandi si possono fare tutti i calcoli necessari in una sola volta, altrimenti li si può spezzare in più parti (che in genere sono di numero inferiore al numero delle classi di adiacenza). La riduzione parallela ha un'altra particolarità: non vengono sommati direttamente i valori dei vari vettori, bensì vengono trattati come indici per il vettore dei valori romani; vengono inoltre sommati solamente i valori romani 1 (gli unici di interesse per il calcolo del GainFactor).
In dettaglio, i parametri necessari di *findIndexedSum()* sono:

- **dValueArray**: Vettore che contiene i valori da sommare. Corrisponde al vettore che contiene i valori romani *NodeValue*.
- **dIndexMatrix**: Matrice frastagliata contenente gli indici relativi a *dValueArray*. Corrisponde alla matrice *dNodeAdjs*.
- **nIndexes**: Numero di vettori contenuti in *dIndexMatrix*. Corrisponde al numero dei vertici *nVertex*.
- **hIndexesCount**, **dIndexesCount**: Vettore (host e device) indicante il numero di elementi di ogni vettore della matrice frastagliata. Corrisponde a *nAdjs* e *dNAdjs*.
- **indexesPerThread**: Numero di indici processati da ogni thread nella prima fase della riduzione parallela.
- **base**: Indica da quale vettore della matrice frastagliata bisogna iniziare le riduzioni parallele. Le riduzioni effettuate riguarderanno i vettori di indice *[base...base + parallelSums]*. Se il numero di somme parallele è uguale al numero di vertici, allora *base* sarà uguale a 0.
- **parallelSums**: Numero di somme parallele da effettuare.
- **dResults**: Vettore contenente i risultati finali delle riduzioni parallele effettuate. Corrisponde al vettore dei valori di GainFactor.

Il lancio del kernel principale *GPUIndexedParallelSum()* viene configurato nel seguente modo. La griglia è bidimensionale e quindi al variare dell'indice y varia il vettore considerato dalla somma parallela (quindi $gridDim.y = parallelSums$). Al variare dell'indice x varia invece il thread all'interno della stessa somma parallela. Il primo passo da effettuare è determinare il numero di thread da lanciare. Poiché il numero di thread è lo stesso per tutti i blocchi, è necessario configurare il lancio nel caso peggiore, ossia per il vettore più grande; l'indice di tale vettore *maxIndexes* viene calcolato una sola volta prima che venga lanciata la vera e proprio esecuzione. Il numero di thread è dato dalla formula:

$$\texttt{NTHREADS} = \left\lceil \frac{\texttt{hIndexesCount}[\texttt{maxIndexes}]}{\texttt{indexesPerThread}} \right\rceil .$$

Il numero di thread per blocco (*THREADSPERBLOCK*) è fissato a 128. Il numero di blocchi viene calcolato di conseguenza:

$$\texttt{NBLOCKS} = \left\lceil \frac{\texttt{NTHREADS}}{\texttt{THREADSPERBLOCK}} \right\rceil .$$

La memoria condivisa conterrà tanti elementi quanti sono i thread, dunque:

$$\texttt{SHMEMSIZE} = \texttt{THREADSPERBLOCK} * \texttt{sizeof}(\texttt{int}).$$

La routine *findIndexedSum()* richiama inizialmente il kernel *GPUIndexedParallelSum()* configurato secondo i calcoli descritti. I parametri sono:

- *dValueArray*.
- *dIndexMatrix*.
- *dIndexesCount*.
- *dIndexMatrix*.
- *indexesPerThread*.
- *g_odata*: Matrice frastagliata di output dove verranno memorizzati i risultati parziali delle varie riduzioni parallele. Tale matrice è necessaria per non alterare i dati di input. Viene creata una sola volta e la sua dimensione è pari a *parallelSums* righe, dove ogni riga contiene tanti elementi quanti sono quelli del vettore più grande (tale numero è dato da *hIndexesCount[maxIndexes]*).
- *g_odata_count*: Vettore (con *parallelSums* elementi) che indica il numero di elementi contenuti in ogni vettore corrispondente di *g_odata*.

Le operazioni svolte dal kernel sono molto simili a quelle di una riduzione parallela classica (passaggio dalla memoria globale a quella condivisa, riduzione nella memoria

condivisa e così via). Durante la somma vengono presi in considerazione solo i valori uguali ad 1. Dopo la riduzione in memoria condivisa, i risultati vengono scritti per essere salvati nei parametri *g_ odata* e *g_ odata_ count*.

Una volta terminata l'esecuzione del primo kernel si ha in output una matrice frastagliata e le lunghezze dei vettori in esso contenuti. A questo punto però i valori contenuti nella matrice non sono più indici, bensì dei semplici valori da sommare direttamente. Per questo motivo è stato introdotto un nuovo kernel *GPUParallelSum()* utilizzato nelle fasi successive delle riduzioni parallele. La configurazione del lancio per questo nuovo kernel è analoga a quella descritta in precedenza. Ci sono però alcune differenze rispetto al primo kernel:

- Bisogna sommare i valori direttamente.
- E' possibile sovrascrivere la matrice passata in input.
- Se una riduzione parallela termina (ossia rimane un solo elemento) allora il risultato deve essere scritto in *dResults*.
- Nella prima fase della riduzione ogni thread si occupa sempre di due elementi, indipendentemente da *indexesPerThread*.

Viene quindi innanzitutto calcolato il numero di iterazioni (ossia il numero di chiamate a *GPUParallelSum()*) necessarie per ridurre tutti i vettori. La formula utilizzata è la seguente:

$$\texttt{nIterations} = \lceil \log_2(\texttt{maxDim}) \rceil$$

dove *maxDim* è la dimensione massima tra tutti i vettori ottenuti dalla prima fase di riduzione parallela. Nel caso particolare in cui i vettori siano già stati tutti ridotti semplicemente con l'ausilio del primo kernel, *GPUParallelSum* viene richiamato comunque e il suo unico compito sarà quello di trasferire i risultati ottenuti dalla matrice frastagliata al vettore *dResults*.

Questo approccio migliora sensibilmente le prestazioni rispetto ai precedenti, grazie alla nuova routine per il calcolo del GainFactor.

4.4 Ottimizzazione per grafi sparsi

La versione esposta di seguito non ha un importanza significativa per i grafi con connettività media superiore allo 0.1 in quanto il guadagno in prestazioni non è considerevole per tali grafi sopratutto se confrontato con il consumo di risorse (come la memoria) per ottenerlo mentre potrebbe rappresentare un notevole guadagno per grafi sparsi. L'obbiettivo preposto è quello di ridurre il numero di GainFactor da dover calcolare in quanto il cambiamento del valore romano dei vertici successivamente ad un operazione di aggiornamento di una posizione riguarda solo gli insiemi del vicinato di primo e di secondo livello. Ridurre il numero di GainFactor da aggiornare non solo riduce il numero di thread da lanciare ma riduce inoltre il numero di accessi in memoria (che risulterebbero non coalescenti).

Per raggiungere tale obbiettivo è necessario possedere informazioni su quali sono i vertici facenti parte dei due particolari insiemi, i vertici del vicinato di primo livello corrispondono alle varie porzioni di *EdgesHeap*. I vertici del vicinato di secondo livello devono essere di volta in volta calcolati partendo dalle informazioni contenute in *EdgesHeap*. Eseguire questi calcoli di volta in volta risulta essere molto costoso, sopratutto se l'operazione viene eseguita sulla GPGPU. La soluzione più comoda è stata quindi quella di far preventivamente calcolare questi insiemi alla CPU assieme ai vettori *Start* ed *EdgesHeap*.

Similmente a quanto fatto in precedenza per rappresentare il grafo si generano due vettori:

- **StartNeigh**: analogamente a *Start*, tale vettore di dimensione n indica a partire da quale indice di *Neighbors* vengono elencati i vertici facente parte del vicinato di primo livello del vertice i-esimo.
- **Neighbors**: analogamente a *EdgesHeap* contiene una serie di vertici che divisi in partizioni rappresentano il vicinato di secondo livello di tutti i vertici del grafo.

Riprendendo il grafo di Figura 4.1, le strutture dati aggiuntive rispetto alla prima versione sono mostrate di seguito.
Il vettore *StartNeigh* sarà della forma:

$$[\ 0,\ \ 3,\ \ 7,\ \ 11,\ \ 16,\ \ 21,\ \ 24,\ \ 27,\ \ 30\ ]$$

mentre il Vettore *Neighbors* è formato come segue:

$$[\ 2,\ \ 5,\ \ 7,\ \ 2,\ \ 3,\ \ 4,\ \ 8,\ \ 0,\ \ 1,\ \ 7,\ \ 8,\ \ 1,\ \ 4,\ \ 6,\ \ 8,$$

$$1,\ \ 3,\ \ 5,\ \ 6,\ \ 8,\ \ 0,\ \ 4,\ \ 7,\ \ 3,\ \ 4,\ \ 8,\ \ 0,\ \ 2,\ \ 5,\ \ 1,\ \ 2,\ \ 3,\ \ 4,\ \ 6\ ]$$

Per facilitare la gestione di tutte queste strutture dati che tra l'altro necessitano di un'ulteriore variabile intera *nNeighbors* che indica la dimensione del vettore *Neighbors*, è stata realizzata una struttura che prende il nome di *SHData* e che contiene tutti i riferimenti necessari alle varie strutture dati, in questo modo lo scambio di questi valori fra i vari metodi viene notevolmente semplificato.

La dimensione della struttura dati cresce più o meno significativamente in relazione alla percentuale di connettività e le strutture coinvolte sono:

- *Start*
- *EdgesHeap*
- *StartNeigh*
- *Neighbors*
- *NodeValue*
- *GainFactors*

Start, *StartNeigh*, *NodeValue* e *GainFactors* sono di dimensione fissa n. Invece la somma degli elementi di *EdgesHeap* e *Neighbors* può raggiungere il valore n^2, in quanto la somma degli elementi delle porzioni relative al vertice i-esimo di *EdgesHeap*

e di *Neighbors* può valere al massimo n. Poiché i vertici sono n, il consumo di memoria risulta essere quadratico sul numero dei vertici del grafo.

Le differenze significative rispetto alla prima versione stanno, oltre che nella creazione delle strutture necessarie, anche nel funzionamento del kernel che si occupa dell'aggiornamento del valore romano di un determinato vertice del grafo.

Vengono quindi aggiunti due parametri di input: *nAdj* e *nAdj2* che indicano rispettivamente la cardinalità del vicinato di primo livello e la cardinalità del vicinato di secondo livello corrispondente al vertice relativo all'iterazione corrente. I thread di indice inferiore a *Adj+nAdj2* ricavano il vertice di cui devono occuparsi leggendo tale valore in *EdgesHeap* ed in *Neighbors* dopo di che procedono con il normale calcolo del GainFactor.

Questa versione è stata pensata per la computazione del numero di Dominazione Romana su grafi con connettività molto bassa (questi grafi sono quelli più interessanti in quanto sono quelli più difficili da gestire su CPU) e per questo motivo questa versione non riesce a fornire grandi vantaggi su grafi con connettività alta in quanto già con connettività superiori al 0.1 la somma di *nAdj + nAdj2* è prossima a n. Da considerare anche la quantità di memoria che queste strutture dati occupano se rapportate a grafi molto grandi come possono esistere in natura. In ogni caso per grafi di natura diversa come possono essere quelli planari, in cui la connettività tra vertici è molto bassa, i tempi di esecuzione migliorano significativamente.

4.5 Ottimizzazione del calcolo del GainFactor

Con la versione esposta nella sezione precedente ci si rende conto ancora di più come il peso maggiore dell'esecuzione è sull'aggiornamento del GainFactor per i vari vertici. Questa operazione causa una grande ridondanza negli accessi al vettore *NodeValue*. Ad esempio il vertice l ed il vertice m se hanno come vicino il vertice n entrambi leggeranno il valore *NodeValue[n]*.

L'idea è quella di assegnare ad ogni vertice un ulteriore valore *Ones* oltre al GainFactor ed al *NodeValue* in cui memorizzare il numero dei suoi vicini che hanno

valore romano 1.

In questo modo il calcolo del GainFactor per il vertice i diventa banale; infatti basta leggere il valore di *Ones[i]*, sommarlo al *NodeValue[i]* e sottrarre due, senza dover accedere ai dati degli altri vertici.

La complessità viene quindi spostata sulla parte che si occuperà di decrementare i valori *Ones* dei vicini dei vertici il cui valore passa da 1 a 0 oppure a 2.

Oltre alle strutture dati presenti nella versione precedente è stata aggiunto solo il vettore *Ones* di dimensione n che funziona allo stesso modo di *GainFactor* e di *NodeValue*, ovvero al vertice i-esimo corrisponde l'i-esimo valore di *Ones*. Per ogni simulazione ogni elemento di tale vettore deve esser inizializzato ed il valore corrisponde per ogni vertice al numero dei suoi vicini. Per questo motivo è stato allocato una copia di questo vettore in modo tale da poter effettuare una copia veloce e poter far ripartire velocemente l'algoritmo alle iterazioni successive.
Modifiche minori riguardano la creazione e il caricamento delle nuove strutture. I kernel modificati sono:

- **updatePositionKernel**: a tale kernel si passa, oltre alle strutture dati necessarie delle precedente versioni, anche il vettore appena introdotto *Ones*. Sia *centerIndex* l'indice del vertice da cui fare partire l'aggiornamento del valore romano, come prima cosa ad ogni thread viene assegnato un vertice *curID*, e dopo si controlla il valore di *NodeValue[centerIndex]*. Se questo valore vale 1, ogni thread decrementa il valore *Ones* associato al vertice. A questo punto tutti i thread aggiornano il *NodeValue[centerIndex]* ponendolo a 2 .

  Dopo questa fase ogni thread controlla se *NodeValue[curID]* ha valore 1, in questo caso provvede ad aggiornare il proprio *NodeValue* al valore 0, ciò comporta la responsabilità per il thread che si occupa del vertice, di decrementare i valori corrispondenti di *Ones* dei vertici adiacenti ad esso. Queste operazioni sono molto delicate in quanto possono verificarsi facilmente corse critiche che porterebbero alla concorrenza di più decrementi su un valore di *Ones* di un vertice, per cui non si potrebbe garantire che tutti i decrementi siano state

concluse correttamente. Per questo motivo occorre utilizzare operazioni di decremento atomiche.

- **updateGainFactor**: una volta aggiornati tutti i valori di *Ones*, il calcolo del GainFactor diventa piuttosto veloce in quanto basta che ogni thread a cui viene assegnato un vertice vada a leggere il corrispondente valore *NodeValue*, lo sommi a quello di *Ones* e sottragga 2.

Durante l'aggiornamento del GainFactor come visto fino ad adesso, per ogni vertice si deve andar a leggere il valore romano di tutti i propri vicini ed i vertici di cui occorre calcolare il GainFactor sono tutti i vertici del vicinato di primo e secondo livello oltre che il vertice stesso e ciò implica il fatto che ogni vertice venga letto più volte e si arriva a leggere anche i vertici a distanza tre dal vertice centrale (per calcolare il GainFactor di un vertice che fa parte del vicinato di secondo livello). Il nuovo approccio ha il grande vantaggio rispetto ai precedenti di non richiedere la lettura di tutti questi vertici ma solo quelli appartenenti al vicinato di primo e di secondo livello. Il problema principale sta nel fatto che occorre effettuare oltre che una lettura anche una scrittura e inoltre queste operazioni devono essere atomiche per i motivi già esposti.

Tutto ciò fa si che questa versione si comporti in maniera inusuale rispetto alle altre versioni per le quali in genere l'aumentare della connettività velocizza l'esecuzione, mentre questa versione si comporta parecchio bene con i grafi meno densi ed invece le sue prestazioni degradano notevolmente con i grafi con connettività alta. Ciò è spiegabile con il fatto che per i grafi più densi si verificano più corse critiche e le scritture quindi vengono eseguite in serie rendendo l'algoritmo molto inefficiente, con i grafi sparsi invece l'algoritmo riesce a dare il meglio di se in quanto le probabilità che si verifichino corse critiche diminuisce drasticamente.

4.6 Ottimizzazione con matrice di adiacenza

La particolarità di questa versione è l'utilizzo di una matrice di adiacenza per rappresentare il grafo. Dalle versioni precedenti, emerge che il calcolo del GainFactor porta ad accessi sparsi e ridondanti. Esplorare i vertici adiacenti e poi nuovamente i loro adiacenti porta a considerare gli stessi vertici più volte, soprattutto per i grafi più densi.

La matrice di adiacenza adatta viene costruita impiegando un bit per vertice, anziché un byte o più. Ciò permette un risparmio in termini di memoria (che comunque rimane quadratico nel numero dei vertici), nonché la manipolazione dei vertici a gruppi, mediante operazioni *and-bitwise*.

Considerando ancora una volta il grafo d'esempio della Figura 4.1, la matrice di adiacenza risulta essere la seguente:

$$\begin{bmatrix} 0101 & 1000 & 0000 \\ 1000 & 0100 & 0000 \\ 0000 & 1100 & 0000 \\ 1000 & 0001 & 0000 \\ 1010 & 0001 & 0000 \\ 0110 & 0000 & 1000 \\ 0000 & 0001 & 0000 \\ 0001 & 1010 & 1000 \\ 0000 & 0101 & 0000 \end{bmatrix}$$

Ogni riga corrisponde ad un vertice. Per questo esempio si assume di poter manipolare 4 bit alla volta (nell'implementazione reale si usa un *int* a 32 bit). I bit evidenziati in rosso sono in eccesso e la loro presenza è dovuta al fatto che il numero di vertici del grafo non è un multiplo di 4.

Viene adottato un formato simile anche per memorizzare i valori romani. Per tale scopo si usano due vettori:

- **nodeValueMaskOne**: Vettore di bit che contiene tanti elementi quanti sono

i vertici. Dato l'indice di un vertice, il valore corrispondente nella maschera *nodeValueMaskOne* è acceso se il vertice ha valore romano 1.

- **nodeValueMaskTwo**: Vettore di bit che contiene tanti elementi quanti sono i vertici. Dato l'indice di un vertice, il valore corrispondente nella maschera *nodeValueMaskTwo* è acceso se il vertice ha valore romano 2.

Per il grafo di Figura 4.1, dopo la prima fase iniziale tali vettori valgono:

$$\begin{bmatrix} \texttt{nodeValueMaskOne} & 1111 & 1111 & 1000 \\ \texttt{nodeValueMaskTwo} & 0000 & 0000 & 0000 \end{bmatrix}$$

La fase di inizializzazione consiste semplicemente nell'impostare ad 1 tutti i bit di *nodeValueMaskOne*, mentre vengono messi a 0 tutti quelli di *nodeValueMaskTwo*. Il primo vettore richiede un po' di attenzione, in quanto bisogna impostare ad 1 solamente i bit relativi al grafo, mentre a 0 quelli in eccesso, i quali sono sempre rappresentati in rosso.

Durante la fase di assegnazione di un valore romano 2, dato un vertice centrale di indice *startNodeID*, i passi da fare sono:

- Spegnere il bit relativo a *startNodeID* in *nodeValueMaskOne*. Nell'esempio, il vettore diventa:

$$\begin{bmatrix} \texttt{nodeValueMaskOne} & 0111 & 1111 & 1000 \end{bmatrix}$$

- Accendere il bit relativo a *startNodeID* in *nodeValueMaskTwo*. Nell'esempio, il vettore diventa:

$$\begin{bmatrix} \texttt{nodeValueMaskTwo} & 1000 & 0000 & 0000 \end{bmatrix}$$

- Spegnere i bit degli adiacenti di *startNodeID* che hanno valore romano uguale a 1. E' necessario usare sia la matrice di adiacenza che *nodeValueMaskOne*. Si affianca la riga di indice 0 della matrice di adiacenza ed il vettore *nodeValue-*

MaskOne:

$$\begin{bmatrix} \texttt{adjMatrix[0]} & 0101 & 1000 & 0000 \\ \texttt{nodeValueMaskOne} & 0111 & 1111 & 1000 \\ \texttt{adjMatrix[0]} \ \& \ \texttt{nodeValueMaskOne} & 0101 & 1000 & 0000 \end{bmatrix}$$

Effettuando l'*and-bitwise* tra le due righe, si ottiene una nuova maschera avente il bit 1 per ogni vertice che:

- E' adiacente al vertice di indice 0.
- Ha 1 come valore romano.

Questa maschera indica esattamente quali sono i bit da azzerare in *nodeValueMaskOne*. Per farlo, bisogna invertire i bit di *mask* (la maschera ottenuta dall'operazione precedente) ed effettuare un *and-bitwise* con *nodeValueMaskOne*. Si ottiene così:

$$\begin{bmatrix} \texttt{nodeValueMaskOne} & 0111 & 1111 & 1000 \\ \sim \texttt{mask} & 1010 & 0111 & 1111 \\ \texttt{nodeValueMaskOne} \ \& \ \sim \texttt{mask} & 0010 & 0111 & 1000 \end{bmatrix}$$

dove l'ultima riga rappresenta il valore aggiornato del vettore `nodeValueMaskOne`.

In definitiva, dopo questi passaggi si ottiene:

$$\begin{bmatrix} \texttt{nodeValueMaskOne} & 0010 & 0111 & 1000 \\ \texttt{nodeValueMaskTwo} & 1000 & 0000 & 0000 \end{bmatrix}$$

I GainFactor vengono ricalcolati sempre per tutti i vertici del grafo. L'operazione più costosa nel calcolo è il conteggio dei vertici adiacenti aventi 1 come valore romano. Tale operazione può essere svolta più rapidamente grazie alle operazioni di *and-bitwise*. Intuitivamente, bisogna sovrapporre (mediante l'*and-bitwise*) il vettore `nodeValueMaskOne` a tutte le righe della matrice di adiacenza, contando quanti bit accesi rimangono (che corrispondono al valore della sommatoria).

$$\begin{bmatrix} \texttt{nodeValueMaskOne} & 0010 & 0111 & 1000 \end{bmatrix}$$

$$\begin{bmatrix} 0101 & 1000 & 0000 \\ 1000 & 0100 & 0000 \\ 0000 & 1100 & 0000 \\ 1000 & 0001 & 0000 \\ 1010 & 0001 & 0000 \\ 0110 & 0000 & 1000 \\ 0000 & 0001 & 0000 \\ 0001 & 1010 & 1000 \\ 0000 & 0101 & 0000 \end{bmatrix}$$

In particolare vengono lanciati tanti thread quanti sono i vertici; ognuno di essi si occupa di calcolare il GainFactor di un singolo vertice.

Dapprima viene letto il primo elemento (4 bit) della riga della matrice di adiacenza. Poi viene fatto l'*and-bitwise* con il primo elemento di *nodeValueMaskOne*. Tale porzione di maschera ha tanti bit accesi quanti sono i vertici adiacenti (relativi alla porzione) del vertice considerato (*index*) che hanno 1 come valore romano.
I valori vengono accumulati nel valore di GainFactor di riferimento (sommando i vari contributi dei diversi elementi) che alla fine viene scritto sulla memoria globale. Il vettore che memorizza i vari GainFactor non è compresso, cioè utilizza un *int* per ogni vertice.

Lo schema proposto presenta uno svantaggio: la matrice di adiacenza viene letta per colonne, mentre per sfruttare la coalescenza è necessario leggerla per righe. La situazione attuale è mostrata in Figura 4.7.

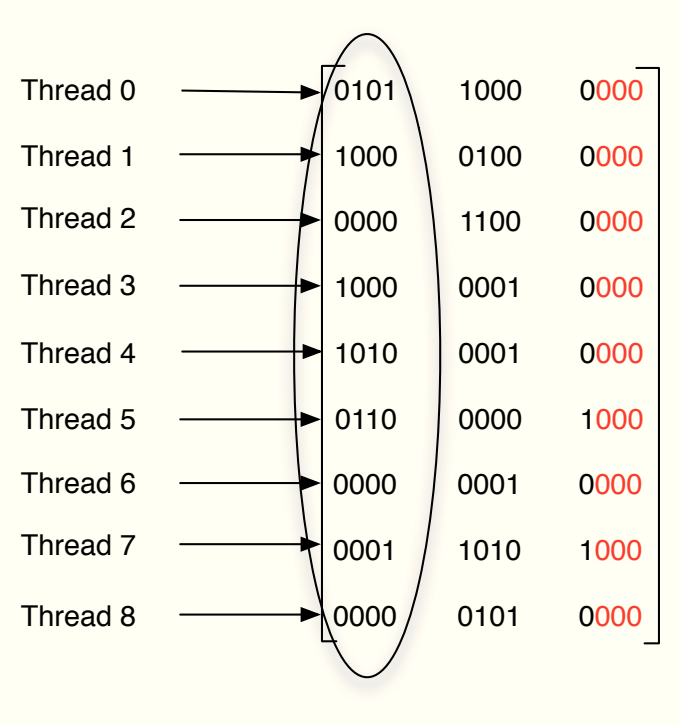


Figura 4.7: Accesso non ottimale alla memoria globale.

Questo problema può essere risolto memorizzando la matrice di adiacenza in ordine *column-major*. In questo modo i thread potranno leggere la matrice per righe.

$$\begin{bmatrix} 0101 & 1000 & 0000 & 1000 & 1010 & 0110 & 0000 & 0001 & 0000 \\ 1000 & 0100 & 1100 & 0001 & 0001 & 0000 & 0001 & 1010 & 0101 \\ 0000 & 0000 & 0000 & 0000 & 0000 & 1000 & 0000 & 1000 & 0000 \end{bmatrix}$$

4.6.1 Considerazioni finali

Poiché viene usata una matrice di adiacenza, la velocità di esecuzione del kernel *updateGainFactor()* è indipendente dalla densità del grafo. Nonostante questo, la densità del grafo influisce comunque sulla durata complessiva dell'esecuzione dell'algoritmo, infatti in un grafo denso bastano poche chiamate che si occupano di aggiornare una determinata posizione con il valore romano 2, ovviamente modificando di conseguenza tutto il vicinato coinvolto, per far scendere il GainFactor di tutti i vertici a valori uguali o minori di zero, mentre per i grafi sparsi si avranno più iterazioni

per raggiungere lo stesso obbiettivo. Si potrebbe pensare di aggiornare solamente il GainFactor relativo ai vertici coinvolti dall'aggiornamento della determina posizione, diminuendo gli accessi in memoria e velocizzando il processo. Purtroppo ciò non è utile né per i grafi densi (poiché un vertici ha quasi tutti gli altri come adiacenti), né per i grafi sparsi.

In un grafo sparso, l'aggiornamento di una posizione coinvolge meno vertici perché ognuno di essi ha un grado limitato. Ma i vertici per cui è necessario calcolare nuovamente il GainFactor non sono solo gli adiacenti del vertice posto a 2, ma anche i vertici a distanza 2 da quello in esame.

Effettuando dei test con un grafo da 1000 vertici e 0.1 di connettività, si è visto che ogni vertice ha in media 80 vertici adiacenti, ma che l'insieme dei vertici a distanza 1 e a distanza 2 da quello centrale copre la maggior parte del grafo, obbligando quindi a ricalcolare il GainFactor per (quasi) tutti i vertici. I tempi di esecuzione vengono mostrati nelle Figure 4.8 e 4.9.

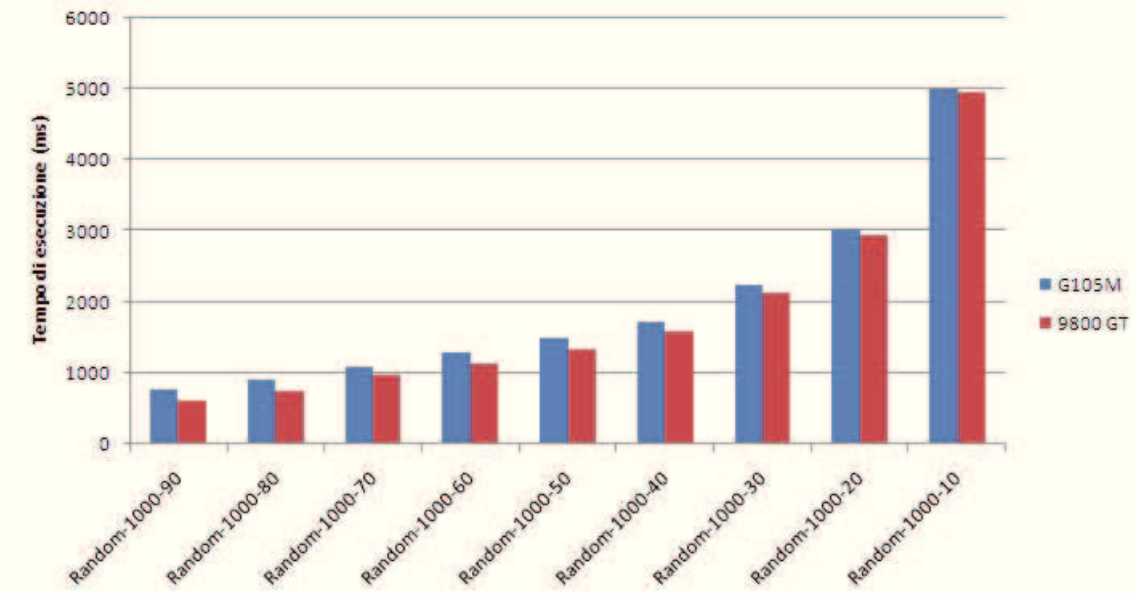


Figura 4.8: Tempi di esecuzione al variare del coefficiente di connessione.

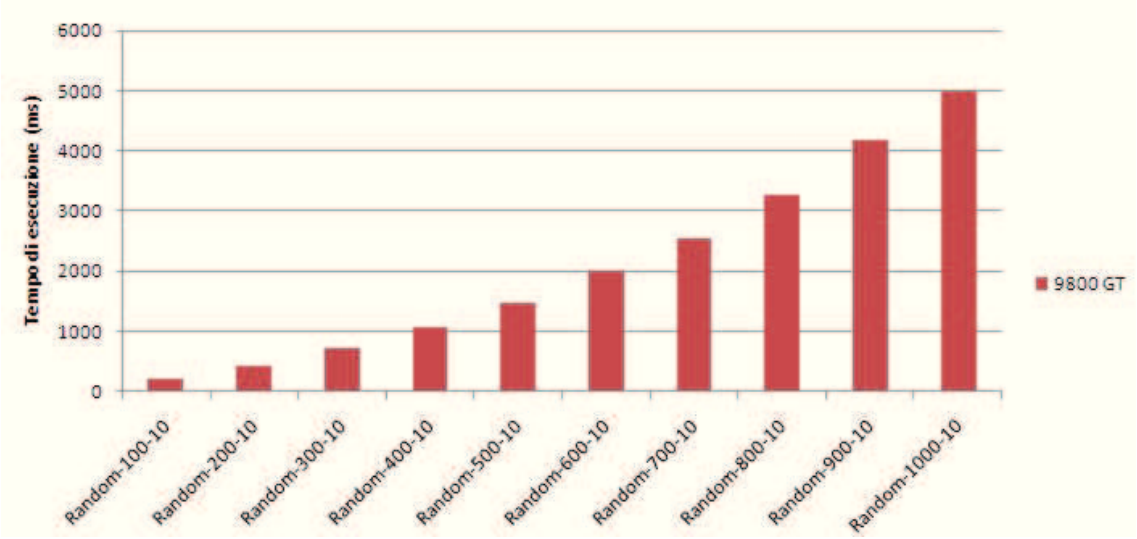


Figura 4.9: Tempi di esecuzione al variare del numero di vertici.

Una possibile strategia per aumentare ulteriormente la velocità di esecuzione potrebbe essere quella di gestire più di una simulazione contemporaneamente. Normalmente ogni simulazione inizia chiamando il kernel *updateGainFactor()* sul vertice iniziale di indice *startNodeID* e poi esegue i vari calcoli, sovrapponendo il vettore *dNodeValueMaskOne* a tutte le righe della matrice di adiacenza. Tra una simulazione e l'altra, però la matrice di adiacenza non cambia. Gestendo più simulazioni alla volta si potrebbe leggere una sola volta la matrice di adiacenza riusando gli stessi valori più volte, risparmiando quindi molte letture in memoria.
Questo approccio però complica notevolmente l'implementazione. In particolare l'uso di memoria, già alto a causa della matrice di adiacenza, aumenta ulteriormente. Indicando con *nSml* il numero di simulazioni effettuate contemporaneamente, sarebbe necessario memorizzare:

- *nodeValueMaskOne* * *nSml* volte: E' necessario duplicare il vettore che memorizza le posizioni con valore romano 1 per ogni simulazione, in quanto differenti tra loro.

- *nodeValueMaskTwo* * *nSml* volte: E' necessario duplicare il vettore che memorizza le posizioni con valore romano 2 per ogni simulazione, in quanto anch'essi differenti tra loro.

- *GainFactors* * *nSml* volte: Ogni simulazione ha i suoi GainFactor associati ai vertici.

- *score*: Un nuovo vettore di *nSml* elementi, contenente per ogni simulazione il suo punteggio finale. Ciò sarebbe necessario in quanto alla fine di tutte le simulazioni è necessario conoscere quella di punteggio minore, quindi determinare quale delle iterazioni ha generato la migliore funzione di Dominazione Romana.

- *startingNodes*: Un nuovo vettore di *nSml* elementi, contenente per ogni simulazione gli indici dei vertici di partenza da cui ha preso inizio l'iterazione.

Il numero di simulazioni effettuabili contemporaneamente, dunque, è influenzato dalla memoria disponibile e dal numero di vertici presenti nel grafo. All'aumentare delle dimensioni del grafo il numero di simulazioni *nSml* possibili diminuisce (in funzione della memoria a disposizione), limitando quindi l'utilità di questo approccio. La Figura 4.10 mostra l'occupazione di memoria riguardante una simulazione fatta con matrice di adiacenza al variare del numero dei vertici. MAX indica il limite massimo (che è pari ad 1GB per la scheda nVidia Geforce 9800 GT).

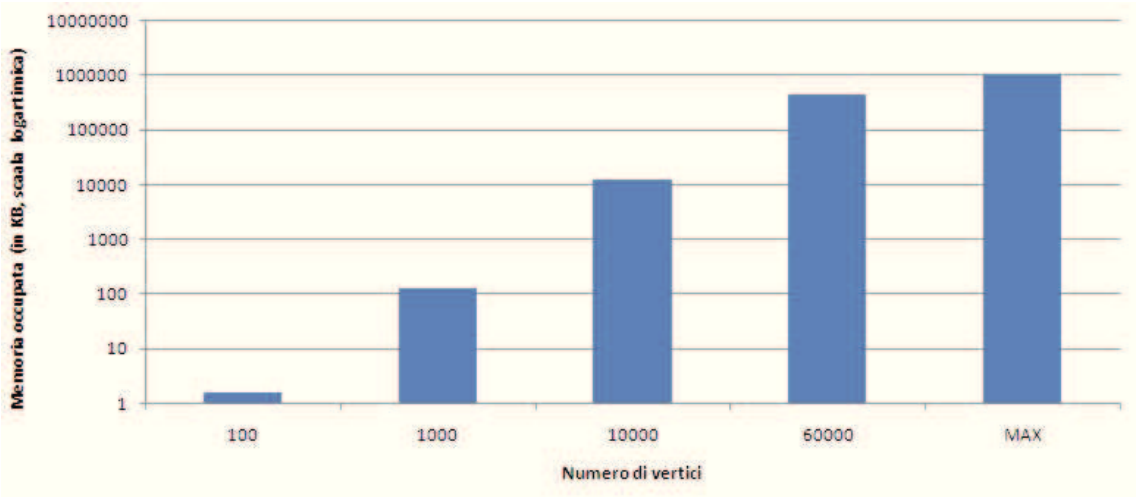


Figura 4.10: Memoria occupata con matrice di adiacenza al variare del numero di vertici.

Conclusioni

Nella prima parte del lavoro appena presentato sono stati mostrati degli schemi di copertura per grafi a griglia che permettono di migliorare il limite superiore dell numero di Dominazione Romana per tali grafi. Tali schemi godono della proprietà di non ridondanza che porta la differenza tra limite superiore e limite inferiore ad essere pari a $(m + n + 2)/5 = \Theta(m + n)$. Rimane un problema aperto capire se il numero di Dominazione Romana di un grafo a griglia di qualsiasi dimensione sia calcolabile in tempo lineare che significherebbe che il problema della Dominazione Romana non è un problema NP-completo nella sua formulazione su grafi a griglia.

Il tentativo di calcolare una buona funzione di Dominazione Romana per grafi bipartiti partendo dal loro insieme di vertex cover ha prodotto una copertura valida in tempo polinomiale ma fallisce confrontato con l'algoritmo successivamente esposto che migliora l'approssimazione del numero di Dominazione Romana anche per questa classe di grafi

In letteratura non esistono soluzioni che approssimano il numero di Dominazione Romana per un grafo di qualsiasi genere. I lavori di ricerca si sono concentrati fino ad adesso nell'individuare una buona soluzione per specifiche classi di grafi che risultano essere particolarmente regolari e nello stesso tempo interessanti per quello che rappresentano. Sicuramente il posizionamento dei server fondamentali all'interno di una rete oppure il posizionamento dei trasmettitori wireless all'interno in un edificio possono avere una progettazione che portano alla creazione di un grafo con struttura regolare. Ma come si è osservato nella fase introduttiva, le connessioni che si creano in natura danno vita a dei grafi irregolari, denominati random per la propria

natura di non riproducibilità. L'algoritmo euristico che fa uso del GainFactor vuole essere una prima soluzione a questa problematica. La complessità dell'algoritmo presentato risulta essere cubica sul numero dei vertici del grafo per grafi molto densi. C'è da considerare che per tali grafi, basterà individuare pochi vertici a cui assegnare il valore romano 2 per far in modo che tutti i vertici del grafo issano coperti. Potrebbe essere possibile produrre un'analisi ammortizzata di tale complessità in maniera da ridurre i tempi di attesa stimati.

Sono state utilizzate varie classi di grafi per valutare i risultati ottenuti ma sarebbe interessante continuare a esplorare i risultati dell'euristica su altre tipologie di grafi magari concentrandosi su quelli che si vengono a creare soprattutto nell'ambito naturale e sociologico. I possibili miglioramenti implementativi da apportare all'algoritmo esposto potrebbero essere:

1. Aggiungere informazioni al valore di GainFactor. Cioè includere delle informazioni che possano migliorare la bontà della scelta del singolo vertice.

2. Trovare un criterio per scegliere la sequenza dei vertici in caso di parità di GainFactor. Cioè a parità di GainFactor, trovare un ulteriore modo di scegliere i vertici.

3. Trovare una struttura dati adatta a rappresentare il vicinato di secondo livello di ogni vertice. In generale sarebbe interessante determinare se è possibile mantenere in memoria tali informazioni e fino a che cardinalità del grafo sarebbe possibile farlo.

Nell'ultimo capitolo si sono studiate varie ottimizzazioni dell'euristica che fa uso del GainFactor con l'obbiettivo di migliorare le prestazioni su CUDA. Sono stati fatti molti passi avanti dalla prima implementazione ma sono ancora sicuramente possibili dei miglioramenti notevoli. I lavori futuri potrebbero concentrarsi nella ricerca di strutture dati che possano migliorare i tempi di esecuzione su tale piattaforma che essendo relativamente giovane è in continua crescita per quanto riguarda la potenza di calcolo che è in grado di mettere a disposizione.

Bibliografia

[1] A-L. Barabási; R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[2] R. Albert; A-L. Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47 – 97, 2002.

[3] B. Bollobás. *Random Graphs*. Cambridge University Press, 2001.

[4] L. Mussi; F. Daolio; S. Cagnon. Evaluation of parallel particle swarm optimization algorithms within the cuda^{TM} architecture. *Information Sciences*, 181(20):4642 – 4657, 2011.

[5] C-H. Liu; G. J. Chang. Upper bounds on roman domination numbers of graphs. *Discrete Mathematics*, 2012.

[6] N. Corporation. Cuda: Compute unified device architecture programming guide. Technical report, Nvidia, 2007.

[7] V. Currò. *The Roman Domination Problem on Grid Graphs*. PhD thesis, Università degli Studi di Catania, 2013.

[8] M. Hussein; A. Varshney; L. Davis. On implementing graph cuts on cuda.

[9] C. F. de Jaenisch. Trait des applications de l'analyse mathematique au jeu des echecs. *Petrograd*, 1862.

[10] P. A. Dreyer. *Applications and variations of domination in graphs*. PhD thesis, Rutgers University, 2000.

[11] S. Wasserman; K. Faust. *Social Network Analysis*. Cambridge University Press, 1994.

[12] M. Soltys; A. Fernandez. A linear-time algorithm for computing minimum vertex covers from maximum matchings. *In press*, 2012.

[13] H. Fernau. Roman domination: A parameterized perspective. In *SOFSEM 2006: Theory and Practice of Computer Science*, volume 3831, pages 262 – 271. Springer Berlin Heidelberg, 2006.

[14] J. Arquilla; H. Fredricksen. Graphing an optimal grand strategy. *Military Operations Research*, 1995.

[15] A. Gibbons. *Algorithmic graph theory*, chapter 5. Cambridge University Press, 1985.

[16] E. N. Gilbert. *Random Graphs*, volume 30, pages 1141–1144. JSTOR, 1959.

[17] T. Y. Chang; W. E. Clark; E. O. Hare. Domination numbers of complete grid graphs, i. *Ars Combinatoria*, 38:97 – 111, 1995.

[18] E. J. Cockayne; P.A. Dreyer; S. M. Hedetniemi; S. T. Hedetniemi. Roman domination in graphs. *Discrete Mathematics*, 278:11–22, 2004.

[19] M. A. Henning; S.T. Hedetniemi. Defending the roman empire - a new strategy. *Discrete Mathematics*, 266:239–251, 2003.

[20] T. W. Haynes; P. J. Slater; S. T. Hedetniemi. *Domination in graphs : Advanced topics*. New York : Marcel Dekker, 1997.

[21] M. A. Henning. A characterization of roman trees. *Discussiones Mathematicae Graph Theory*, pages 325 – 334, 2002.

[22] M. A. Henning. Defending the roman empire from multiple attacks. *Discrete Mathematics*, 271(1 - 3):101 – 115, 2003.

[23] M. A. Henning. A survey of selected recent results on total domination in graphs. *Discrete Mathematics*, 309(1):32 – 63, 2009.

[24] W. Shang; X. Hu. The roman domination problem in unit disk graphs. In *Computational Science*, volume 4489. Springer Berlin Heidelberg, 2007.

[25] S. Ryoo; C. I. Rodrigues; S. S. Baghsorkhi; S. S. Stone; D. B. Kirk; W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.

[26] S. Davis; R. Impagliazzo. Models of greedy algorithms for graph problems. *Algorithmica*, 54(3):269 – 317, 2009.

[27] X. Fu; Y. Yang; B. Jiang. Roman domination in regular graphs. *Discrete Mathematics*, 309(6):1528 – 1537, 2009.

[28] M. R. Garey; D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[29] J. E. Hopcroft; R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[30] X. Zhu; J. Yu; W. Lee; D. Kim. New dominating sets in social networks. *Journal of Global Optimization*, 48(4):633–642, 2012.

[31] D. König. Gráfok és alkalmazásuk a determinánsok és a halmazok elméletére. *Matematikai és Természettudományi Értesítő*, pages 104–119, 1916.

[32] F. Wang; E. Camacho; K.Xu. Positive influence dominating set in online social networks. In *Combinatorial Optimization and Applications*. Springer Berlin Heidelberg, 2009.

[33] F. Wang; E. Camacho; K.Xu. On the approximability of positive influence dominating set in social networks. *Journal of Combinatorial Optimization*, pages 1–17, 2012.

[34] C. Wang; Z. Hu; X. Li. A constructive characterization of total domination vertex critical graphs. *Discrete Mathematics*, 309(4):991 – 996, 2009.

[35] M. Liedloff. Finding a dominating set on bipartite graphs. *Information Processing Letters*, 107(5):154–157, 2008.

[36] Mathieu M. Liedloff; T. Kloks; J. Liu; S-L. PengLiedloff, Ton Kloks, Jiping Liu, and Sheng-Lung Peng. Roman domination over some graph classes. In *Graph-Theoretic Concepts in Computer Science*, volume 3787, pages 103 – 114. Springer Berlin Heidelberg, 2005.

[37] S. Gravier; M. Mollard. On domination numbers of cartesian product of paths. *Discrete Applied Mathematics*, 80:247 – 250, 1997.

[38] F.V. Fomin; D. Kratsch; H. Müller. Algorithms for graphs with small octopus. *Discrete Applied Mathematics*, 134:105 – 128, 2004.

[39] E. J. Cockayne; P.J.P. Grobler; W.R. Gründlingh; J. Munganga. Protection of a graph. *Utilitas Mathematica*, 67:19–32, 2005.

[40] J. A. Bondy; U. S. R. Murty. *Graph Theory with Applications*. North Holland, 1976.

[41] P. Harish; P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, pages 197 – 208. Springer-Verlag, 2007.

[42] V. Vineet; P. J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, 2008.

[43] M.H. El-Zahar; S.M. Khamis; Kh.M. Nazzal. On the domination number of the cartesian product of the cycle of length n and any graph. *Discrete Applied Mathematics*, 155(4):515 – 522, 2007.

[44] T. Nieberg. *Independent and Dominating Sets in Wireless Communication Graphs*. University of Twente, 2006.

[45] V. Currò; V. Cutello; S.M. Nolassi. Heuristics for roman domination problem. preprint.

[46] V. Currò; V. Cutello; S.M. Nolassi. The roman domination problem on grid graphs. In *Middle-European Conference on Applied Theoretical Computer Science (MATCOS-13)*, 2013.

[47] S. Hong; S. K. Kim; T. Oguntebi; K. Olukotun. Accelerating cuda graph algorithms at maximum warp. *SIGPLAN Not.*, 46(8):267–276, 2011.

[48] N. Dixit; R. Keriven; N. Paragios. Gpu-cuts: Combinatorial optimisation, graphic processing units and adaptive object extraction. Technical report, CERTIS, 2005.

[49] B. Awerbuch; B. Berger; L. Cowen; D. Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 1998.

[50] M. Liedloff; T. Kloks; J. Liu; S-L. Peng. Efficient algorithms for roman domination on some classes of graphs. *Discrete Applied Mathematics*, 156(18):3400–3415, 2008.

[51] S. Alanko; S. Crevals; A. Isopoussu; P. R. J. Östergård; V. Pettersson. Computing the domination number of grid graphs. *The Electronic Journal of Combinatorics*, 18, 2011.

[52] L. Lovász; M. D. Plummer. *Matching Theory*, volume 367. AMS Chelsea Publishing, 2009.

[53] A. Lim; Y. Zhu; Q. Lou; B. Rodrigues. Heuristic methods for graph coloring problems. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, 2005.

[54] R. G. Downey; M. R. Fellows; C. McCartin; F. Rosamond. Parameterized approximation of dominating set problems. *Information Processing Letters*, 109(1):68 – 70, 2008.

[55] C.S. ReVelle; K.W. Rosing. Defendens imperium romanum: A classical problem in military strategy. *American Mathematical Monthly*, 107, 2000.

[56] S. Klavžar; N. Seifter. Dominating cartesian products of cycles. *Discrete Applied Mathematics*, 59(2):129 – 136, 1995.

[57] X. Song; W. Shang. Roman domination in a tree. *Ars Combinatoria*, 98:73 – 82, 2011.

[58] O. Favaron; H. Karami; R. Khoeilar; S.M. Sheikholeslami. On the roman domination number of a graph. *Discrete Mathematics*, 309(10):3447 – 3451, 2009.

[59] S. Pemmaraju; S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003.

[60] T. W. Haynes; S. Hedetniemi; P. Slater. *Fundamentals of Domination in Graphs*, volume 208. Taylor & Francis, 1998.

[61] N. Alon; J. H. Spencer. *The Probabilistic Method*. Wiley, 1992.

[62] D. G. Corneil; S. Olariu; L. Stewart. Linear time algorithms for dominating pairs in asteroidal triple-free graphs. In *Automata, Languages and Programming*. Springer Berlin Heidelberg, 1995.

[63] I. Stewart. Defend the roman empire! *Scientific American*, pages 136–139, 1999.

[64] M. E. J. Newman; D. J. Watts; S. H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences of USA*, 99:2566 – 2572, 2002.

[65] L. Jia; R. Rajaraman; T. Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Computing*, 15(4):193 – 205, 2002.

[66] M. Göös; J. Suomela. No sublogarithmic-time approximation scheme for bipartite vertex cover. In *Proceedings of the 26th International Conference on Distributed Computing*, pages 181 – 194, 2012.

[67] J. Hopcroft; R. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549 – 568, 1974.

[68] D. Gonçalves; A. Pinlou; M. Rao; S. Thomassé. The domination number of grids. *SIAM Journal on Discrete Mathematics*, 25(3), 2011.

[69] D. Lee; I. Dinov; B. Dong; B. Gutman; I. Yanovsky; A. W. Toga. Cuda optimization strategies for compute and memory-bound neuroimaging algorithms. *Computer Methods and Programs in Biomedicine*, 106(3):175 – 187, 2012.

[70] G. Di Battista; P. Eades; R. Tamassia; I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, 1998.

[71] A. Hansberg; L. Volkmann. Upper bounds on the k-domination number and the k-roman domination number. *Discrete Appl. Math.*, 1457(7):1634 – 1639, 2009.

[72] K. Kaemmerling; L. Volkmann. Roman k-domination in graphs. *Journal of the Korean Mathematical Society*, 46(6):1309 – 1318, 2009.

[73] M. Kaufmann; D. Wagner. *Drawing graphs: methods and models*. Springer-Verlag, 2001.

[74] D. Z. Du; P. J. Wan. *Connected Dominating Set: Theory and Applications*. Springer, 2013.

[75] F. Kuhn; R. Wattenhofer. Constant-time distributed dominating set approximation. In *In Proc. of the 22 nd ACM Symposium on the Principles of Distributed Computing (PODC)*, pages 25 – 32, 2003.

[76] E. W. Chambers; B. Kinnersley; N. Prince; D. B. West. Extremal problems for roman domination. *SIAM Journal on Discrete Mathematics*, 23(3):1575 – 1586, 2009.

[77] A. Pagourtzis; P. Penna; K. Schlude; K. Steinhöfel; D. S. Taylor; P. Widmayer. Server placements, roman domination and other dominating set variants. *IFIP TCS Conference Proceedings*, 271:280 – 291, 2002.

[78] N. C. Wormald. Analysis of greedy algorithms on graphs with bounded degrees. *Discrete Mathematics*, 273(1 - 3):235 – 260, 2003.

[79] J. Chen; L. A. Kanj; G. Xia. *Improved Parameterized Upper Bounds for Vertex Cover*, volume 4162, pages 238–249. Springer Berlin Heidelberg, 2006.

[80] J. Wu; M. Cardei; F. Dai; S. Yang. Extended dominating set in ad hoc networks using cooperative communication. *Parallel and Distributed Systems, IEEE Transactions on*, 17(8):851–864, 2006.

[81] P. Pavlic; J. Zerovnik. Roman domination number of the cartesian products of paths and cycles. *The Electronic Journal of Combinatorics*, 19(3), 2012.

[82] I. Blöchliger; N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers and Operations Researc*, 35(3):960 – 975, 2008.

Ringraziamenti

Il Dottorato di Ricerca rappresenta il più alto grado di istruzione previsto nell'ordinamento accademico italiano. Sono veramente orgoglioso di averne potuto far parte e di aver potuto usufruire del percorso formativo che mette a disposizione.

Vorrei sentitamente ringraziare il mio supervisore, il Prof. Vincenzo Cutello, persona eccezionale sia dal punto di vista umano che accademico. Senza la sua guida, nulla sarebbe stato possibile. Un ringraziamento speciale va a Vincenzo Currò con cui ho condiviso il percorso formativo e l'argomento della mia ricerca.

I miei ringraziamenti vanno anche a tutte le persone che hanno curato la mia formazione nel corso degli anni. La maestra Giuseppina Russo, unica insegnante delle scuole elementari, il professore Terlato, che mi ha fatto scoprire per primo la mia attitudine alla matematica alle scuole medie, il professore Zanchi e la professoressa Sciuto, professori e maestri di vita.

Un mio grazie va anche al Dipartimento di Matematica e Informatica di Catania che mi ha ospitato in questi anni, mettendo a disposizione spazi e risorse necessarie per la mia ricerca.

Vorrei ringraziare anche tutti gli amici e colleghi che mi hanno supportato e incoraggiato in questi anni. Grazie a Alessandro, Cristiano, Giuseppe, Jole, Lorenzo, Luigi, Marco, Raffaela e tanti altri, il mio dottorato è stato anche occasione di arricchimento dello spirito.

Il ringraziamento più grande va ai miei genitori, per avermi cresciuto con i valori che mi hanno permesso di diventare quello che sono oggi.

Finito di stampare il 10 dicembre 2013 utilizzando LaTeX 2ε