Politecnico di Torino

Politecnico di Torino

Laurea Magistrale in

Ingegneria Informatica - Grafica e Multimedia

A.a. 2020/2021

Sessione di Laurea Dicembre 2021

Problema del Massimo Comune Sottografo: Concorrenza ed Estensioni per Multipli Grafi

Implementazioni many-core ed

estensioni dell'algoritmo McSplit

Relatore:

Dr. Stefano Quer

Professore Associato

Candidato:

Lorenzo Cardone

Matr. 278037

Abstract

Il problema del Massimo Comune Sottografo (MCS) è ben noto in letteratura. Esso consiste nel trovare il più ampio grafo che è simultaneamente isomorfo rispetto a due grafi dati. Trovare tale grafo ha complessità NP-hard e, pertanto, ci si è spesso limitati a ricercare delle soluzioni approssimate, ovvero di dimensione non massima. Nonostante la complessità del problema, grazie anche al costante aumento della potenza di calcolo dei dispositivi, diversi approcci moderni tentano di ottenere una soluzione esatta. Inoltre, in alcuni ambiti il problema è stato esteso alla gestione di più di due grafi. Ad esempio nel campo della scienza molecolare [1] può essere utile confrontare diverse molecole contemporaneamente.
Questa tesi si concentra sull'estensione dell'algoritmo McSplit, un algoritmo di partizionamento per i problemi di massimo comune sottografo tra i più efficienti oggi disponibili. I nostri sforzi si sono orientati in due principali direzioni: estendere l'algoritmo originale, in modo tale da permettere di risolvere il problema del massimo comune sottografo tra su un numero n di grafi e migliorare l'implementazione many-core [2] tramite una più efficace selezione dei nodi su cui proseguire la ricerca, pur senza estenderla oltre i due grafi.
Inoltre abbiamo esplorato la possibilità di utilizzare le GPU, componenti hardware in rapida evoluzione, che ad oggi vedono un limitato impiego in molti processi. Sfruttando quindi una GPU cercheremo di determinare se sia possibile migliorare le prestazioni degli algoritmi analizzati.

Abbiamo prodotto tre versioni esatte, di crescente efficienza, estendendo l'algoritmo McSplit [3] per gestire n grafi ed una quarta versione che, pur producendo un risultato che non assicura la soluzione esatta, ha il pregio di fornire soluzioni mediamente accurate in tempi estremamente contenuti.

Ci siamo poi concentrati sulla versione many-core, raggiungendo un elevato aumento delle prestazioni su coppie di grafi al crescere del tempo di soluzione. La versione attuale dell'algoritmo non è ancora in grado di competere con l'implementazione originale di McSplit in un caso generale, ma sono state individuate possibili evoluzioni del software in grado di apportare significative riduzioni dei tempi di calcolo.

Sommario

Abstract 2
Capitolo 1 - Introduzione 6
1.1 Definizioni 6
1.2 Complessità 8
Capitolo 2 - Problema del MCS 11
2.1 Applicazioni 11
2.2 Approcci noti 12
Programmazione a vincoli 12
Inferenza 12
Branch and bound 13
2.3 Estensione al caso n grafi 14
Capitolo 3 - Stato dell'arte 15
Capitolo 4 – Background 19
4.1 McSplit 19
Definizioni 19
Algoritmo 22
Esecuzione tipo 24
Capitolo 5 - Implementazioni CPU 30
5.1 Versione single-thread per multipli grafi 30
5.2 Versione multi-thread per multipli grafi 32
Considerazioni 33
5.3 Versione multi-thread per multipli grafi approssimata 33
Considerazioni 36
5.4 Problemi affrontati 38
Adattamento ad n grafi di McSplit parallelo 38
Capitolo 6 - Architettura GPU 42
6.1 Introduzione a CUDA 42
Capitolo 7 - Implementazioni GPU 47
7.1 Implementazioni esistenti 47
7.2 Miglioramenti 48

Considerazioni 53
Capitolo 8 – Risultati 57
8.1 Risultati per n grafi delle versioni per CPU 57
8.2 Risultati dell'approccio su architettura many-core 62
Capitolo 9 – Conclusioni 64
9.1 Soluzione di insieme di grafi su architetture multi-core 64
9.2 Soluzione di insieme di grafi su architetture many-core 64
9.3 Soluzione di coppie di grafi su architetture many-core 65
Indice delle immagini 66
Indice degli algoritmi ed estratti di codice 67
Bibliografia 68

Capitolo 1 - Introduzione

I grafi sono delle strutture dati estremamente generali e flessibili. Molti altri problemi possono essere riformulati per sfruttare le caratteristiche della rappresentazione a grafo. I problemi del Massimo Comune Sottografo (MCS) sorgono in numerose aree, tra cui computer vision, analisi del codice, compilatori ecc.

Con l'incremento della potenza di calcolo il problema del MCS è stato esteso per prendere in considerazione un numero di grafi maggiore di due e quindi trovare il massimo sottografo comune tra un numero arbitrario di grafi di partenza.

In questa tesi si prendono in considerazione più approcci sia nell'ambito della parallelizzazione su CPU che su GPU.

1.1 Definizioni

La terminologia, nella teoria dei grafi, nei vari testi non è uniforme. Ho ritenuto opportuno anticipare queste definizioni alle quali farò riferimento nel contesto di questo lavoro.

Definizione 1.1 - Un **grafo** G è una coppia ordinata di insiemi $G = \{V_G, E_G\}$.

V_G è un insieme di entità dette **vertici**.

Il prodotto cartesiano di V_G con V_G indicato con $V_G \times V_G = V_G^{[2]}$ è l'insieme di tutte le coppie <v,w> con v e w ϵ V_G.

E_G è un sottoinsieme di $V_G^{[2]}$ e i suoi elementi sono detti **archi**.

Se almeno una delle coppie <v,w> o <w,v> ϵ E_G allora v e w sono detti vertici **adiacenti**.

Qualunque arco <v,v> è detto **cappio**.

Definizione 1.2 - Un grafo è detto **non diretto** se, per ogni coppia <v,w> in E_g è anche presente la coppia simmetrica <w,v>, altrimenti il grafo è **diretto**.

Definizione 1.3 - L'**ordine** di un grafo G è la cardinalità di V_G e la **dimensione** di G è la cardinalità di E_G.

Definizione 1.4 - L'insieme di tutti i vertici N(G, v) adiacenti ad un vertice v è detto **intorno** di v.

Si dice **grado** del nodo v la cardinalità del suo intorno.

Definizione 1.5 - Quando E_G contiene ogni possibile combinazione non ordinata di vertici in V_G il grafo è detto **cricca**.

Definizione 1.6 - Se ad un vertice o ad un arco sono associate altre informazioni (come pesi, distanze, colori…) il grafo è detto **etichettato**.

Definizione 1.7 - Un grafo H è detto **sottografo** di G se $V_H \subseteq V_G$ e $E_H \subseteq E_G$.

Se H contiene tutti gli archi di G allora H è detto sottografo indotto, altrimenti è detto **non indotto** o **parziale**.

Definizione 1.8 - Due grafi G ed H sono detti isomorfi se esiste una funzione biunivoca $f : V_G \rightarrow V_H$ tale per cui, una coppia di vertici <v, w> è appartenente ad E_G se e solo se la coppia <f(v), f(w)> appartiene ad E_H.

Esempi

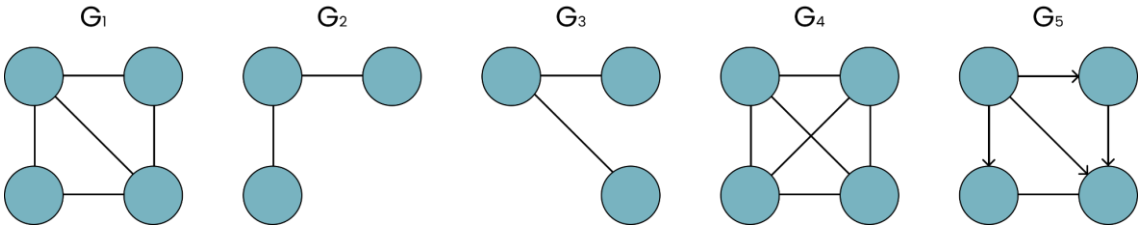

Figura 1: Esempi di grafi

G_1 - Grafo non diretto (di ordine 4 e dimensione 5)

G_2 - Sottografo indotto di G_1

G_3 - Sottografo non indotto di G_1

G_4 - Cricca

G_5 - Grafo diretto

1.2 Complessità

Risolvere il problema del massimo comune sottografo in maniera esaustiva è un problema di elevata complessità, appartiene infatti ad una famiglia di problemi definiti NP-hard, ovvero problemi che per essere risolti con un computer basato su una macchina di Turing deterministica possono richiedere un tempo esponenziale rispetto alla dimensione dell'input. Tornando al problema del massimo sottografo comune la crescita di complessità è collegata al numero di vertici dei grafi in input e quindi risolvere grafi di ordine elevato si è rivelato estremamente dispendioso.
Ad analizzare la complessità dei problemi si è dedicata una branca della teoria della calcolabilità: la teoria della complessità computazionale. Per comprendere la complessità del problema daremo quindi le seguenti definizioni.

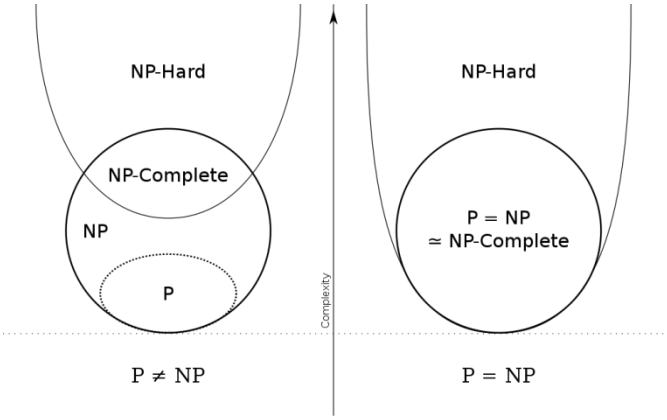

Figura 2: By Behnam Esfahbod, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3532181

Una macchina di Turing (MdT) deterministica è una macchina ideale tale che, dato uno stato interno ed un input, è in grado di calcolare un output deterministico, come i computer odierni.
Un problema è definito di complessità P o polinomiale se una MdT deterministica è in grado di risolverlo in una quantità di tempo polinomiale rispetto alla dimensione dei suoi input. Pertanto dato un problema con un input di dimensione

n, possiamo risolverlo con un numero di istruzioni che può essere calcolato come una funzione polinomiale di n.

Un problema è detto di complessità NP, o polinomiale non deterministica, se è possibile risolvere un tale problema in un tempo polinomiale sfruttando una MdT non deterministica. Una tale macchina, dato un input ed uno stato, sarebbe in grado di produrre diversi output e quindi avrebbe un'esecuzione non deterministica, in quanto un dato stato ed un corrispondente input non possono essere mappati ad un unico output. Il procedimento di una MdT non deterministica può essere assimilato al procedimento di un insieme di cardinalità crescente di MdT deterministiche, che aumentano di numero con il procedere dell'esecuzione.

La teoria della complessità computazionale descrive inoltre la possibilità di ridurre un problema ad un altro tramite l'applicazione di specifiche trasformazioni sui dati in ingresso. In tal modo un problema di una certa complessità può essere risolto sfruttando la soluzione di un problema di minore complessità, assunto che esista un modo di convertire gli input del primo problema in adeguati input per il secondo. Questo permette di risolvere problemi di grande complessità sfruttando soluzioni preesistenti per problemi più semplici.

Un problema di dice NP-completo se è un problema al quale è possibile ridurre qualunque altro problema di complessità NP. Essere in grado di risolvere uno di questi problemi con una complessità polinomiale permetterebbe quindi di risolvere qualunque problema NP con una complessità P. Al giorno d'oggi non esiste però nessuna dimostrazione che un problema NP completo possa essere ridotto ad un problema polinomiale né è mai stato dimostrato che ciò non sia possibile.

Un problema si dice, infine, NP-difficile o NP-arduo se esso si annovera tra i problemi più complessi da risolvere nello spettro dei problemi P ed NP. Risolvere un problema NP-arduo è non meno difficile che risolvere un problema di complessità NP-completo, che sono a loro volta problemi NP-difficili. I problemi

NP-completi sono l'intersezione tra i problemi NP e i problemi NP-ardui, ma i problemi NP-difficili sono un soprainsieme proprio dei problemi NP-completi.
Il problema del massimo comune sottografo appartiene alla classe di complessità NP-difficile e come tale ad oggi non esiste nessun metodo noto per risolverlo in un tempo polinomiale.
Per affrontare problemi di tale complessità ci si è spesso ridotti a delle approssimazioni. Esistono algoritmi in grado di approssimare con una data precisione la soluzione di alcuni problemi NP-difficili con una complessità polinomiale. Sfortunatamente non solo non esiste alcun algoritmo in grado di approssimare questo problema con una data precisione ma è stato provato nel 1992 da Kann [4] che appartiene alla famiglia di problemi per cui vale il principio di complessità di approssimazione e, pertanto, a meno di essere in grado di provare che la complessità NP sia riducibile a P, anche approssimare queste soluzioni richiede una complessità NP.
Dal momento che non esiste un sistema per approssimare la soluzione al problema MCS con una complessità polinomiale molti sforzi sono stati diretti verso la ricerca di algoritmi efficienti per la soluzione esatta.
Gli algoritmi che proporremo nei prossimi capitoli sono tutti di natura esatta, eccetto nel caso della ricerca su più di due grafi dove studieremo la possibilità di applicare un approccio non esatto alla ricerca della soluzione anche a costo di non poter garantire un'approssimazione di una determinata qualità e studieremo la perdita di precisione in funzione dell'incremento di prestazioni che questo approccio è in grado di fornire.

Capitolo 2 - Problema del MCS

Il problema del massimo comune sottografo è stato ampiamente studiato in letteratura ed ha ricevuto diverse definizioni in base a cosa si sta cercando di massimizzare. Vi sono infatti due principali versioni di questo problema: massimizzare il numero di nodi oppure massimizzare il numero di archi che il sottografo comune deve possedere.

In questa tesi tratteremo il primo caso e pertanto la definizione del problema che seguiremo sarà la seguente:

- Il massimo comune sottografo di una coppia di grafi G ed H è un grafo contenente il massimo numero di nodi possibile che è contemporaneamente indotto ed isomorfo sia a G che ad H.

In base all'applicazione del problema potremmo voler tenere anche conto della direzione degli archi tra i vari nodi o delle etichette su nodi ed archi. In tali casi ulteriori vincoli saranno utilizzati nell'abbinare i vertici di un grafo a quelli dell'altro.

2.1 Applicazioni

Grazie alla flessibilità dei grafi è possibile modellare un grande numero di problemi in modo da poterli risolvere con tecniche sviluppate per i grafi. Diversi campi di studi hanno trovato quindi applicazioni per il problema del massimo sottografo comune, in particolar modo il campo della medicina e lo studio di strutture molecolari [3] [5].

Ulteriori studi hanno evidenziato come le applicazioni si possano estendere a molti altri campi, come la sicurezza informatica [6] [7], la computer vision [8], l'analisi di codice sorgente e di circuiti elettronici, analisi di dataset ecc.

Il caso n grafi ha trovato applicazioni prevalentemente nel campo delle scienze molecolari, dove era necessario comparare allo stesso tempo numerose molecole per verificare la presenza di sottostrutture comuni.
La ricerca del MCS può essere applicata a molti problemi in cui non è solamente necessario verificare la presenza o meno di un elemento ma anche la disposizione dello stesso rispetto agli altri.

2.2 Approcci noti

Il problema del massimo comune sottografo è studiato fin dagli anni '70 e diversi approcci sono sorti nel tentativo di risolverlo con maggiore efficienza.

Programmazione a vincoli

Forse il sistema più semplice per risolvere questo problema è la programmazione a vincoli: si crea un insieme contenente tutte le possibili soluzioni, ottenute associando i nodi di un grafo con tutte le permutazioni dei nodi del secondo, e si chiede al programma per ciascuna di esse se sia valida o meno. Per quanto questo metodo permetta di ottenere una soluzione corretta, il semplice numero di possibili combinazioni dei nodi di due grafi rende l'approccio non pratico per risolvere grafi che hanno più di una manciata di nodi.

Inferenza

Estendendo l'idea della programmazione a vincoli cerchiamo di ridurre il numero di combinazioni ancora prima di verificare se gli accoppiamenti proposti siano validi o meno. Ci possono essere diverse euristiche per limitare quali nodi del primo grafo possono essere abbinati a quali del secondo.
Un semplice esempio: si supponga di avere una soluzione parziale del sottografo comune e di voler aggiungere ad essa una coppia di vertici. Il semplice conteggio dei nodi appartenenti alla soluzione adiacenti a ciascuno dei vertici della coppia ci

permette di inferire la validità dell'abbinamento. Se un nodo selezionato sul primo grafo non ha adiacenze ai nodi appartenenti alla soluzione non potrà mai essere abbinato ad un nodo del secondo grafo adiacente ad uno o più nodi della soluzione. Sfortunatamente, sebbene questo sistema sia in grado di diminuire anche in quantità elevata il numero di combinazioni possibili esso richiede un tempo di calcolo non indifferente e le implementazioni che si sono rivelate migliori non ricadono tra quelle in grado di rimuovere il massimo numero di combinazioni ma quelle che bilanciano al meglio il costo computazionale e la capacità di pruning dell'approccio.

Branch and bound

Nel 1982 McGregor [9] propose di applicare al problema del massimo comune sottografo un procedimento basato sulla logica branch and bound. Il procedimento si basa su una **funzione di bound**, ovvero una funzione in grado di stimare il massimo risultato raggiungibile proseguendo dallo stato attuale della ricerca, mentre definisce come **ramo** ciascun abbinamento di una coppia di nodi. L'approccio si basa quindi sulla selezione di due nodi e successivamente sul calcolo del bound data la soluzione raggiunta. A questo punto, l'algoritmo è in grado di proseguire aggiungendo una nuova coppia di nodi nel caso in cui il bound calcolato sia superiore alla dimensione finora raggiunta. In caso contrario, scarta questo ramo della ricerca eliminando l'ultima selezione di nodi e cerca una nuova coppia in grado di produrre un bound più soddisfacente.

Questo approccio è quello che al giorno d'oggi ha ottenuto il maggiore successo ed è stato poi raffinato nei lavori di Krissinel e Henrick [10] e ancora nell'algoritmo McSplit di McCreesh, Prosser e Trimble [3].

Questo approccio è poi anche stato esteso nella versione k-down, dove il bound è impostato a priori ad un valore pari al numero di nodi del grafo più piccolo e, ad ogni esecuzione che non riesce a trovare una soluzione soddisfacente il valore del bound è ridotto di 1 fino a convergere al massimo comune sottografo. Quest'ultima

variante ha il vantaggio di scartare con maggiore rapidità i rami di dimensione non massima ma che possiedono un bound in grado di superare la dimensione richiesta senza mai raggiungerla veramente.

2.3 Estensione al caso n grafi

Il caso dove è richiesto risolvere più di due grafi contemporaneamente è, ad oggi, ancora poco studiato. Alcuni autori hanno indicato l'utilità nel trovare una soluzione a questo problema, ma hanno spesso fallito nel proporre una loro soluzione o hanno seguito degli approcci con risultati insoddisfacenti.
Una di queste soluzioni consisteva nell'enumerare tutti i possibili sottografi in ordine di dimensione crescente per ciascun grafo per poi tentare di ritrovare queste strutture negli altri grafi [11]. La complessità della ricerca era poi contenuta producendo solamente i sottografi di dimensione m estendendo quelli di dimensione m-1 di cui si era verificata la presenza in tutti i grafi. Altri lavori, basati su questo algoritmo, hanno poi modificato il metodo di ricerca di sottografi comuni tra i grafi da comparare sfruttando vari metodi, dalla generazione di valori tramite funzioni di hashing alla ricerca di sottostrutture [12] [13].

Capitolo 3 - Stato dell'arte

Come già descritto nel capitolo precedente McGregor [9] propose un algoritmo basato sulla tecnica branch and bounds. In questo procedimento ciascun ramo rappresenta un possibile abbinamento di nodi che continua a venir esteso fino a quando una funzione di bound, che stima quanti nodi possono ancora essere aggiunti alla soluzione, non ritorna un valore per quel ramo pari o inferiore alla massima dimensione raggiunta fino a quel punto. Nel 2004 questo approccio è poi stato rivisitato e raffinato da Krissinel ed Henrick [10] che, pur mantenendo la struttura dell'algoritmo, hanno proposto una nuova e più efficiente funzione di ricerca degli abbinamenti.

L'applicazione di strategie di programmazione a vincoli può migliorare ulteriormente questi algoritmi. L'approccio di backtracking di McGregor ha molte similitudini con il framework di problemi di soddisfazione di vincoli, come citato nell'introduzione di Vismara e Valery [14] nel loro articolo sulla ricerca di massimi sottografi comuni connessi. Nello stesso articolo si presenta il primo modello esplicito di programmazione a vincoli. Dati due grafi G ed H il loro approccio crea un abbinamento tra ciascuno dei vertici v di G con i vertici di H, oppure con uno speciale valore che indica il non abbinamento. La mappatura di ciascun vertice di G con uno di H o il valore speciale utilizza un sistema di vincoli sugli archi per garantire l'adiacenza e la non adiacenza di ciascuno dei vertici abbinati.

Un ulteriore sviluppo di questo approccio di programmazione a vincoli è stato sviluppato da Ndiaye e Solnon [15]. Questo ultimo lavoro si è concentrato sulla sostituzione dei vincoli imposti dal lavoro precedente con un vincolo globale di completa differenza, nel tentativo di massimizzare i nodi di G assegnati ad H, pur mantenendo la limitazione che gli abbinamenti devono essere completamente diversi, quando non assegnati alla variabile speciale di non abbinamento. Questo ultimo lavoro ha inoltre evidenziato che diverse tecniche per mantenere la correttezza della soluzione erano vantaggiose in diverse situazioni. Nello specifico,

per un grafo privo di etichette, i vincoli sugli archi hanno sperimentalmente dimostrato di produrre i migliori risultati, mentre per grafi etichettati le migliori performances erano ottenute da un algoritmo di forward checking. In entrambi i casi l'algoritmo di bound applicato per determinare se è possibile assegnare un numero sufficiente di nodi per superare la dimensione attuale raggiunta è basato su un processo di matching derivato dalla semplificazione di un precedente algoritmo di verifica della consistenza tra gli archi di Petit, Régin e Bessière [16].
A partire da questi studi, uno degli sviluppi più recenti è stato presentato nel lavoro di McCreesh, Trimble e Prosser [3]. Essi hanno prodotto una soluzione in grado di ottimizzare la replicazione dei dati e un più efficiente sistema di backtraking per velocizzare il processo di ricerca.
Altri approcci sono stati sviluppati valutando anche differenti tecniche, come l'applicazione di reti neurali affiancate ad un insieme di vincoli per garantire la correttezza della soluzione trovata [17] [18].
La complessità del problema e delle tecniche utilizzate non hanno comunque consentito di raggiungere un breakthrough, ma questo campo di ricerca è ancora ampiamente studiato al giorno d'oggi.

Come descrivono Hariharan e al. nel 2011 [1] il caso per multipli grafi è prevalentemente applicato alla scienza molecolare e molti degli approcci esistenti si basano sul selezionare una sottostruttura presente in una molecola o comune tra due e poi verificarne la presenza nelle altre. Vista la limitata letteratura in altri campi, per il resto del capitolo, faremo alternatamente riferimento a molecole o a grafi, dal momento che le prime sono modellate tramite queste strutture.
Come già anche citato in precedenza i lavori più influenti in questo campo sono quelli di Varkony e al. [11] dove i sottografi erano generati a partire da sottografi noti per essere comuni a tutti i grafi presi in considerazione, in modo da limitare il numero di possibili sottostrutture tra cui ricercare.

Altri lavori si sono concentrati su sistemi per rendere più efficiente la ricerca di questi sottografi all'interno di altri grafi, generando delle chiavi tramite funzioni di hashing [12] [13].
Ritornando all'idea originale di limitare la ricerca di strutture comuni ad un paio di molecole alla volta troviamo il lavoro della ChemAxon [19] che ha sviluppato un approccio chiamato Library MCS che utilizza la ricerca su due grafi per ottenere la lista di sottografi da ricercare con un algoritmo apposito sulle restanti molecole.
Come faremo anche noi all'interno di questa tesi, Hiriharan mostra che non è sufficiente applicare un algoritmo di MCS su coppie di grafi per riuscire ad ottenere il massimo comune sottografo tra N grafi e quindi questi algoritmi devono essere adattati per poter gestire più di due grafi e garantire di trovare la soluzione di dimensione massima. L'autore dell'articolo propone poi un suo algoritmo basato sulla ricerca di tutte le strutture di dimensione massima trovate comparando ogni altra molecola con una scelta come "pivot". Le strutture così prodotte diventano degli elementi che vengono ricercati in ogni altra molecola e viene conservata la struttura di dimensione massima ottenuta tramite questo procedimento. Tutta una serie di ottimizzazioni sono poi implementate, come un'efficiente scelta della molecola pivot ad un approccio dividi e conquista, che spezza le molecole per raggiungere più rapidamente una soluzione, senza però perdere in correttezza della soluzione trovata.

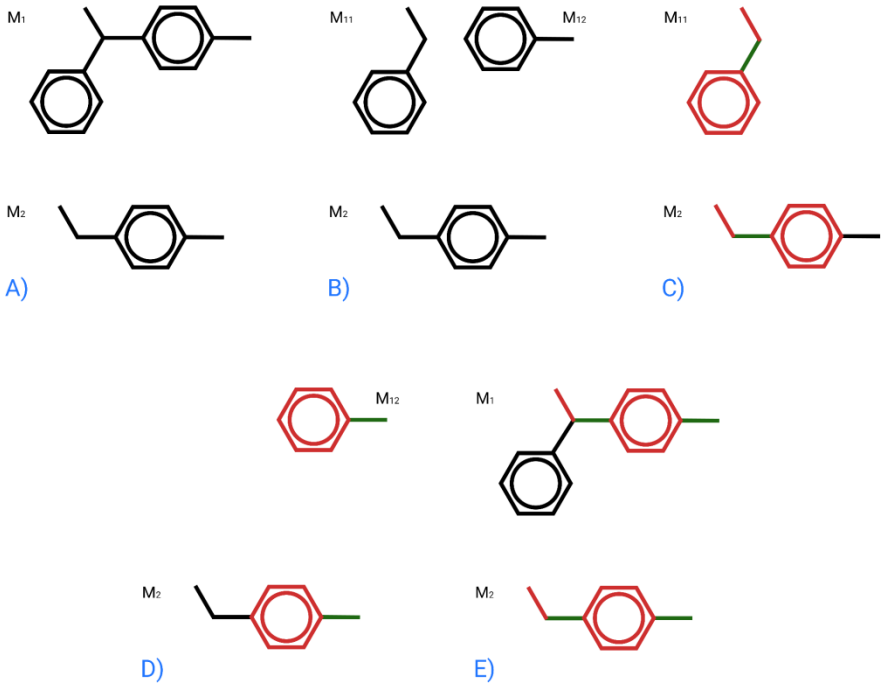

Figura 3: approccio dividi e conquista di Hariharan [1]

Come illustrato nella precedente immagine vediamo al punto A le due molecole che vogliamo comparare: M_1 e M_2. Al punto B M_1 è stata divisa in due parti: M_{11} e M_{12}. Al punto C viene calcolato il MCS tra M_{11} e M_2. Al punto D viene calcolato il MCS tra M_{12} e M_2. Infine in E i M_{11} e M_{12} vengono riuniti in un unico grafo e le soluzioni trovate vengono estese al fine di ottenere il massimo comune sottografo dell'intera molecola.

Infine, nel 2012 Dalke e Hastings [20] hanno pubblicato un articolo, dove studiano un nuovo algoritmo basato sull'enumerazione di sottografi e testing dell'isomorfismo per trovare il massimo comune sottografo. Rispetto ai precedenti approcci propongono miglioramenti degli algoritmi esistenti e nuove euristiche per competere con i precedenti metodi.

Capitolo 4 – Background

Nel capitolo precedente abbiamo presentato numerosi approcci che nell'arco degli anni hanno portato innovazioni nell'ambito della soluzione del problema del MCS. Uno degli algoritmi trattati era McSplit, sviluppato da McCreesh, Trimble e Prosser, che ha fortemente influenzato il lavoro oggetto della nostra tesi.
In questo capitolo analizzeremo il funzionamento dell'algoritmo McSplit. Partiremo da un insieme di definizioni introduttive, per chiarire l'uso dei termini utilizzati, seguito da una spiegazione dell'algoritmo McSplit e da un esempio di esecuzione tipo per illustrare il procedimento presentato.

4.1 McSplit

Definizioni

Definizione 4.1 - Dato un sottoinsieme di nodi appartenenti ad un grafo, si dice **classe di adiacenza**, o semplicemente **classe**, l'insieme di tutti i nodi dell'insieme complementare a quello scelto che condividono lo stesso stato di adiacenza con i nodi selezionati. La relazione di adiacenza dei nodi del grafo con quelli selezionati genera una relazione di equivalenza che crea una partizione dell'insieme complementare. La classe di adiacenza è un concetto astratto comune a più grafi, in quanto è applicabile nello stesso modo e con analoghi risultati a tutti i grafi che si vogliono confrontare.
Definizione 4.2 - Per tali motivi, per ogni grafo, è utile indicare con il termine di **dominio** ogni sottoinsieme della partizione, i cui nodi sono associati fra loro dalla stessa classe di adiacenza. Pertanto, ad esempio, per n grafi distinti avremo, per ciascuna classe, n domini (uno per ogni grafo). Più in particolare, nel caso di due grafi, avremo due domini per ogni classe e questa struttura è detta **bidominio**

Definizione 4.3 - I nodi di due o più grafi associati dalla stessa classe di adiacenza sono detti **compatibili.**

Definizione 4.4 - Si dice **soluzione** un insieme di coppie di nodi compatibili appartenenti a due grafi diversi che possono essere abbinati per formare un sottografo comune. Nel caso di n grafi la soluzione sarà costituita da n-ple di nodi compatibili.

Per chiarire meglio il significato dei termini oggetto delle precedenti definizioni presentiamo alcuni esempi.

Consideriamo il grafo della figura e scegliamo, del tutto arbitrariamente, uno dei suoi nodi, ad esempio, il nodo 1 (ma può andare bene qualunque altro nodo).

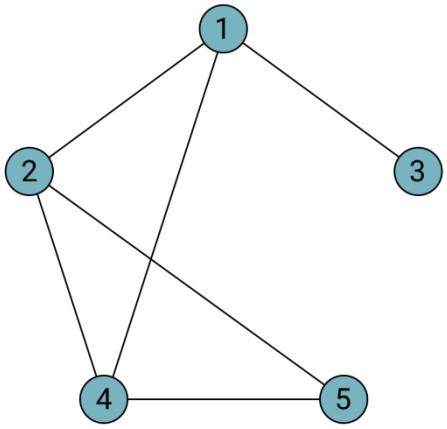

Figura 4: grafo di esempio

La definizione di adiacenza (def. 1.1) applicata all'insieme degli altri nodi rispetto al nodo scelto crea una partizione in due domini, appartenenti alle due classi di adiacenza, che contrassegniamo con le cifre binarie 1 (adiacenza) e 0 (non adiacenza). Al primo dominio appartengono, evidentemente, i nodi 2, 3 e 4 ($D_1 = \{2, 3, 4\}$), mentre al secondo il nodo 5 ($D_0 = \{5\}$).

Aggiungiamo adesso alla selezione iniziale un secondo nodo, ad esempio il nodo 2. Per contrassegnare le classi di adiacenza occorrono, in questo caso, due cifre binarie:

- 11 individua la classe dei nodi adiacenti sia al nodo 1 che al nodo 2 ($D_{11} = \{4\}$)
- 10 individua la classe dei nodi adiacenti al nodo 1 ma non al nodo 2 ($D_{10} = \{3\}$)
- 01 individua la classe dei nodi non adiacenti al nodo 1 ma al nodo 2 ($D_{11} = \{5\}$)
- 00 individua la classe dei nodi adiacenti sia al nodo 1 che al nodo 2 ($D_{11} = \{\ \}$)

Se alla selezione ora aggiungiamo un terzo nodo saranno necessarie etichette di 3 cifre binarie per contrassegnare le nuove otto classi di adiacenza (111, 110, …, 000) con i rispettivi domini.
Nel prossimo paragrafo affrontiamo la ricerca del massimo sottografo comune a due grafi per mezzo dell'algoritmo McSplit. Occorre allora estendere le considerazioni fatte per un grafo al caso in cui si vogliano confrontare due o più grafi diversi.

Accostiamo dunque al precedente un secondo grafo come nella Figura 5.

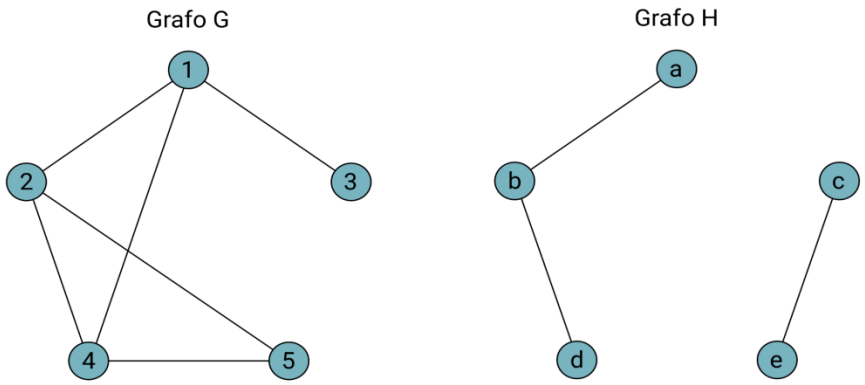

Figura 5: coppia di grafi di esempio

Con la solita arbitrarietà scegliamo il nodo **1** e il nodo **a** nei due grafi ipotizzando l'associazione (mappatura) {(1,a)} come primo embrione di soluzione e

determiniamo, con il precedente criterio delle adiacenze i domini corrispondenti a tale scelta:

- per il grafo G: $D^G_1 = \{2, 3, 4\}$, $D^G_0 = \{5\}$;
- per il grafo H: $D^H_1 = \{b\}$, $D^H_0 = \{c,e,d\}$.

I nodi del dominio D^G_1 sono compatibili con i nodi del dominio D^H_1 perché appartengono alla stessa classe (dei nodi adiacenti rispettivamente a **1** e ad **a**). I nodi del dominio D^G_0 sono compatibili con i nodi del dominio D^H_0 perché appartengono alla stessa classe (dei nodi non adiacenti rispettivamente a **1** e ad **a**).

Ora aggiungiamo una nuova coppia di nodi alla precedente soluzione prendendoli dai domini compatibili. Scegliamo (come diremo più dettagliatamente nel paragrafo descrittivo dell'algoritmo McSplit) dal dominio di cardinalità minore del grafo G, D^G_0 , l'unico nodo presente 5 e dal corrispondente nella compatibilità del grafo H, $D^H_0 = \{c,e,d\}$, il nodo di etichetta minore c, formando la coppia (5,c) che estende così la soluzione con la seguente mappatura {(1,a), (5,c)}. Analizziamo, come prima, le adiacenze in ciascun grafo.

- per il grafo G: $D^G_{11} = \{2, 4\}$, $D^G_{10} = \{3\}$, $D^G_{01} = \{\}$, $D^G_{00} = \{\}$;
- per il grafo H: $D^H_{11} = \{\}$, $D^H_{10} = \{b\}$, $D^H_{01} = \{e\}$, $D^H_{00} = \{d\}$.

Notiamo che i nodi del dominio $D^G_{11} = \{2, 4\}$ non hanno corrispondenti in D^H_{11} , né, per la stessa ragione vi è corrispondenza tra i nodi di $D^G_{01} = \{\}$ e $D^H_{01} = \{e\}$, nonché per $D^G_{00} = \{\}$ e e $D^H_{00} = \{d\}$ ma ciò non capita per i domini $D^G_{10} = \{3\}$ e $D^H_{10} = \{b\}$; pertanto si possono avere altre compatibilità, ed è possibile estendere la soluzione con la coppia (3,b) e la nuova mappatura {(1,a), (5,c), (3,b)}.
Ovviamente la ricerca può anche continuare sui nodi non scelti precedentemente ma rimandiamo tale discussione al prossimo paragrafo.

Algoritmo

McSplit [3] è un algoritmo ricorsivo basato sulla programmazione a vincoli e un procedimento branch and bound per risolvere il problema del massimo comune

sottografo sviluppato nel 2017 presso l'Università di Glasgow. Esso sfrutta strutture dati efficienti per la replicazione dei dati nei rami dell'esecuzione e migliora il procedimento di backtracking e pruning producendo un miglioramento consistente delle prestazioni rispetto alle precedenti versioni dell'algoritmo.

L'algoritmo McSplit basa il proprio calcolo del bound su un sistema di etichette che estende, dopo la selezione di ciascun nodo, in base a quali nodi siano o meno adiacenti all'ultima selezione. Le etichette consistono in una serie di bit che identificano l'adiacenza a ciascuno dei nodi appartenenti alla soluzione e questo permette una suddivisione in classi molto efficace per far convergere rapidamente il bound a valori prossimi all'ordine della soluzione massima che si potrebbe raggiungere risolvendo quel ramo.

Algoritmo 1: The recursive branch-and-bound McSplit procedure.

```
    McSplit (G, H, M, M_best)
1   if (|M| > |M_best|) then
2     M_best = M
3   end if
4   bound = ComputeBound (G, H, M)
5   if (bound ≤ |M_best|) then
6     return
7   end if
8   bidomain = SelectLabelClass (G, H)
9   v = SelectVertex (bidomain)
10  G = G|v
11  for all u ∈ H do
12    M = M ∪ (v, u)
13    H = H\u
14    newG = FilterDomain (G, v)
15    newH = FilterDomain (H, u)
16    McSplit (newG, newH, M, M_best)
17  end for
18  McSplit (G, H, M, M_best)
```

Esecuzione tipo

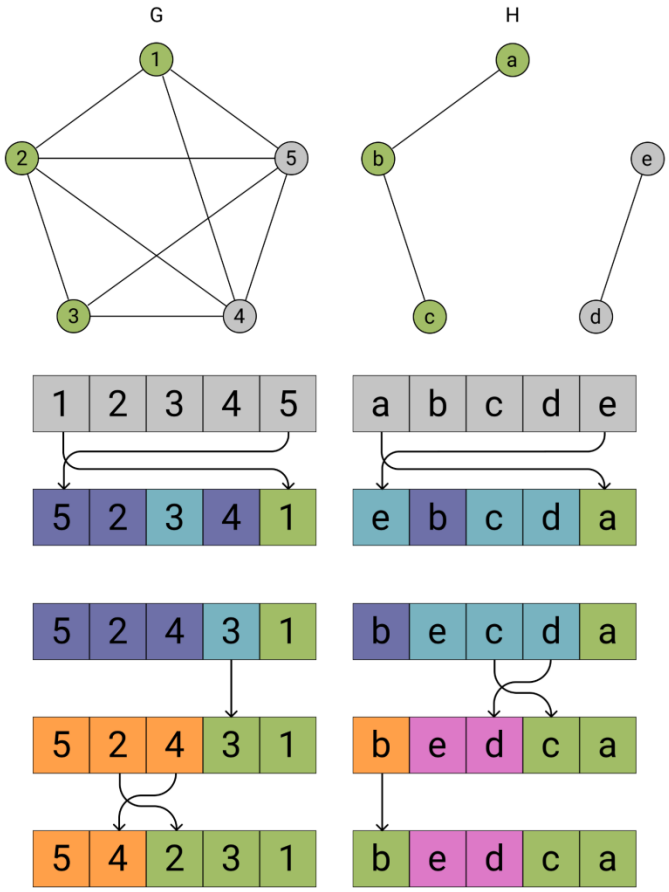

Figura 6: esecuzione tipo McSplit

Per meglio spiegare i precedenti concetti descriveremo il procedimento illustrato nella Figura 6. L'algoritmo risolve la coppia di grafi sfruttando un procedimento di ricerca in profondità. Si sceglie un nodo di G in maniera arbitraria, abbinandolo ad un nodo di H ad esso compatibile (appartenente alla stessa classe, quindi etichettato con lo stesso codice binario), e ricorrendo per selezionare un nuovo nodo sul primo grafo.

Nell'immagine si utilizzano colori diversi per identificare etichette binarie diverse. Nella situazione iniziale (illustrata nei primi due vettori) possiamo vedere che i nodi di entrambi i grafi sono colorati di un colore grigio e questo sta ad indicare che non hanno alcuna etichetta a loro assegnata, sono pertanto tutti compatibili

per essere associati l'uno con l'altro. Dal momento che non abbiamo etichette possiamo anche inferire che il bound della coppia di grafi è pari all'ordine del grafo più piccolo in quanto, nel migliore dei casi, tutti i nodi di quel grafo potranno essere abbinati a quelli dell'altro.

Selezioniamo quindi un nodo del primo grafo che, per coerenza con l'algoritmo McSplit, sarà il nodo con il numero minore, quindi il nodo **1** del grafo G. Come appena detto non esistono ancora etichette in grado di dividere i nodi in base alla loro compatibilità e questo implica che data la selezione attuale tutti i nodi di H sono adatti per essere associati al nodo appena selezionato. Nuovamente per coerenza selezioneremo il nodo **a** del grafo H.

Scelta la prima coppia di nodi abbiamo finalmente informazioni per dividere i nodi restati in "domini" in base alla loro adiacenza o meno. Abbiamo colorato i nodi **1** ed **a** di un colore verde per indicare che sono già parte della soluzione e non verrà più preso in considerazione per le successive ricorsioni. Abbiamo colorato di viola i nodi adiacenti al nodo selezionato sul corrispettivo grafo e di azzurro quelli non adiacenti. Possiamo vedere che abbiamo, rispettivamente, tre nodi adiacenti a **1** su G e un nodo adiacente ad **a** su H, inoltre abbiamo un nodo non adiacente ad **1** su G e tre nodi non adiacenti ad **a** su H. Il bound si può calcolare in questo modo: numero di nodi già selezionati (1) più, per ciascuna tipologia di dominio (adiacenza o non adiacenza), la cardinalità di quello più piccolo tra G ed H (in questo caso 1 per i nodi adiacenti ed 1 per i nodi non adiacenti), per un totale di una soluzione massima di 3 nodi.

Proseguendo come McSplit scegliamo il dominio più piccolo sul grafo di sinistra, che in questo caso è il dominio contenente il nodo **3** (non adiacente ad **1**) e scegliamolo come secondo nodo della soluzione. Su H il dominio corrispondente comprende tre nodi di cui noi sceglieremo quello con numero minore, il nodo **c**.

Data questa nuova coppia di nodi possiamo aggiornare i domini e coloreremo quindi in arancione i nodi adiacenti sia ai nodi **1-a** che ai nodi **3-c**, mentre coloreremo di fucsia i nodi non adiacenti ai nodi **1-a**, né ai nodi **3-c**. In questo caso

non esistono nodi adiacenti ai nodi **1-a** ma non ai nodi **3-c** o viceversa, quindi i rispettivi domini non verranno in alcun modo rappresentati, ma dato un grafo differente essi potrebbero comparire. Data la situazione raggiunta possiamo inoltre vedere che i nodi di colore fucsia compaiono solo nel secondo grafo e questo comporta che da ora in poi non potranno più essere abbinati a nessun nodo di G, siccome non soddisfano le condizioni di adiacenza e non adiacenza ai nodi facenti parte della soluzione. Se ricalcoliamo il bound a questo punto dell'esecuzione troveremo che esso è rimasto pari a tre.
Il passo successivo consiste nel selezionare sui due grafi rispettivamente i nodi **2** e **b** seguendo la stessa procedura precedentemente descritta portandoci nella soluzione illustrata nell'ultima riga. Abbiamo quindi selezionato tre nodi ed i rimanenti non potranno mai entrare a far parte della soluzione in quando appartengono a domini incompatibili tra di loro. A questo punto il bound ha raggiunto lo stesso valore della dimensione della soluzione attuale, tre, e pertanto non possiamo più ricorrere nella speranza di trovare una soluzione migliore. Tuttavia possiamo provare a scartare parte della soluzione tentando di ottenere dei bound maggiori inserendo, al loro posto, nodi differenti nella soluzione. Dal momento che è stata la scelta della coppia **1-a** a portare il bound da cinque a tre, per cercare una soluzione migliore dovremo ricominciare da un diverso abbinamento iniziale.

Come illustrato nell'immagine precedente possiamo vedere che i nodi appartenenti ad uno stesso dominio vengono ordinati in modo da trovarsi adiacenti gli uni agli altri e, da quel momento in poi, qualunque successivo riordinamento avviene solo all'interno del dominio. Questa scelta è stata effettuata al fine di permettere una rappresentazione compatta dei dati ed utilizzare un minore quantitativo di memoria per conservare le informazioni relative a ciascun dominio.

McSplit definisce un insieme minimo di informazioni che devono essere mantenute in memoria per poter descrivere un bidominio:

- posizione di inizio dominio sul primo grafo
- lunghezza del dominio sul primo grafo
- posizione di inizio dominio sul secondo grafo
- lunghezza del dominio sul secondo grafo
- adiacenza o meno all'ultimo nodo selezionato

Questo algoritmo risulta, inoltre, essere parallelizzabile, dal momento che, raggiunto un certo punto in un ramo della ricerca, è possibile creare una copia della situazione della ricerca in quell'istante e delegare il resto del lavoro sul ramo ad un thread helper. Se la replicazione delle informazioni avviene in maniera efficiente possiamo trovarci in situazioni dove il carico di lavoro è distribuito tra diversi thread indipendenti riducendo in proporzione il tempo impiegato.

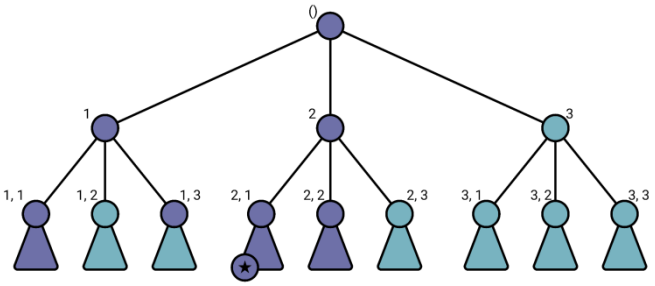

Figura 7: ramificazione dell'esecuzione di McSplit [21]

Questa immagine illustra l'ordine di esecuzione dell'algoritmo McSplit su una coppia arbitraria di grafi. Ogni elemento colorato di scuro è un elemento che deve obbligatoriamente essere visitato per garantire che la soluzione trovata sia veramente quella di dimensione massima. Il simbolo ☆ presente nell'immagine ci indica dove possiamo trovare la soluzione ottima seguendo un'esecuzione single thread. Infine gli elementi colorati in chiaro ci illustrano quelle parti dell'esecuzione che potrebbero essere ignorate se abbiamo già trovato la soluzione

massima, in quanto il loro bound è inferiore alla dimensione della soluzione massima.

Parallelizzare questo procedimento consiste nel selezionare un punto nell'albero precedentemente illustrato e assegnare ad un altro processore la soluzione da quel momento in poi.

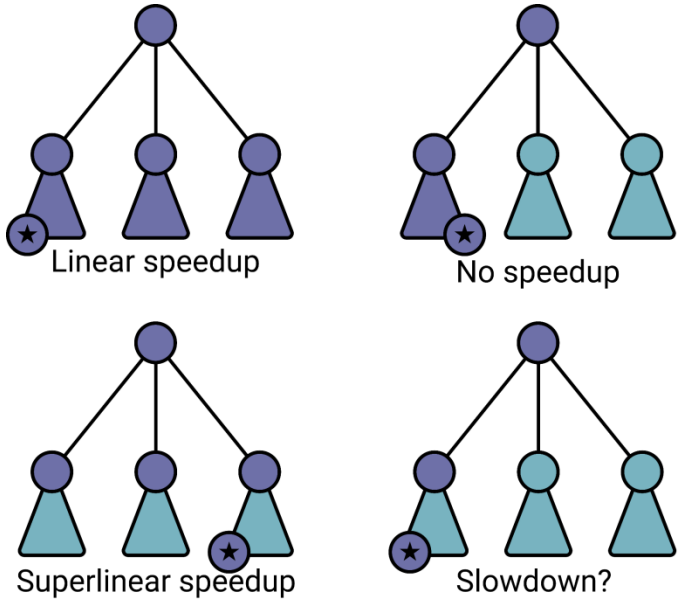

Figura 8: vantaggi della parallelizzazione di McSplit [21]

Come possiamo vedere in questa seconda immagine, però, parallelizzare il processo non garantisce un aumento di prestazioni.

Linear speedup: se il carico di lavoro è equamente distribuito tra i vari thread ci troviamo in una situazione in cui l'incremento di efficienza è linearmente dipendente dal numero di core a nostra disposizione.

No speedup: se i processori addizionali si trovano invece ad affrontare un lavoro inutile, in quanto potrebbe essere scartato trovando la soluzione ottima, e questa verrebbe comunque trovata nelle prime ricorsioni del processo single-thread, ci troviamo di fronte ad una situazione in cui nessuno speed up è ottenuto dalla parallelizzazione.

Superlinear speedup: se la parallelizzazione ci permette di trovare prima del tempo la soluzione ottima e questa innalza il bound ricercato in modo da scartare ancora più soluzioni parziali l'incremento di efficienza è più che lineare.
Slowdown: se la soluzione ottima potesse essere trovata immediatamente dal processo single thread che sarebbe poi in grado di scartare i restanti rami dell'esecuzione, ci potremmo trovare di fronte ad un rallentamento, in quanto parte del tempo verrebbe sprecata per la replicazione di dati da passare ai thread aiutanti che non farebbero altro che scartare immediatamente il ramo.
Questa immagine non considera però uno dei casi più frequenti: rami sbilanciati. Completare l'esplorazione di rami diversi dell'esecuzione potrebbe richiedere tempi anche molto diversi. A causa di questo fenomeno, anche se ci trovassimo in una situazione che dovrebbe garantire un certo miglioramento delle prestazioni, questo potrebbe non verificarsi.
Se l'esplorazione di uno dei rami è significativamente più lunga di tutti gli altri ci troveremo inevitabilmente in una situazione dove uno o più thread hanno già completato le operazioni a loro assegnate, mentre altri sono ancora in esecuzione. In questo caso si parla di un carico di lavoro sbilanciato e, sfortunatamente, data la natura del problema, non esiste un sistema efficace per prevenire questo fenomeno.

Capitolo 5 - Implementazioni CPU

Nel capitolo ci occupiamo delle varie modifiche dell'algoritmo studiate per permettere la soluzione di n grafi contemporaneamente. Il capitolo è diviso in 4 sezioni: nella 5.1 presentiamo una prima implementazione, per un singolo thread, prodotta al fine di dimostrare la possibilità di adattare l'algoritmo preesistente per risolvere il problema esteso ad n grafi. La sezione 5.2 estende questo approccio parallelizzandolo per sfruttare thread multipli. La sezione 5.3 tratta della possibilità di ricercare una soluzione approssimata al problema, riducendo significativamente il tempo di calcolo in cambio della garanzia di trovare la soluzione esatta. Infine discutiamo dei problemi affrontati durante il nostro lavoro nella sezione 5.4.

5.1 Versione single-thread per multipli grafi

Siamo partiti da una versione in grado di funzionare come proof of concept per il resto del nostro lavoro. Per questa implementazione preliminare ci siamo basati sulla versione single thread di McSplit per mantenere la complessità al minimo e dimostrare come fosse possibile un'estensione dell'algoritmo originale ad N grafi.
Questo approccio consiste nella semplice estensione dell'algoritmo e delle strutture dati per gestire un numero indefinito di grafi. Dove l'algoritmo originale crea una partizione tra i nodi di due grafi, raggruppandoli in base alla loro compatibilità, questa implementazione estende il raggruppamento in modo che venga effettuato su tutti i grafi allo stesso tempo, seguendo la stessa logica. Allo stesso modo per creare gli accoppiamenti tra i vari nodi, l'algoritmo non si ferma a selezionare un nodo sui primi due grafi, ma ricorre fino a che tutti gli altri siano stati esaminati. Non cambiano i sistemi secondo cui l'algoritmo sceglie su quale nodo ricorrere, come i domini vengano generati o come venga calcolato il bound.

Algoritmo 2: The multi-graph branch-and-bound procedure.

```
    MultiMCS (G1…n, C, S, level)
1   if (level % G.length) == 0 then
2     FirstGraph (G1…n, C, S, level)
3   else
4     OtherGraph(G1…n, C, S, level)
5   end if

    FirstGraph (G1…n, C, S, level)
6   if (|C| > |S|) then
7     S = C
8   end if
9   bound = ComputeBound(G, H, C)
10  if (bound ≤ |S|) then
11    return
12  end if
13  bidomain = SelectLabelClass (G, H)
14  v = SelectVertex (bidomain)
15  S = S ∪ v
16  G = G\v
17  OtherGraph (G, H, C, S, level + 1)
18  FirstGraph (G, H, C, S, level + G.length)

    OtherGraph(G1…n, C, S, level)
19  for all u ∈ H do
20    H = H\u
21    S = S ∪ u
22    if ((level % G.length) == G.length - 1) then
23      newG1…n = FilterDomain (G1…n, S)
24      FirstGraph (newG1…n, C, S, level + 1)
25    end if
26  end for
27  OtherGraph (G1…n, C, S, level + 1)
```

Come possiamo vedere nell'algoritmo 2 le modifiche all'algoritmo originale sono contenute. Nonostante molte parti del codice originale richiedessero di ricevere in

input due soli grafi, a livello algoritmico l'estensione non richiede un significativo sforzo. Invece di avere un unico corpo della funzione in grado di selezionare un nodo prima sul primo e poi sul secondo grafo, abbiamo separato l'esecuzione in due funzioni principali: una in grado di selezionare un nodo dal primo grafo e una seconda che verrà chiamata ricorsivamente per ogni altro grafo. Ci teniamo a far notare che per due soli grafi si può ricondurre questo algoritmo all'algoritmo 1 descritto nel capitolo 3.1 dove discutevamo l'implementazione dell'algoritmo originale.

5.2 Versione multi-thread per multipli grafi

Nel tentativo di parallelizzare questo processo siamo ripartiti dal codice di Trimble, già parallelo, estendendo in maniera simile al precedente caso strutture dati ed algoritmi.

Per poter risolvere n grafi contemporaneamente una buona parte del codice ha dovuto subire modifiche per accomodare variabili di dimensione dinamica e questo ha portato ad un aumento significativo del tempo di esecuzione, in quanto molte strutture dati devono essere allocate a runtime invece di avere una dimensione fissa. Questo comportamento è stato verificato comparando le performances del nuovo codice con l'originale. La nuova versione sui grafi di test si è dimostrata diverse volte più lenta, soprattutto per i grafi di dimensione più piccola.

Per verificare che la diminuzione delle prestazioni del codice era dovuta agli elementi dinamici, a scopo dimostrativo, abbiamo prodotto una versione del programma in grado di gestire un numero massimo di 10 grafi, numero scelto arbitrariamente, ma sufficientemente grande da rendere la soluzione non calcolabile in un tempo contenuto. Questa versione ha dimostrato di essere in grado di rimanere al passo con la versione originale quando comparata su una sola coppia di grafi.

Volendo superare il problema dell'allocazione fissa sarebbe possibile sviluppare un allocatore apposito in grado di gestire le allocazioni e deallocazioni dell'algoritmo in modo da ridurre al minimo il tempo sprecato da una dimensione variabile delle strutture dati, ma questo lavoro non è stato intrapreso come parte di questa tesi.

Considerazioni

L'ordine con cui i grafi vengono presi dall'algoritmo influenza la rapidità con cui esso è in grado di raggiungere una soluzione. Per via empirica è stato trovato che, se i grafi hanno un diverso numero di nodi, ordinarli secondo un pattern oscillante decrescente (minimo, massimo, secondo minimo, secondo massimo…) porta dei vantaggi significativi in termini di tempo di esecuzione. Il miglioramento prodotto dall'ordinamento è significativo per i primi due grafi, mentre l'ordine dei grafi successivi influisce sempre meno.

5.3 Versione multi-thread per multipli grafi approssimata

A causa delle limitate performances raggiunte dall'approccio originale all'aumentare del numero di grafi si è quindi presa in considerazione la possibilità di ricercare una soluzione approssimata del problema. L'approccio preso in considerazione consiste nella ripetuta applicazione dell'algoritmo McSplit prima su una coppia di grafi e poi, ottenuto il risultato, tra questo ed il successivo grafo. Questo approccio non garantisce che il risultato così ottenuto sia il sottografo di dimensione massima tra gli n grafi presi in considerazione, ma permette una crescita lineare della complessità del problema al crescere del numero di grafi. Per cercare di contenere il peggioramento della soluzione si è modificato l'algoritmo McSplit in modo che non ritornasse una sola soluzione di dimensione massima, ma un numero deciso dall'utente. Risolvendo poi le n soluzioni di dimensione massima

con il successivo grafo si selezionano nuovamente n soluzioni e si prosegue iterativamente con ciascuno dei grafi successivi.
I risultati sperimentali evidenziano uno speed-up significativo all'aumentare del numero di grafi e una riduzione generalmente minima della dimensione della soluzione trovata.

Algoritmo 3: The trivial multi-graph branch-and-bound procedure.

```
   ConnectedMultiMCS (G, H1...n−1)
1  if H1...n−1 = ∅ then
2    return G
3  end if
4  M = Mbest = best = ∅
5  McSplit (G, H1, M, Mbest)
6  sol = ConnectedMultiMCS (Mbest, H2...n−1)
7  if best = ∅ OR |sol| > |best| then
8    best = sol
9  end if
10 return best
```

L'algoritmo 3 illustra, tramite pseudocodice, la procedura precedentemente descritta. Esso fa affidamento sull'algoritmo 1 (McSplit originale) per risolvere le coppie di grafi, ma riutilizza la soluzione ritornata al fine di utilizzarla come grafo G per la successiva ricorsione.
Esistono però delle situazioni in cui questo algoritmo ha un comportamento significativamente peggiore della versione esatta. In particolare abbiamo trovato questa seguente regola che ci delimita un insieme di grafi in cui la soluzione approssimata potrebbe essere di dimensione inferiore.
Dati tre grafi A, B e C e detto mcs(A,B) il massimo comune sottografo tra A e B, la versione approssimata di mcs(A,B,C) otterrà una soluzione inferiore a quella esatta se mcs(A,C) richiede di selezionare dei nodi non presenti in mcs(A,B) e questi nodi trovano una corrispondenza in B, oppure se mcs(B,C) richiede di

selezionare dei nodi non presenti in mcs(A,B) e questi nodi trovano una corrispondenza in A.

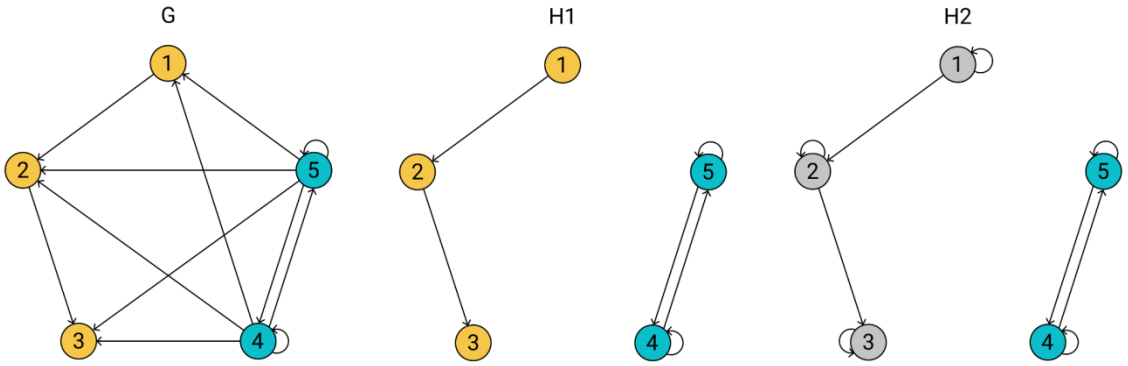

Figura 9: esecuzione esatta vs esecuzione approssimata

Illustriamo adesso la regola appena descritta sfruttando la Figura 9. La soluzione esatta tra il grafo G ed il grafo H1 corrisponde ai nodi 1, 2 e 3 di entrambi i grafi (colorati di giallo), poiché la scelta di questi tre nodi produce il massimo comune sottografo tra di essi. Se calcoliamo il massimo comune sottografo tra questa soluzione e H2 otteniamo però un insieme vuoto, dal momento che i nodi 1, 2 e 3 nel grafo H2 contengono un cappio, incompatibile con i nodi di G ed H1. La soluzione esatta di mcs(G, H1, H2) invece seleziona i nodi 4 e 5 dei tre grafi.

Facendo riferimento all'enunciato precedente la soluzione di mcs(G, H2) e la soluzione di mcs(H1, H2) richiedono di selezionare i nodi 4 e 5 dei rispettivi grafi per trovare il massimo comune sottografo. Inoltre i nodi 4 e 5 selezionati trovano una corrispondenza sul rimanente dei tre grafi, facendo sì che la soluzione approssimata si riveli di dimensione minore rispetto a quella esatta.

Questo esempio dimostra che nel caso approssimato l'ordine in cui vengono risolti i grafi è un elemento importante nel determinare la dimensione della soluzione finale raggiunta.

Considerazioni

La qualità delle soluzioni sembra rimanere invariata sia per grafi con una bassa densità di archi sia per grafi con un'alta densità, portandoci a credere che questo approccio potrebbe essere utile in casi in cui una soluzione trovata rapidamente sia più importante di una soluzione esatta.

Una possibile soluzione per ridurre la possibilità di errore prodotta dal caso descritto precedentemente consiste nel semplice riordinamento dell'ordine di soluzione dei grafi, infatti, costringendo l'algoritmo a selezionare quei nodi indispensabili per raggiungere la soluzione di dimensione massima nei primi passi della soluzione ci permette di evitare interamente la problematica. Sfortunatamente non abbiamo trovato un'euristica adatta a riordinare i grafi a priori e quindi prevenire il presentarsi del problema e, non avendo una stima di quanto la nostra soluzione approssimata sia vicina a quella esatta non abbiamo nemmeno modo di ripetere in automatico l'esecuzione modificando l'ordine dei grafi se la soluzione dovesse essere insoddisfacente. Se uno di questi metodi potesse essere sviluppato, sia tramite un ben studiato insieme di regole sia tramite una rete neurale, l'algoritmo potrebbe raggiungere risultati ancora migliori. Facendo infatti riferimento alla figura 1 selezionare i grafi in un ordine diverso previene il problema indicato. La soluzione approssimata del problema del massimo comune sottografo dove H2 non è selezionato come terzo grafo produce la soluzione di dimensione massima, dal momento che la soluzione di H2 con G o H1, seleziona già i nodi 4 e 5 che risultano critici nel raggiungere la soluzione esatta.

Un metodo che abbiamo impiegato per assicurarci che le n soluzioni migliori di un passaggio selezionate fossero le più distinte possibili è stato una superficiale ricerca di isomorfismo tra di esse. Trovare l'isomorfismo tra due grafi non è un problema semplice [21], pertanto, per contenerne la complessità, abbiamo deciso solamente di controllare se i nodi che la soluzione aveva selezionato fossero distinti su entrambi i grafi per almeno 1 nodo ciascuno, in tal caso la soluzione non

veniva scartata. Questo approccio garantisce di eliminare solamente grafi isomorfi, ma non garantisce che tutti i grafi isomorfi vengano scartati. Se si potesse trovare un differente algoritmo in grado di compiere questo lavoro in maniera più precisa, senza aumentarne significativamente la complessità, potremmo migliorare la precisione della soluzione finale senza rallentare il programma.

Estratto 1: funzione per prevenire la ricerca su soluzioni isomorfe.

```
1   bool controlla_isomorfirmo(vector<vector<VtxPair>>& incumbent, vector<VtxPair>& current, const
    Graph& g0, const Graph& g1, int size) {
2     vector<bool> v_corrente(g0.n, false), w_corrente(g1.n, false);

3     //imposta a vero tutti i nodi appartenenti alla soluzione
4     for (int i = 0; i < size; i++) {
5       v_corrente[current[i].v] = true;
6       w_corrente[current[i].w] = true;
7     }

8     //per ogni altra soluzione
9     for (auto v_vtxpair : incumbent) {
10      vector<bool> v_incombente(g0.n, false), v_complessivo(g0.n, false), w_incombente(g1.n, false),
    w_complessivo(g1.n, false);
11      bool v_int_res = false;
12      bool w_int_res = false;

13      //imposta a vero i vertici selezionati
14      for (int i = 0; i < size; i++) {
15        v_incombente[v_vtxpair[i].v] = true;
16        w_incombente[v_vtxpair[i].w] = true;
17      }

18      //se un nodo è selezionato su solo una delle due soluzioni
19      for (int i = 0; i < g0.n; i++) {
20        v_complessivo[i] = v_corrente[i] != v_incombente[i];
21      }
22      for (int i = 0; i < g1.n; i++) {
23        w_complessivo[i] = w_corrente[i] != w_incombente[i];
```

```
24          }

25          //diventa true se almeno un elemento è true
26          v_int_res = std::accumulate(v_complessivo.begin(), v_complessivo.end(), v_int_res, or_function);
27          w_int_res = std::accumulate(w_complessivo.begin(), w_complessivo.end(), w_int_res, or_function);
28          //deve ritornare true se e solo se sia v_res che w_res sono true
29          if (!(v_int_res && w_int_res)) {
30              return true;
31          }
32      }
33      return false;
34  }
```

5.4 Problemi affrontati

Adattamento ad n grafi di McSplit parallelo

L'algoritmo supporta nativamente il parallelismo ma non è in grado di gestire N grafi.

La porzione parallela del codice usa una mappa per creare i task che devono essere eseguiti. Dal momento che l'ID di ogni task è creato in funzione dei nodi selezionati sui primi due grafi, esso può portare dei problemi quando la sezione di codice ricorsiva, che sceglie i vertici da aggiungere alla soluzione, deve essere chiamata più volte. Per aggirare il problema i task hanno ricevuto un ID univoco non dipendente dal numero di ricorsioni.

Estratto 2: strutture dati per coppie di grafi.

```
1   struct VtxPair {
2       int v;
3       int w;
4       VtxPair(int v, int w): v(v), w(w) {}
5   };
6   struct Bidomain {
7       int l,      r;      // start indices of left and right sets
```

```
8       int left_len, right_len;
9       bool is_adjacent;
10      Bidomain(int l, int r, int left_len, int right_len, bool is_adjacent):
11          l(l),
12          r(r),
13          left_len (left_len),
14          right_len (right_len),
15          is_adjacent (is_adjacent) { };
16  };
```

Il precedente estratto mostra la strutture dati presenti nel codice originale di Trimble. Come abbiamo già detto l'algoritmo è impostato in modo da gestire solamente una coppia di grafi alla volta e questo si riflette nelle strutture dati. Come possiamo vedere alla riga 2, 3, 8 e 9 tutti i dati sono divisi in coppie: una coppia di vertici per ogni abbinamento (v per il grafo di sinistra e w per il grafo di destra), una coppia di indici per l'inizio dei bidomini sui due grafi e una coppia di lunghezze per i rispettivi grafi.

Estratto 3: strutture dati adattate ad n grafi.

```
1   struct VtxPair {
2     vector<int> vv;
3     VtxPair(vector<int> v) : vv(v) {};
4   };
5   struct Bidomain {
6     vector<int> sets;
7     vector<int> len;
8     bool is_adjacent;
9     Bidomain(vector<int> sets, vector<int> len, bool adj) :
10      sets(sets),
11      len(len),
12      is_adjacent(adj) { };
13  };
```

La versione modificata del codice deve essere invece in grado di adattarsi indipendentemente dalla numero di grafi di input. Possiamo quindi vedere le

modifiche, dove la coppia della riga 2 e 3 è stata sostituita da un vettore di interi vv così come le coppie alle righe 8 e 9, sostituite da altrettanti vettori.
Queste modifiche rendono l'algoritmo in grado di salvare un numero di dati adatto a risolvere qualunque numero di grafi, ma allo stesso tempo l'intero corpo del codice è stato modificato al fine di poter gestire queste nuove strutture dati. Dove le assegnazioni erano fatte per una coppia di variabili, ora sono state sostituite da cicli in grado di considerare l'intera estensione dei vettori presenti.

Estratto 4: nuova sezione per processo iterativo.

```
1   nuovo_abbinamento.push_back(w);
2   if (n_nodi_inseriti == argomenti.arg_num - 1) {
3     vector<Bidomain> nuovi_domini = filter_domains(...);
4     soluzione_corrente.push_back(VtxPair(nodi_inseriti));
5     solve(...);
6     soluzione_corrente.pop_back();
7   }
8   else {
9     vector<Bidomain>nuovi_domini = domini;
10    Bidomain &dominio = nuovi_domini [bd_idx];
11    solve_recursive(...);
12  }
13  nuovo_abbinamento.pop_back();
```

In queste righe illustriamo invece, tramite uno pseudocodice, il processo iterativo al cuore delle modifiche apportate. Dal momento che non possiamo sapere a priori quale sia il numero di grafi di cui dobbiamo ricercare la soluzione abbiamo un'esecuzione divergente alla riga 2 di questo estratto. Se abbiamo inserito un numero di nodi pari al numero di grafi da risolvere (valore salvato in argomenti.arg_num), possiamo ricorrere sulla funzione solve originale, aggiungendo alla soluzione corrente gli abbinamenti e procedendo a selezionare un nuovo nodo dal primo grafo. Al contrario, se non abbiamo ancora selezionato un nodo da ciascun grafo, ricorreremo sulla funzione solve_recursive, che

procederà alla selezione di un nodo appartenente al dominio compatibile con il resto della selezione parziale.
Come possiamo vedere anche i domini su cui dobbiamo iterare vengono aggiornati nel primo caso mentre restano uguali a quelli già selezionati nel secondo.
Infine una coppia di push_bask e pop_back ci permettono di inserire e rimuovere nodi dalla soluzione parziale permettendoci di modificare rapidamente la selezione.

Così come nella versione originale dell'algoritmo McSplit, quando creiamo una copia dei dati necessari a proseguire una soluzione parziale, per poi assegnarla ad un thread indipendente, non vi è nessuna garanzia che la soluzione di ciascuna replica impieghi lo stesso tempo. Questo intrinseco sbilanciamento del carico di lavoro, accentuato dalla nuova e più complessa versione del problema, può portare un numero limitato di thread a dover effettuare molti più calcoli di altri. Dovesse verificarsi una tale situazione si verificherebbe un allungamento del tempo di esecuzione dal momento che non staremmo sfruttando a pieno le risorse. Potrebbe essere vantaggioso aggiungere una sezione di logica in grado, quando alcuni thread non hanno più lavoro da svolgere, di creare delle repliche dei dati dei processi attualmente in esecuzione per poi eseguire in parallelo la porzione mancante del processo, ma non abbiamo esplorato questa possibilità nella tesi corrente.

Capitolo 6 - Architettura GPU

Questo capitolo presenta l'architettura CUDA e nello specifico le caratteristiche di ciascuno dei suoi tipi di memoria, illustrando vantaggi e svantaggi che presentano. I concetti di questo capitolo saranno utili in quello successivo per chiarire la scelta di quali memorie utilizzare per ottimizzare il nostro software.

6.1 Introduzione a CUDA

Il nostro lavoro, basandosi su quello sviluppato in una precedente [2], è stato sviluppato per l'architettura CUDA, tipica delle schede NVIDIA. La nostra scelta è ricaduta su questa opzione, invece del framework OpenCL, dal momento che si dimostra mediamente più performante, anche se a scapito della compatibilità con altri sistemi.

Per comprendere meglio le modifiche apportate al codice ci dilungheremo in un'introduzione alla memoria della scheda video e alle differenze tra le diverse sezioni disponibili.

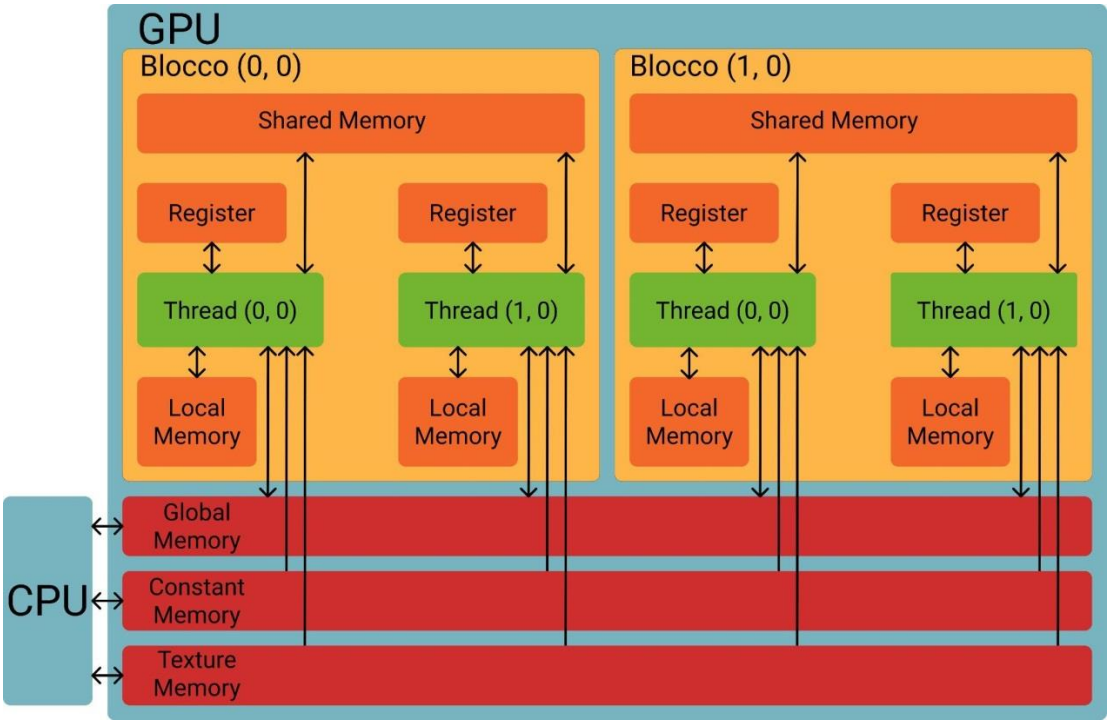

Figura 10: struttura GPU NVIDIA
(http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html)

La figura schematizza la struttura di una GPU NVIDIA: come sono distribuite le sue memorie e la divisione in blocchi dei thread.

Registri

I registri sono la sezione di memoria più rapida a disposizione della scheda video e sono situati fisicamente vicino ai processori. Il principale svantaggio di questa memoria è la sua limitata dimensione che impedisce di salvarvi un grande numero di variabili.

In questa memoria vengono salvate tutte le variabili dichiarate nel corpo di una funzione, compresi i vettori dichiarati in funzioni kernel, ammesso che i loro indici non possano cambiare durante l'esecuzione, dal momento che in tal caso non potrebbero essere ottimizzati al momento della compilazione.

Ogni registro è privato per ciascun thread e tramite essi non è possibile condividere nessuna informazione, nemmeno tra i thread interni allo stesso

blocco. Nel momento in cui un thread termina la sua esecuzione i suoi registri divengono inaccessibili.
Ogni chiamata ad una funzione kernel deve inizializzare i propri registri, dal momento che non è possibile mantenere i registri di una precedente esecuzione.
Questa memoria può essere sia scritta che letta e, dal momento che non è condivisa, non richiede nessun tipo di sincronizzazione.

Local memory

Questa memoria è utilizzata quando i registri subiscono un overflow, ovvero quando nuove informazioni devono essere inserite in un registro, ma tutti quelli disponibili sono già stati occupati.
Questa memoria condivide molte delle caratteristiche dei registri, ma ha un tempo di accesso significativamente peggiore, simile a quello della memoria globale, riducendo sensibilmente le performances di un codice che deve fare frequentemente accesso a queste variabili. Per ridurre l'impatto di accessi ripetuti, su dispositivi più nuovi, a partire da compute capability 2.x, gli accessi a questa memoria producono delle copie in una cache.
Oltre alle variabili che non possono essere salvate in un registro a causa dell'overflow, in questa memoria vengono mantenuti vettori di cui non è possibile predeterminare la posizione al momento della compilazione e strutture dati troppo grandi per essere mantenute in un registro.

Shared memory

Contiene le variabili dichiarate come __shared__. Questa memoria è condivisa tra i thread appartenenti ad un unico blocco e permette ad essi di comunicare tra di loro.
Al contrario della memoria globale, che similmente permette una comunicazione tra thread diversi, questa memoria si distingue per una latenza significativamente ridotta (~100 volte più rapida).

La memoria shared deve essere dichiarata all'interno di una funzione kernel e ha una durata pari al blocco all'interno del quale è condivisa.
Questa memoria può essere sia letta che scritta, ma per garantire che non vi siano conflitti tra i vari thread è necessario sincronizzare gli accessi ad una data variabile oppure fare in modo che diversi thread non accedano mai allo stesso spazio.

Global memory

Le variabili dichiarate come __device__ vengono conservate in questa memoria.
Queste variabili hanno una durata pari a quella dell'applicazione e non sono limitate in accesso ad un solo thread né blocco e possono essere utilizzate per condividere dati tra tutti i thread della scheda video, oltre che direttamente con la CPU.
L'accesso a questa memoria può avvenire sia in lettura che in scrittura, ma la latenza in accesso a questi valori è significativamente maggiore rispetto alla memoria di tipo shared, fino a 100 volte peggiore. Il vantaggio principale della global memory è la sua ampiezza, fino a diversi GB sui modelli più nuovi, di gran lunga superiore alle altre memorie. Per limitare l'impatto dovuto all'elevato tempo di accesso per dispositivi con compute capability 2.0 o maggiore è possibile salvare in una cache i dati a cui si fa accesso, riducendo il tempo per accedere nuovamente a questi o i valori adiacenti, ma i valori contenuti in queste cache sono invalidati nel momento in cui avviene una scrittura e devono nuovamente essere recuperati dalla memoria.
Questa memoria deve essere allocata e, successivamente, deallocata da processi host, ovvero eseguiti sulla CPU.

Constant memory

La memoria costante è una memoria di tipo read-only che, come quella globale, deve essere dichiarata in un ambito globale, quindi in una funzione eseguita dalla

CPU. Nonostante la memoria permetta solo letture da parte della GPU, è possibile inizializzarla tramite funzioni host.

La memoria costante ha un tempo di accesso decisamente ridotto rispetto alla memoria globale nonostante sia disponibile a tutti i thread del dispositivo, grazie al fatto che, non potendo essere scritta è sempre mantenuta in cache, ma per la stessa ragione la quantità disponibile è significativamente ridotta a soli 64KB indipendentemente dal dispositivo.

Esattamente come la memoria globale essa ha una durata pari a quella dell'applicazione.

Texture memory

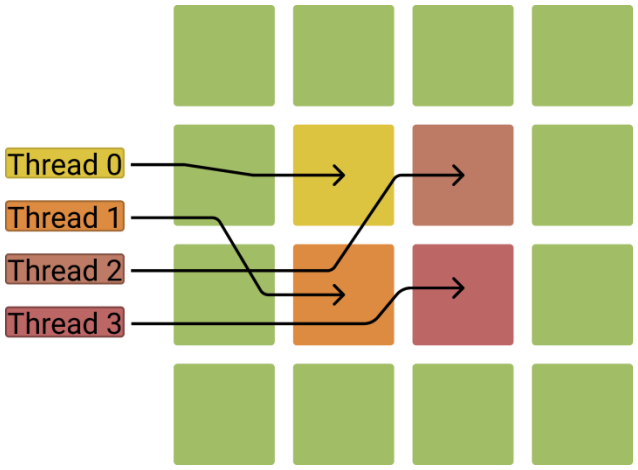

Figura 11: Texture memory (http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html)

La memoria texture ha un comportamento simile a quella costante, in quanto risulta essere read-only. La differenza principale tra i due tipi di memoria è però il sistema di caching, in quanto questa memoria è ottimizzata per l'accesso a strutture dati bidimensionali, come le texture, e quindi, dato un accesso ad una posizione, invece di caricare in cache i valori precedenti e successivi, la cache verrà riempita con i valori di tutte le celle adiacenti.

Capitolo 7 - Implementazioni GPU

Questo capitolo presenta l'implementazione esistente su cui il nostro lavoro si è basato (7.1) e le modifiche da noi apportate per migliorarne le prestazioni (7.2). Una sezione finale di conclusioni elenca i problemi ancora esistenti con la nostra implementazione, la cui soluzione porterebbe un significativo decremento del tempo di esecuzione.

7.1 Implementazioni esistenti

Il lavoro dal quale siamo partiti nel tentativo di migliorare l'algoritmo corrente è quello della tesi "Solving the maximum common subgraph problem on many-cores architectures" [2].
Come discusso nella tesi appena citata l'approccio seguito è stato quello di modificare l'algoritmo McSplit. Per renderlo compatibile con la struttura delle GPU moderne, che hanno uno spazio limitato per permettere le ricorsioni, è stato necessario modificare l'algoritmo in modo da essere in grado di risolvere il problema sfruttando un processo iterativo.
La necessità di riscrivere l'intero codice senza utilizzare la ricorsione ha richiesto di creare delle strutture in grado di salvare tutti i dati necessari invece di potersi affidare allo stack.
Per mantenere tali dati in memoria è stato utilizzato un vettore di strutture dati fatto in modo da minimizzare la memoria occupata. Esattamente come in McSplit esiste una struttura dati utilizzata per contenere tutti i dati relativi a ciascun dominio che è ottimizzata in modo da utilizzare un solo byte per ciascuna informazione. Questa limitazione riduce allo stesso tempo la dimensione massima dei grafi che possono essere risolti e l'autore stesso impone un limite di 64 nodi per ciascun grafo, ragionando che in ogni caso le performances raggiunte non permetterebbero di risolvere grafi di dimensioni maggiori in un tempo contenuto.

La principale struttura dati su cui l'intero algoritmo si basa contiene 8 byte di informazioni:

- indice di inizio del dominio sul grafo di sinistra
- indice di inizio del dominio sul grafo di destra
- lunghezza del dominio sul grafo di sinistra
- lunghezza del dominio sul grafo di destra
- informazione se il dominio è adiacente all'ultimo nodo selezionato
- lunghezza della soluzione corrente
- ultimo nodo selezionato sul secondo grafo
- valore iniziale della lunghezza del grafo di destra (utilizzato come flag per determinare se sono già state provate tutte le combinazioni possibili di nodi sul grafo di destra)

I domini sono poi salvati in un vettore di dati e, dal momento che multipli domini possono essere abbinati ad una soluzione, un secondo vettore di indici tiene in memoria la posizione di inizio e fine del gruppo di domini appartenenti a ciascuna soluzione.

7.2 Miglioramenti

In seguito al lavoro di Mosca abbiamo tentato di migliorare i tempi dell'implementazione di McSplit su GPU. Il lavoro è partito cercando di verificare se una migliore selezione dei nodi potesse portare ad un incremento di performance. L'algoritmo originale funziona raccogliendo una soluzione parziale per ogni thread della GPU che verrà lanciato, per poi passare l'intero blocco di dati alla GPU. Il nostro approccio è stato di aumentare il numero di soluzioni parziali raccolte dall'algoritmo, per poi ordinarle in base al loro bound e quindi procedere a passare alla GPU il blocco di soluzioni più promettenti.

Estratto 5: strutture dati per l'algoritmo su GPU.

```
1   struct Dati {
2               uchar domains[BDS - 2];
3               uchar cur[__gpu_level][2];
4               uchar left[MAX_GRAPH_SIZE];
5               uchar right[MAX_GRAPH_SIZE];
6
7               Dati(...) {...}
8   };
9
10  struct wrapperDati  {
11              uint i;
12              uint number_of_bidomains;
13              uint bound_size;
14  };
```

Per semplificare il procedimento di riordinamento abbiamo creato delle strutture dati in grado di mantenere in memoria tutti i dati che sfruttava l'algoritmo originale, oltre ad un paio di informazioni aggiuntive. In particolare, per il riordinamento, è necessario conoscere la bound_size di un dato dominio e per questo il dato deve essere mantenuto in memoria. Inoltre, mentre nell'algoritmo originale il numero di bidomini era determinato dalla differenza tra l'inizio di una soluzione parziale e la successiva (registrato nella variabile i), una volta che i wrapper dati vengono riordinati non è più possibile risalire al punto di terminazione della soluzione e pertanto un'informazione relativa al numero di bidomini è divenuta necessaria.

Il bound delle soluzioni rimanenti è poi comparato con la dimensione della soluzione riportata dalla GPU e quelle che non possono superare tale bound vengono scartate.

Estratto 6: funzione per rimuovere dati con bound insufficienti.

```
1  void pulisci_argomenti(vector<Dati>& argomenti, vector<wrapperDati>& argomenti_i, uchar
   dim_raggiunta, uint& n_args, int& n_threads) {
2          //rimuovo le soluzioni parziali che ho passato alla GPU
3              n_threads -= min(N_BLOCKS * BLOCK_SIZE, n_threads);
4              if (n_threads == 0) {
5                      n_args = 0;
6                      argomenti.resize(0);
7                      return;
8              }
9              uint bound_inferiore = 0;
10             uint bound_superiore = n_threads-1;
11             uint i;
12             uint pos_inserimento_args_i = 0;
13
14             vector<Dati> args(0);
15             args.reserve(argomenti.size());
16             //cerchiamo il primo elemento con bound > dim_raggiunta
17             while (bound_inferiore != bound_superiore) {
18                     i = (bound_inferiore + bound_superiore) / 2;
19                 if (argomenti_i[i].bound_size <= dim_raggiunta) {
20                             bound_inferiore = i + 1;
21                     }
22                     else {
23                             bound_superiore = i;
24                     }
25             }
26         n_threads -= bound_inferiore;
27             //sono tutti elementi da eliminare
28             if (n_threads == 1 && argomenti_i[bound_inferiore].bound_size <= dim_raggiunta) {
29                     n_threads = 0;
30                     n_args = 0;
31                     argomenti.resize(0);
32                     return;
33             }
34         //non ci sono elementi da eliminare
35             if (bound_inferiore == 0) {
36                     return;
```

```
37              }
38              //spostiamo all'inizio del vettore gli elementi con bound sufficiente
39              for (i = 0; i < n_threads; i++) {
40                      argomenti_i[pos_inserimento_args_i] = argomenti_i[bound_inferiore + i];
41                      argomenti_i[pos_inserimento_args_i++].i = args.size();
42                      for (int j = 0; j < argomenti_i[bound_inferiore + i].number_of_bidomains; j++) {
43                      args.push_back(argomenti[argomenti_i[bound_inferiore + i].i + j]);
44                      }
45              }
46              argomenti = args;
47              n_args = args.size();
48      }
```

L'ordinamento delle soluzioni parziali in funzione del bound è avvenuto per ordine crescente, in modo tale da permettere alcune ottimizzazioni della funzione di pulizia. Siccome il vettore di argomenti e argomenti_i tende a raggiungere dimensioni estremamente elevate il semplice atto di spostare tutte le strutture dati verso l'inizio del vettore costa una quantità significativa di tempo. Avere all'inizio di questo vettore i dati con bound massimo, e quindi quelli eseguiti dalla GPU provoca rallentamenti significativi dal momento che quasi l'intero vettore deve essere spostato alla fine di ciascuna esecuzione. Con questo ordinamento dobbiamo scartare la porzione iniziale del vettore solo nel caso in cui il bound sia aumentato rispetto all'esecuzione precedente. Sebbene questo possa capitare più volte, la dimensione della soluzione tende a raggiungere il valore finale dopo poche esecuzioni del kernel GPU e pertanto lo spostamento degli elementi tende ad avvenire un piccolo numero di volte durante un'esecuzione, risparmiando una quantità significativa di tempo quando devono essere lanciati numerosi kernel.
Questa modifica migliora l'algoritmo originale in due modi: lancia delle esecuzioni su delle soluzioni con un bound superiore mediamente più alto, il che permette di non perdere tempo risolvendo delle soluzioni parziali che non possono raggiungere la dimensione richiesta, ed inoltre raggruppa le soluzioni parziali in gruppi in cui le singole istanze necessitano di un simile quantitativo di tempo per

essere risolte, riducendo il tempo che i thread sprecano in attesa della conclusione del lavoro del kernel. Tale approccio dimostra un decremento significativo del tempo di calcolo, a scapito di un maggiore quantitativo di tempo necessario per raccogliere le soluzioni parziali, pertanto questo approccio si dimostra vincente per grafi sufficientemente grandi dove il tempo di soluzione è molto maggiore di quello di raccolta dati.

In un primo momento, siccome i blocchi di thread di CUDA non sono in grado di accedere alle variabili di tipo shared create dai thread di differenti blocchi, abbiamo optato per lanciare una prima esecuzione contenente un solo blocco di thread per riuscire rapidamente ad avere una dimensione sotto la quale scartare tutte le soluzioni parziali con bound insufficiente. Questa modifica evita che alla GPU vengano assegnate un grande numero di soluzioni parziali incapaci di raggiungere la dimensione necessaria. Complessivamente, tende a migliorare il tempo di esecuzione, dal momento che altri blocchi potrebbero impiegare molto tempo nel calcolo di una soluzione parziale che non potrebbe comunque superare la dimensione della soluzione più grande già trovata da un altro blocco.

Nonostante la precedente modifica, dalle nostre analisi è parso evidente che l'algoritmo spendeva quantità di tempo significative nel tentativo di risolvere delle soluzioni parziali che non uguagliavano o superavano comunque il bound raggiunto da altri thread dello stesso kernel. Abbiamo dunque optato di utilizzare una variabile mantenuta nella memoria globale per registrare la dimensione della soluzione migliore finora trovata, invece di una variabile shared. Questo approccio comporta un rallentamento dovuto all'accesso alla memoria globale della scheda video, quasi 100 volte più lento dell'accesso alla memoria shared quando l'accesso è di tipo non cached [22] ma produce un beneficio di gran lunga superiore grazie alla maggiore collaborazione tra i thread della scheda video, in grado di scartare più rapidamente soluzioni non idonee a superare il nuovo bound. Vogliamo infatti

far notare che, sebbene siano necessarie ripetute letture della memoria condivisa il numero di scritture è significativamente limitato, contenendo l'aumento del tempo speso in accesso alla memoria.
Grazie a questa metodologia viene anche a mancare la necessità di lanciare un singolo blocco di thread come prima esecuzione in quanto la comunicazione tra diversi blocchi è garantita dalla variabile globale e le informazioni sul bound da rispettare sono così condivise tra tutti i thread prima del termine dell'esecuzione del kernel.

Considerazioni

Nonostante i risultati raggiunti, l'algoritmo attuale potrebbe ricevere ancora significativi miglioramenti.
Sul fronte della CPU, utilizzata per produrre le soluzioni parziali da passare alla GPU, l'algoritmo è eseguito da un singolo core. Siccome questa parte dell'esecuzione richiede un quantitativo di tempo non indifferente potremmo ottimizzare il processo dividendo la raccolta dati su più thread. Un ulteriore miglioramento potrebbe venire dall'utilizzo della CPU mentre la GPU sta risolvendo il kernel corrente, che al momento è in attesa della terminazione dell'esecuzione attuale.
Dal lato della GPU abbiamo riscontrato un utilizzo delle risorse estremamente limitato, il che ci porta a pensare che un algoritmo in grado di sfruttare più a fondo il parallelismo sarebbe anche in grado di migliorare significativamente le performances che siamo stati in grado di raggiungere.
È da sottolineare che per i test abbiamo utilizzato un buffer fisso di 50 volte il numero di thread per i grafi più grandi e 10 volte per quelli più piccoli da lanciare in un singolo kernel. Dai test effettuati risulta però evidente che avere una dimensione del buffer fissa non produce sempre i migliori risultati.
Per grafi dove la GPU impiega un tempo contenuto a risolvere le singole istanze avere un buffer troppo grande peggiora solamente le prestazioni dal momento che

raccogliere un grande numero di dati per poi dover ordinarli e gestirne il posizionamento nel vettore di buffer è computazionalmente più costoso del minimo vantaggio ottenuto in termini di tempo di soluzione. In tal caso l'utilizzo del buffer in primo luogo è controproducente, dal momento che richiede una serie di operazioni che vengono effettuate comunque, come l'ordinamento, che risulta poi completamente inutilizzato.

Al contrario, dove la GPU impiega un tempo significativo a risolvere i dati passati durante un singolo kernel, riuscire ad aggregare diverse soluzioni parziali con un simile tempo di esecuzione in un singolo kernel produce dei benefici significativi, soprattutto perché threads a cui è assegnato un bound più piccolo tendono a terminare con enorme anticipo rispetto agli altri e questo porta ad un limitato utilizzo della potenza di calcolo della scheda video.

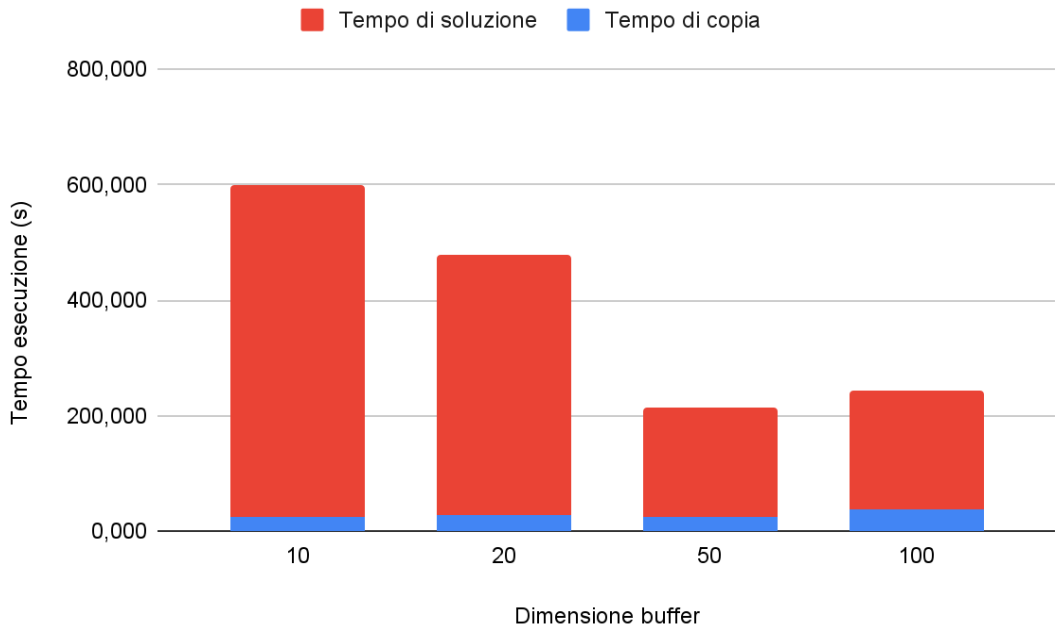

Figura 12: buffer adeguato all'esecuzione

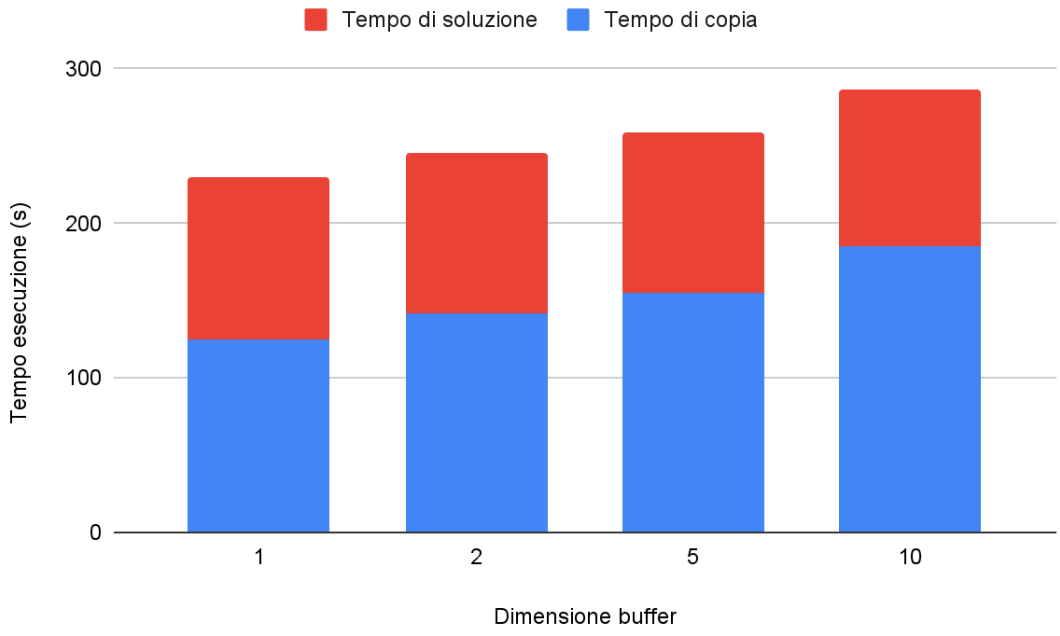

Figura 13: buffer inadeguato all'esecuzione

Nei due grafici di esempio possiamo vedere i tempi di esecuzione per una coppia di grafi la cui soluzione occupa una porzione significativa del grafo con tempi di soluzione molto elevati e una seconda dove vengono lanciati molti più kernel con un tempo di esecuzione estremamente ridotto. Come anticipato, avere un elevato numero di esecuzioni dalla breve durata, porta a dei problemi significativi per il nostro approccio con buffer che spende una significativa quantità di tempo cercando di aggregare soluzioni con un tempo di esecuzione comparabile quando tutte le soluzioni parziali possono essere risolte in meno tempo di quanto ne venga speso cercando di ordinarle.

Per produrre una versione in grado di gestire grafi differenti in modo più efficace sarebbe possibile studiare un sistema in grado o di predire quale buffersize possa essere la più vantaggiosa, oppure creare un sistema che, basandosi sul tempo di copia dati, rispetto al tempo di soluzione, sia in grado di modificare a runtime la dimensione del buffer. Questo non produrrebbe una soluzione in grado di risolvere gli altri problemi descritti, ma aiuterebbe a mitigare gli effetti di una

cattiva selezione della dimensione del buffer. Sfortunatamente non abbiamo avuto tempo per implementare questo approccio nell'ambito di questa tesi.

Capitolo 8 – Risultati

In questo capitolo compariamo i risultati ottenuti con le diverse versioni su una molteplicità di test, prima tra gli algoritmi per CPU su n grafi (8.1) e, a seguire, tra le implementazioni GPU messe a confronto con l'algoritmo McSplit originale (8.2).

8.1 Risultati per n grafi delle versioni per CPU

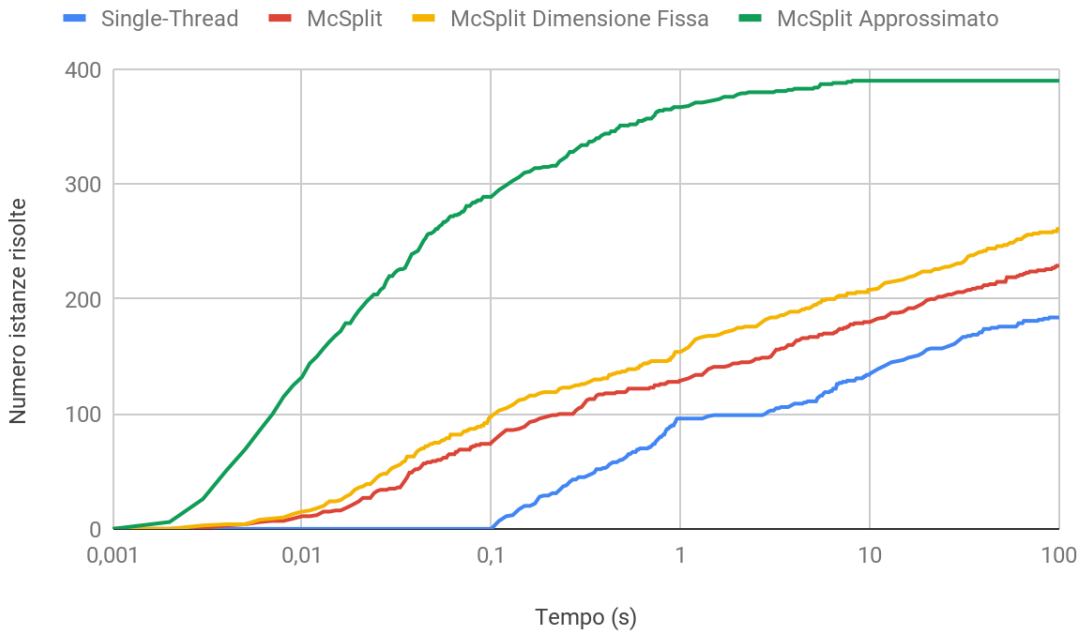

Figura 14: sommario multipli grafi non diretti, tre grafi

Nel precedente grafico possiamo vedere il numero di triplette di grafi risolti entro un certo intervallo di tempo. I grafi selezionati sono non diretti ed equamente distribuiti tra vari metodi di generazione e spaziano tra grafi di 15 a grafi di 30 nodi.

La curva verde, quella che sovrasta tutte le altre, rappresenta le performances della soluzione approssimata precedentemente descritta. Questa implementazione risolve tutte le triplette fornite entro 10 secondi, pertanto la relativa curva risulta piatta nell'ultimo tratto.

Partendo dal basso, la curva azzurra rappresenta la versione single thread dell'algoritmo modificato per gestire multipli grafi, la curva rossa rappresenta la versione di McSplit, già parallela, estesa per gestire N grafi, e la curva gialla rappresenta la precedente versione con molte delle variabili dinamiche sostituite con array statici.

Come già citato in precedenza, la versione approssimata non trova necessariamente la soluzione di dimensione massima e potrebbe, infatti, trovare soluzioni estremamente più piccole dovessero verificarsi le condizioni adatte. Dati questi test abbiamo però evidenziato che in un caso generale il comportamento dell'algoritmo è decisamente migliore delle nostre iniziali aspettative. Infatti su 390 triplette di grafi eseguite, con un timeout di 100s abbiamo riscontrato il seguente comportamento:

- in 1 caso (<1%) la soluzione approssimata era di 2 nodi più piccola di quella esatta.
- in 92 casi (24%) la soluzione approssimata era di 1 nodo più piccola di quella esatta.
- in 26 casi (6%) la soluzione approssimata era più grande di quella degli algoritmi esatti.

Da questi risultati possiamo inferire che generalmente l'algoritmo approssimato trova una soluzione discretamente simile a quella esatta, mentre impiega solo una piccola frazione del tempo per concludere l'esecuzione. I 26 casi in cui la soluzione approssimata supera quella fornita dagli algoritmi esatti erano dovuti ad un timeout che interrompeva la ricerca negli algoritmi più lenti.

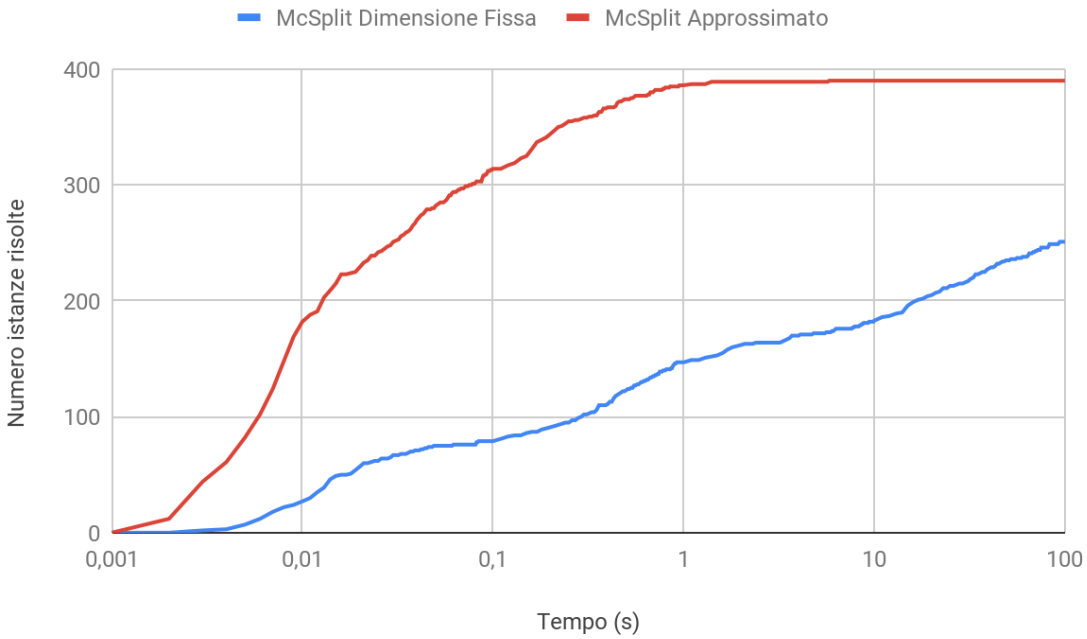

Figura 15: sommario multipli grafi diretti, tre grafi

Abbiamo ripetuto i test su triplette di grafi diretti, selezionate nello stesso modo delle precedenti, per verificare la qualità della soluzione approssimata. Il grafico mostrato nella figura precedente limita i risultati alla versione approssimata e quella con strutture dati di dimensione fissa, già dimostrata essere la più veloce. I risultati in termini di tempo sono molto simili a quelli del caso diretto e, ancora una volta, la versione approssimata è ordini di grandezza più performante.

La qualità della soluzione sembra peggiorare leggermente rispetto al caso non diretto, ma non sostanzialmente. Su 390 esecuzioni:

- in 109 casi (28%) la soluzione esatta era di 1 nodo più ampia di quella approssimata
- in 25 casi (6%) la soluzione esatta, terminata in timeout, era più piccola di quella approssimata

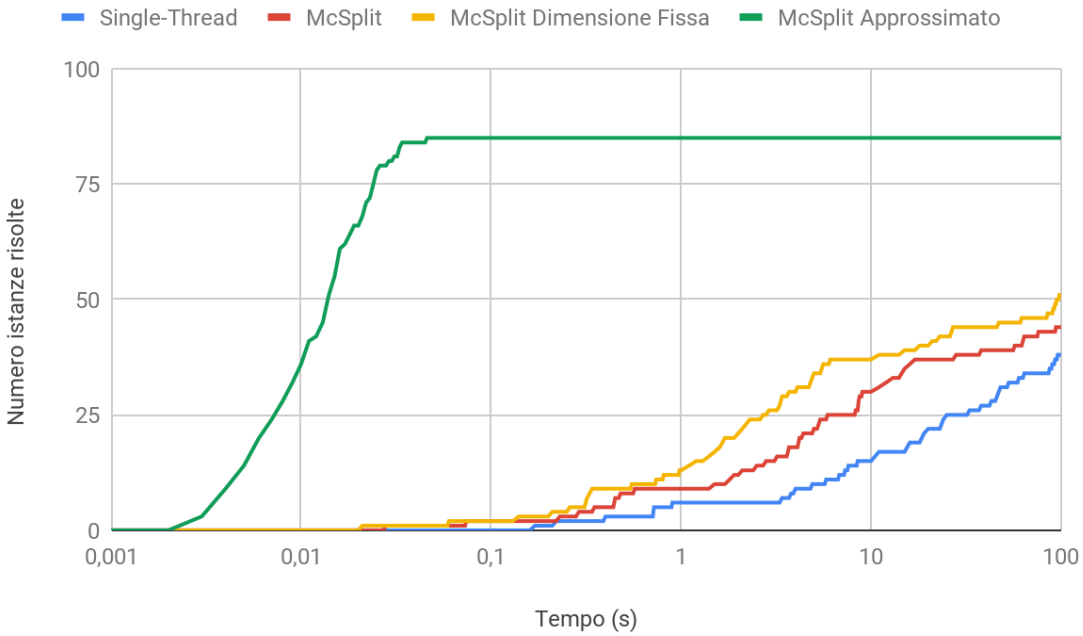

Figura 16: sommario multipli grafi, quattro grafi

In questo secondo grafico mostriamo invece l'andamento dei vari algoritmi quando risolviamo 4 grafi contemporaneamente. Possiamo vedere un andamento molto simile al precedente, ma, in questo caso, la soluzione approssimata termina ancora più rapidamente. Questo comportamento è da imputare al fatto che, a causa della crescente complessità del problema, ci siamo limitati a considerare grafi da 15 a 20 nodi e nonostante ciò la maggior parte delle esecuzioni è comunque terminata in un timeout dopo 100s.

Dal momento che la complessità della soluzione approssimata cresce solo linearmente con il numero di grafi considerati essa si trova in una posizione di enorme vantaggio all'aumentare del numero di grafi da risolvere e questo lo possiamo riscontrare nel crescente divario dei tempi di calcolo.

Le tre versioni esatte seguono un andamento estremamente simile a quello descritto nell'esempio precedente, quindi limiteremo i nostri commenti a far notare come la versione parallela con strutture dati di dimensione fissa mantiene un significativo vantaggio su tutte le altre esatte.

Anche in questo caso abbiamo un comportamento simile in termini di qualità della soluzione approssimata sugli 85 bench lanciati:

- in 18 esecuzioni (21%) la versione esatta trova una soluzione di 1 nodo superiore a quella approssimata.
- in 5 esecuzioni (6%) la versione approssimata trova una soluzione di ordine maggiore a quella esatta a causa dei timeout.

Dal momento che reputavamo troppo stringente un limite di 100 secondi per la soluzione esatta abbiamo eseguito nuovamente la versione con allocazione statica delle strutture dati con un timeout esteso a 1000s, ma anche in questo caso solo 71 delle 85 esecuzioni è terminata entro il tempo limite. Ancora una volta la qualità della versione approssimata è stata confermata, soprattutto in funzione della minima quantità di tempo richiesta dalla soluzione:

- in 21 casi (25%) la soluzione esatta era di 1 nodo superiore a quella approssimata.
- in 1 caso (1%) la soluzione approssimata era di 3 nodi superiore a quella esatta.

8.2 Risultati dell'approccio su architettura many-core

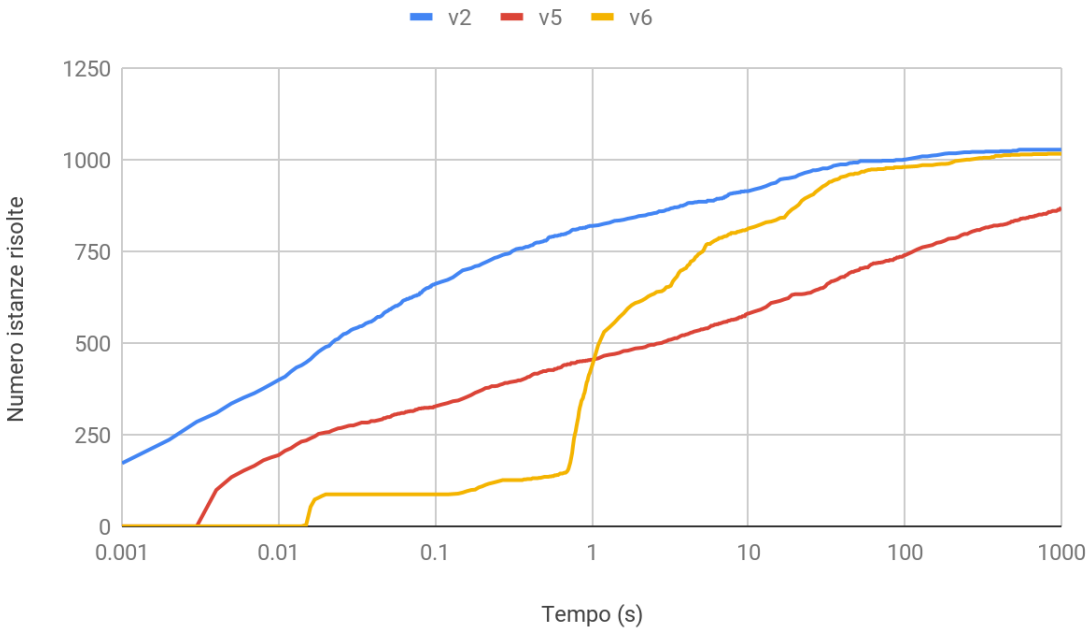

Figura 17: sommario risultati GPU

Nel precedente grafico possiamo vedere tre versioni dell'algoritmo McSplit:

- v2, una delle versioni originali di Trimble, un'implementazione di McSplit scritta in C.
- v5, l'implementazione su GPU di McSplit dell'ingegnere Gabriele Mosca.
- v6, la nuova implementazione che estende la v5.

Come già descritto parzialmente nelle conclusioni relative all'implementazione della versione 6, possiamo vedere che questa implementazione dell'algoritmo si trova in una posizione di significativo svantaggio quando dobbiamo trattare grafi di piccole dimensioni. La causa di questo comportamento può essere ricondotta a due principali fattori: in primo luogo, come già descritto nel lavoro precedente a quello attuale, preparare le risorse per la GPU richiede un discreto quantitativo di tempo e pertanto nelle fasi iniziali dell'esecuzione il divario è significativo. L'ultima versione, inoltre, deve raccogliere un quantitativo di dati assai superiore a cosa faceva la versione 5 prima ancora di poter lanciare il primo kernel GPU.

Il ritardo iniziale passa infatti da 0.004s per la v5 a circa 0.015s per la v6 a causa della presenza del buffer. Analizzando l'andamento delle curve possiamo inoltre notare che la v6 ha un lungo periodo di apparente inattività dopo il primo batch di soluzioni. Questo comportamento può essere ricondotto ai calcoli che seguono il raggiungimento di un nuovo bound e il lancio del successivo kernel. Come già descritto è necessario un processo di pulizia delle soluzioni parziali e un riordinamento di quelle ancora valide prima di poter proseguire, inoltre a questo punto la raccolta di soluzioni parziali richiede un tempo decisamente maggiore dal momento che molte di esse, trovate durante la ricerca, non soddisferanno il bound raggiunto.

Capitolo 9 – Conclusioni

Basandoci sui risultati presentati nel precedente capitolo, queste conclusioni danno delle indicazioni su possibili sviluppi che abbiamo identificato, pur senza aver avuto la possibilità di implementarli nell'ambito di questo lavoro.
Nella sezione 9.1 presentiamo possibili evoluzioni di questo lavoro su multipli grafi pur limitandoci all'utilizzo della sola CPU. Nella sezione 9.2 discutiamo come l'uso di una GPU possa velocizzare la soluzione del problema del MCS su n grafi. La sezione 9.3 illustra ulteriori miglioramenti all'approccio mostrato nel capitolo 7, dedicato alla GPU.

9.1 Soluzione di insieme di grafi su architetture multi-core

Reputiamo che i risultati ottenuti non siano definitivi. L'approccio approssimato non garantisce in alcun modo che la soluzione raggiunta sia di buona qualità, come anche dimostrato mentre parlavamo dell'algoritmo implementato. L'unica ragione per cui riusciamo effettivamente a raggiungere buoni risultati è che non è richiesto di trovare una specifica soluzione di dimensione massima ma una qualunque delle innumerevoli esistenti e quindi raggiungere un buona approssimazione non risulta impossibile. Altre modifiche, come quella già accennata sulla ricerca delle soluzioni isomorfe, potrebbero ancora essere apportate al fine di migliorare ulteriormente la qualità di questa soluzione.

9.2 Soluzione di insieme di grafi su architetture many-core

La possibilità di estendere il precedente caso all'utilizzo di una GPU ci permette di ipotizzare ulteriori modifiche che migliorerebbero le prestazioni della nostra implementazione. Una semplice divisione del lavoro tra i due dispositivi, come abbiamo visto nel capitolo 7, potrebbe essere un approccio preliminare. Estendendo la versione approssimata all'utilizzo di una GPU sarebbe possibile

risolvere coppie di grafi in parallelo, una assegnata alla CPU ed una alla GPU, invece di limitarci ad un approccio pipeline come descritto nella sezione 5.3.

9.3 Soluzione di coppie di grafi su architetture many-core

Le numerose limitazioni imposte dall'algoritmo attuale e, nonostante queste, i discreti risultati raggiunti, ci portano a pensare che con ulteriori ottimizzazioni sarebbe possibile produrre un'implementazione più prestante di quella corrente di McSplit.

Il principale problema da affrontare allo stato attuale è l'impossibilità di far lavorare la CPU e la GPU contemporaneamente: quando una di esse ha l'algoritmo in esecuzione l'altra è inattiva e viceversa.

In primo luogo, dal momento che il procedimento sulla CPU non è stato in alcun modo parallelizzato, potremmo estendere l'algoritmo in modo che, durante l'esecuzione del kernel GPU, i thread non in attesa si occupino della soluzione di altre istanze, esattamente come succede nell'implementazione parallela di McSplit.

Inoltre si potrebbe verificare se l'utilizzo di uno stream di dati tra la CPU e la GPU sia in grado di eliminare completamente il periodo di inattività dei due dispositivi, permettendo alla CPU di fornire costantemente dati alla GPU e, a questa seconda, di non dover attendere che l'intero kernel abbia terminato la propria esecuzione prima di poter ricevere nuovi dati.

Indice delle immagini

Figura 1: Esempi di grafi 7
Figura 2: By Behnam Esfahbod, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3532181 8
Figura 3: approccio dividi e conquista di Hariharan [1] 18
Figura 4: grafo di esempio 20
Figura 5: coppia di grafi di esempio 21
Figura 6: esecuzione tipo McSplit 24
Figura 7: ramificazione dell'esecuzione di McSplit [21] 27
Figura 8: vantaggi della parallelizzazione di McSplit [21] 28
Figura 9: esecuzione esatta vs esecuzione approssimata 35
Figura 10: struttura GPU NVIDIA 43
Figura 11: Texture memory (http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html) 46
Figura 12: buffer adeguato all'esecuzione 54
Figura 13: buffer inadeguato all'esecuzione 55
Figura 14: sommario multipli grafi non diretti, tre grafi 57
Figura 15: sommario multipli grafi diretti, tre grafi 59
Figura 16: sommario multipli grafi, quattro grafi 60
Figura 17: sommario risultati GPU 62

Indice degli algoritmi ed estratti di codice

Algoritmo 1: The recursive branch-and-bound McSplit procedure. 22
Algoritmo 2: The multi-graph branch-and-bound procedure. 30
Algoritmo 3: The trivial multi-graph branch-and-bound procedure. 33
Estratto 1: funzione per prevenire la ricerca su soluzioni isomorfe**.** 36
Estratto 2: strutture dati per coppie di grafi. 37
Estratto 3: strutture dati adattate ad n grafi. 38
Estratto 4: nuova sezione per processo iterativo. 39
Estratto 5: strutture dati per l'algoritmo su GPU. 48
Estratto 6: funzione per rimuovere dati con bound insufficienti. 49

Bibliografia

[1] H. Ramesh, J. Anand, N. Ramaswamy, S. Bhupender, V. Sajith, L. Gregory and S. Ansgar, MultiMCS: a fast algorithm for the maximum common substructure problem on multiple molecules, 2011.

[2] M. Gabriele, Solving the maximum common subgraph problem on many-cores architectures, 2009.

[3] M. Ciaran, P. Patrick and T. James, A Partitioning Algorithm for Maximum Common Subgraph Problems, 2017.

[4] V. Kann, On the approximability of the maximum common subgraph problem, 1992.

[5] D. Edmund, D. H. John and W. Peter, Maximum Common Subgraph Isomorphism Algorithms, 2016.

[6] H. Y. Park, Deriving common malware behavior through graph clustering, 2011.

[7] A. Sirageldin, A. Selamat and R. Ibrahim, Graph–based simulated annealing and supportvector machine in malware detection, 2011.

[8] J. C. Diane and B. H. Lawrence, Substructure Discovery Using Minimum Description Length and Background Knowledge, 1994.

[9] J. M. James, Backtrack search algorithms and the maximal common subgraph problem, 1982.

[10] B. K. Evgeny and H. Kim, Common subgraph isomorphism detection by backtracking search, 2004.

[11] Varkony, T. H., Shiloach, Yossi, Smith and D. H, Computer-Assisted Examination of Chemical Compounds for Structural Similarities, 1979.

[12] Accelrys, Pipeline pilot, 2008.

[13] T. Yoshimasa, S. Yuzuru, S. Hajime and S. Shin-ichi, Recognition of Largest Common Structural Fragment among a Variety of Chemical Structures, 1987.

[14] V. Philippe and V. Benoit, Finding Maximum Common Connected Subgraphs using clique detection or Constraint Satisfaction algorithms, 2008.

[15] N. N. Samba and S. Christine, CP Models for Maximum Common Subgraph Problems, 2011.

[16] T. Petit, J.-C. Régin and C. Bessière, Specific filtering algorithms for over-constrained problems, 2001.

[17] A. Shoukry and M. Aboutabl, Neural network approach for solving the maximal common subgraph problem, 1996.

[18] S. Kristina and W. Fritz, Comparing Structures Using a Hopfield-Style Neural Network, 1999.

[19] ChemAxon, Library MCS, version 0.7, 2008.

[20] D. Andrew and H. Janna, FMCS: a novel algorithm for the multiple MCS problem, 2012.

[21] M. Ciaran, Solving Hard Subraph Problems in Parallel, 2017.

[22] H. Mark, Using Shared Memory in CUDA C/C++, developer.nvidia.com, 2013.