Politecnico di Torino

Politecnico di Torino

Corso di Automation and intelligent cyber-physical systems
A.a. 2024/2025
Sessione di Laurea Dicembre 2025

RRSplit: implementazione multi-thread su CPU e su GPU

Miglioramento ed analisi prestazioni del codice per la risoluzione del massimo comune sottografo

Relatori:
Dr. Stefano Quer
Professore Associato
Dr. Lorenzo Cardone
Professore Esterno

Candidato:
Andrea Cavallo
Matr. s320122

Abstract

Il problema del massimo comune sottografo è una questione ben conosciuta in informatica, e consiste nel ricercare, dati almeno due grafi, quello che è contemporaneamente isomorfo e di massima dimensione. Tale dilemma viene anche sfruttato in altri ambiti, quali la scienza molecolare, per ritrovare somiglianze tra proteine o altre molecole, o, addirittura, nel rilevamento di malware. Tuttavia, esso è NP-hard, e questo ha inizialmente costretto, chiunque lo abbia affrontato, a cercare di approssimare la soluzione ottimale. Negli ultimi anni, però, in parte grazie allo sviluppo tecnologico, ed in parte grazie alla scoperta di nuovi paradigmi, che permettessero di risolvere il MCS in tempi più accettabili, tale problema è stato, in parte, alleviato. Tra i nuovi metodi di risoluzione del MCS, vi è il cosiddetto McSplit, un algoritmo branch and bound che, nel corso degli ultimi anni, è stato preso come riferimento per la risoluzione del problema, e via via modificato in vari modi. Nessuna di queste nuove implementazioni, tuttavia, sembra aver implementato una versione adatta al multi-thread. Nella seguente tesi si prende, come riferimento, una delle ultime "versioni" del McSplit, ovvero RRSplit, e, prendendo gli schemi di parallelismo presenti nel McSplit originale, si cerca di rendere tale lavoro funzionante anche in un contesto multi-thread su CPU. Successivamente, vengono condotte anche delle analisi per capire:

- Quanto l'RRSplit abbia giovato di un'implementazione multithread;
- Quanto le modifiche apportate in RRSplit abbiano velocizzato, in un contesto multithread, la ricerca di un MCS, rispetto al McSplit parallelo originale.

In aggiunta, sono state condotte anche delle analisi sul dataset che è stato utilizzato per la risoluzione di tale problema, e come le principali caratteristiche dello stesso possono aver influenzato, positivamente o negativamente, le prestazioni degli algoritmi.
Come ultima parte di questa tesi, prendendo in considerazione un framework che andasse ad eseguire l'algoritmo di McSplit su GPU, sono state attuate delle modifiche tali da permettere l'implementazione, sullo stesso, degli elementi chiavi caratterizzanti RRSplit. Da lì sono state condotte delle analisi per capire l'efficacia di tale paradigma, confrontando le prestazioni ottenute su RRSplit, implementato su GPU, con quelle che si sono riscontrate con McSplit su GPU.

Sommario

1 Introduzione

I grafi sono strutture dati molto generiche e particolarmente flessibili, rendendoli quindi molto adattabili ed utilizzabili in una vasta gamma di problemi, che sono apparentemente molto eterogenei tra di loro, quali: la chimica [1] [2] e la scienza molecolare [3] [4], andando a trovare un vasto utilizzo addirittura nel campo della computer vision [5] e malware detection [6], fino ad anche a problemi di networking [7]. Con l'evoluzione dei dispositivi atti a risolvere e ricercare il massimo comune sottografo, così come anche delle strategie risolutive, tale problema è stato esteso per grafi che fossero via via più grandi. Ci sono anche alcuni ambiti(biologia molecolare, ad esempio), in cui la ricerca del massimo comune sottografo viene anche svolto tra molteplici grafi [8]. In questa tesi, tuttavia, ci si soffermerà sulla ricerca del MCS tra due grafi, andando ad esaminare un approccio parallelo su CPU e GPU.

1.1 Definizioni

Al fine di rendere facile la compresione dell'argomento in questione, sono state raccolte qui alcune delle definizioni che verranno usate nel corso della trattazione in maniera consistente:

Definizione 1 : Un grafo G è una coppia ordinata degli insiemi V_{G} e E_{G}, rappresentanti gli insiemi dei vertici e degli archi, rispettivamente($G = \{V_{\mathrm{G}}, E_{\mathrm{G}}\}$).

Definizione 2 : Si definisce adiacenza se, in $\mathrm{E_G}$, si ritrova una tra le seguenti coppie di vertici, $< v, w >$ oppure $< w, v >$.

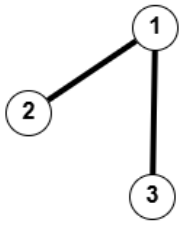

Figura 1: Grafo di esempio

Come è possibile notare dalla figura 1, il nodo 1 è adiacente ai vertici 2 e 3, mentre il nodo 2 non è adiacente al nodo 3 e viceversa.

Definizione 3 : Un grafo può essere **diretto** o **non diretto**: se è non diretto, ciò vuol dire che $\mathrm{E_G}$ è una relazione simmetrica, ovvero, per ogni arco che viene
rappresentato dalla coppia di vertici$< w, v >$, vi sarà anche il suo simmetrico, $< v, w >$.

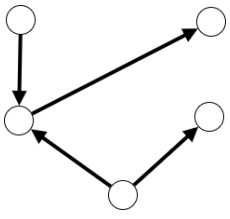

Figura 2: Grafo diretto

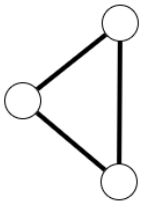

Figura 3: Grafo non diretto

Definizione 4 : Un grafo H viene detto **sottografo** di G se $V_H \subseteq V_G$ e $E_H \subseteq E_G$. Inoltre, se H contiene tutti gli archi dei nodi selezionati in G, H è definito sottografo **indotto**, diversamente viene chiamato sottografo **non indotto**, o **parziale**.

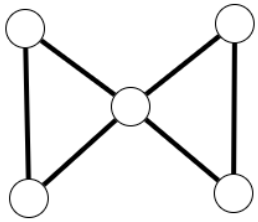

Figura 4: Grafo G

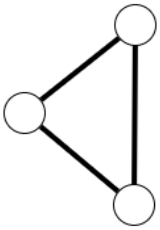

Figura 5: Grafo indotto di G

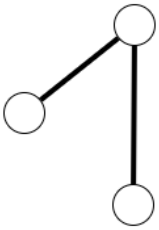

Figura 6: Grafo non indotto di G

Definizione 5 : Due grafi G ed H sono definiti **isomorfi** se esiste una funzione biunivoca $\phi : V_G \rightarrow V_H$ tale per cui, presa una coppia di vertici $< v, w >$ con v e w $\in V_G$, essa appartiene a E_G se e solo se la coppia $< \phi(v), \phi(w) >$ appartiene a E_H.

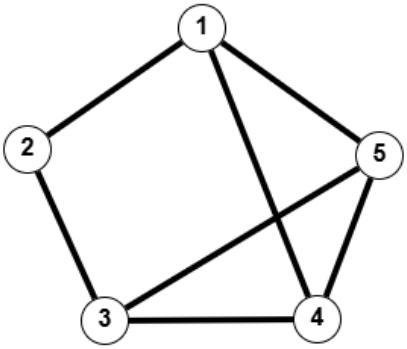

Figura 7: Grafo G

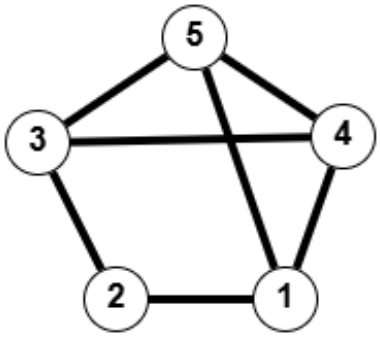

Figura 8: Grafo isomorfo di G

2 Problema del MCS

Il problema del massimo comune sottografo ha riscosso particolare attenzione nel corso del tempo, e numerose definizioni sono state date in base a ciò che si vuole massimizzare. Le principali definizioni tuttavia tentano di massimizzare, rispettivamente, il numero di nodi e il numero di archi che suddetto sottografo deve contenere al suo interno. Nel nostro caso si andrà a considerare la prima delle due definizioni, ovvero: dati due grafi G e H, il massimo comune sottografo che si andrà a ricavare dai essi sarà quello che conterrà il maggior numero di nodi, e sarà, allo stesso tempo, indotto ed isomorfo sia a G che a H. Tale problema è anche estensibile ai casi in cui si vuole esplorare un grafo orientato, rendendo necessario considerare anche la direzione degli archi stessi. È anche possibile l'utilizzo di etichette applicate sui nodi e sugli archi stessi, portando alla generazione di ulteriori vincoli da tenere in conto.

2.1 Applicazioni

I grafi sono, di per sè, delle strutture molto generiche e adattabili, e ciò ha permesso di modellare un grande numero di problemi, risolvibili mediante tecniche che sono state sviluppate per essere applicate sugli stessi. Diversi campi di studi hanno trovato applicazioni per tale problema, tra i quali il campo della chimica [1] [2] e lo studio di strutture molecolari [3] [4]. Ulteriori studi hanno portato alla luce l'applicabilità di tali tecniche per altri campi, come la sicurezza informatica [6], la computer vision [5], l'analisi di codice sorgente [9] e di circuiti digitali [10].
Ci sono anche dei casi, come accade nelle scienze molecolari [8], dove, anzichè andar a cercare il massimo comune sottografo prendendo in esame solamente due grafi, tale ricerca viene estesa e condotta su molteplici grafi contemporaneamente.
Generalmente parlando, la ricerca del MCS può essere applicata ad un qualunque problema, dove non è solamente necessario verificare la presenza o meno di un elemento, ma anche la disposizione dello stesso rispetto agli altri.
Un esempio in cui l'utilizzo di algoritmi per la risoluzione del massimo comune sottografo è stato impiegato riguarda, come già citato in precedenza, la scienza molecolare. Il caso in questione è quello citato da boukhris [3], in cui tale problema viene impiegato per esaminare la somiglianza tra i siti di legame delle proteine.
Tuttavia, il problema del massimo comune sottografo, applicato a questo caso, è stato adattato in modo tale da presentare una certa tolleranza, sfruttando un algoritmo di quasi-clique detection.
In questo modo si possono meglio rappresentare le caratteristiche dei siti di legame delle proteine, che possono essere soggette a mutazioni, evitando quindi che la ricerca di un MCS, applicata in modo rigido, porti a scartare tanti risultati comunque validi.

2.2 Complessità dell'algoritmo

Il motivo che ha portato molteplici informatici a cercare soluzioni sempre più efficienti per questo problema, andando addirittura a convertirlo in un altro suo simile(ad esempio, andando a risolverlo come si risolverebbe il problema della cricca massima, come citato da McCreesh [11]), risiede nella complessità del MCS. Esso infatti è un problema NP-hard, e questo significa che, data la dimensione dell'input(in questo caso, grafi composti da archi e vertici/nodi), il tempo richiesto per la sua risoluzione può essere di tipo esponenziale (da qui il motivo per cui i grafi presi in esame sono generalmente piccoli, con un numero di nodi e di archi piuttosto limitato). Per capire come questo possa essere possibile e le sue implicazioni, verranno date le seguenti definizioni di complessità di un problema, che sono state coniate da una branca della teoria della calcolabilità, ovvero la teoria della complessità computazionale. Tuttavia, viene inizialmente fatta l'assunzione che, come calcolatore, si stia usando una macchina di Turing deterministica, capace quindi di produrre, dato lo stesso input, e le stesse istruzioni da compiere, lo stesso output. La MdT è un calcolatore che va ad utilizzare di un modello matematico, capace di simulare il processo di calcolo umano, e quindi permettendone la sua scomposizione nei suoi passi

ultimi. La macchina presenta una testina utilizzata per la lettura e una per la scrittura, con le quali è in grado di leggere e scrivere su un nastro potenzialmente infinito, partizionato, in maniera discreta, in caselle. Preso un istante di tempo t_1, la macchina si trova in un preciso stato interno s_1, che è dato dall'elaborazione dei dati letti. Le componenti da considerare sono:

- Numero di cella osservata;
- Il suo contenuto;
- L'istruzione da eseguire;

Gli stati che si possono distinguere sono tre, ovvero:

- ***Configurazione iniziale***, ovvero la configurazione che si ha prima che il programma venga eseguito, per $\boldsymbol{t = t_0}$;
- ***Configurazione intermedia***, cioè quando si sta per eseguire una specifica istruzione ad un tempo $\boldsymbol{t = t_i}$;
- ***Configurazione finale***, che si ottiene dopo che il programma ha terminato la sua esecuzione, per $\boldsymbol{t = t_n}$.

Da ciò si può dedurre che andare ad implementare un algoritmo di tal tipo vuol semplicemente dire che esso deve essere capace, volta dopo volta, di compiere una delle seguenti operazioni, ad ogni passo:

- Spostarsi di una casella a destra
- Spostarsi di una casella a sinistra
- Scrivere un simbolo, preso da un insieme di simboli a sua disposizione, su una casella
- Cancellare un simbolo già scritto sulla casella che si sta osservando
- Fermarsi

Andare ad eseguire un'operazione o_1 tra gli istanti di tempo t_1 e t_2 vuol dire passare dallo stato interno s_1 a quello interno s_2, interpretabile anche in questo modo: dato $\{s_1,a_1,o_1,s_2\}$, dove a_1 va ad indicare il simbolo osservato dalla macchina, e data l'operazione o_1, per un detto istante t_1 verrà eseguita tale operazione, che porterà lo stato interno a cambiare da s_1 a s_2.
Le varie tipologie di problema che si possono incontrare sono le seguenti:

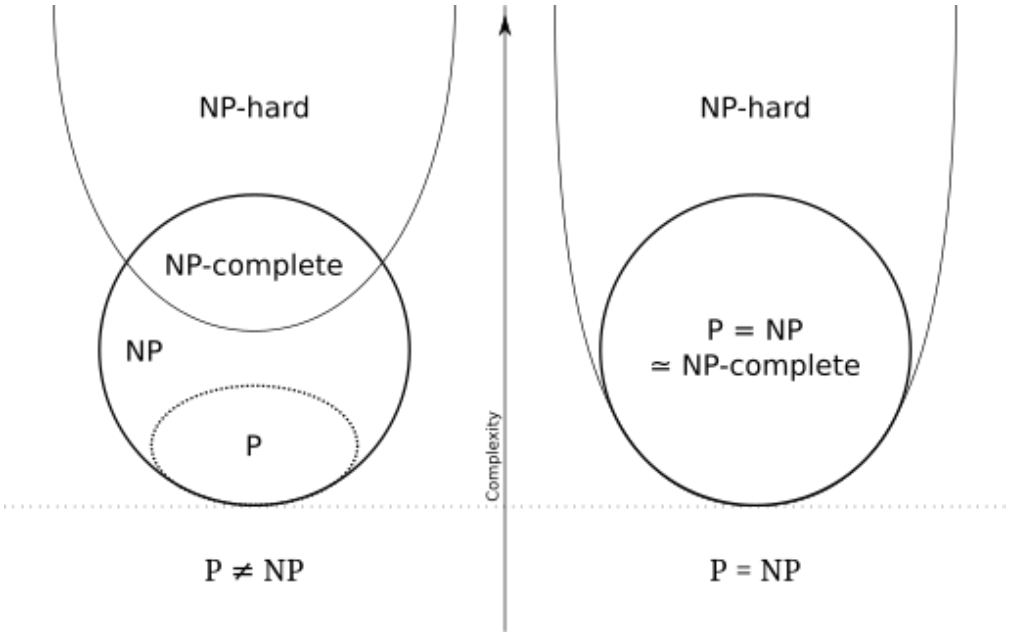

Figura 9: Behnam Esfahbod, CC BY-SA 3.0
https://commons.wikimedia.org/w/index.php?curid=3532181

- **Problema P**: qualunque problema che, dato un input di dimensione n, possa essere risolto con un numero di istruzioni polinomiale rispetto all'input stesso, viene definito problema P, ovvero problema polinomiale;
- **Problema NP**: in questa definizione rientrano tutti i problemi di tipo polinomiale non deterministico, ovvero la macchina utilizzata per risolvere tale problema è non deterministica, implicando, dato un input ed uno stato iniziale, il possibile ottenimento di molteplici output come risultato.
- **Problema NP-completo**: Un problema si dice NP-completo se è un problema al quale è possibile ridurre qualunque altro problema di complessità NP.

  Da qui la conclusione che se si potesse risolvere anche uno solo di questi problemi, NP-completo, in tempo polinomiale, allora tutti i problemi NP che possono essere ridotti al primo potrebbero essere risolti in tempo polinomiale, rientrando quindi nella categoria dei problemi 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.
- **Problema NP-hard**: un problema viene definito tale per via della sua difficoltà di risoluzione, che alcune volte supera anche quella di un problema np-completo, difficile già di per sè. Un problema NP-hard non è però detto che sia anche NP-completo, e questo perchè i problemi NP-completi sono l'intersezione tra i problemi NP-hard e quelli NP, e quindi questo significa anche che non è nemmeno detto che, dato un problema NP-hard, esso sia anche NP.

Per affrontare problemi di tale complessità ci si è spesso ridotti ad approssimare, con una certa precisione, la soluzione di alcuni problemi NP-hard, rendendo quindi il problema di complessità polinomiale.
Purtroppo, non solo non esiste alcun algoritmo in grado di approssimare questo problema con una data precisione, ma è stato anche provato, nel 1992, da Kann [12], che esso appartiene alla famiglia di problemi per cui vale il principio di complessità di approssimazione. 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. Non essendo quindi possibile approssimare la soluzione al problema del MCS con una complessità polinomiale, la maggior parte degli studi, nel corso del tempo, è stata devoluta nel tentativo di trovare un algoritmo che potesse risolvere tale problema in modi sempre più efficienti, andando quindi non più ad approssimare la soluzione ottimale, ma al contrario andando a ricercarla e a trovarla.

2.3 Approcci noti

Essendo un problema studiato da parecchio tempo(se ne parla già dagli anni 70), diversi paradigmi sono stati sviluppati, portando questo problema ad essere risolto, con l'avanzare del tempo, in modi diversi e sempre più efficienti. Tali tipi, che possono essere anche combinati tra di loro, che più sono degni di nota sono i seguenti:

- **Programmazione a Vincoli**

  È il metodo più semplice, ma che può presentare i tempi di risoluzione più lunghi, poichè è molto simile ad un approccio brute force. L'idea alla base, infatti, consiste nell'andare a prendere un insieme di nodi dal primo grafo, e di andare a generare tutte le possibili soluzioni(se si vuole andare ad enumerarle tutte, altrimenti ci si ferma alla prima soluzione corretta ed ottimale) associandolo a tutte le possibili permutazioni di nodi del secondo grafo. Una volta giunti a questo punto, basterà verificare la correttezza delle soluzioni. Per quanto semplice come metodo di risoluzione, è molto suscettibile all'aumento del numero di nodi, e ciò può aumentare sensibilmente il tempo necessario per risolvere il problema.

- **Inferenza**

  La si potrebbe considerare l'evoluzione diretta della programmazione a vincoli. Si basa sull'idea di andare a ridurre il numero possibile di soluzioni ottenute mediante combinazioni già dall'inizio, senza quindi che che poi debbano anche essere verificate. Ciò viene svolto facendo delle assunzioni, o sfruttando delle euristiche, che ci permettono di scartare delle combinazioni che non condurrebbero a soluzioni ottimali. Un esempio semplice di tale tecnica può essere il seguente: 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, esso non potrà mai essere abbinato ad un nodo del secondo grafo adiacente ad uno o più nodi della soluzione. Per quanto risulti essere più efficiente della programmazione a vincoli, si deve comunque tener conto del costo computazionale che vi è dietro l'applicazione stessa delle inferenze. Infatti, 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.

- **Ricerca(Search)**

  L'idea alla base è che, nonostanze le inferenze che si possono fare, spesso la filtrazione che ne viene da esse non è sufficiente per trovare una soluzione, o dimostrare che essa non esiste. Si è quindi costretti a tirare ad indovinare, prendendo una variabile, e forzandola a prendere un valore tra quelli rimasti nel dominio, sperando che ciò possa permettere ulteriori sviluppi, nel tentativo di trovare la soluzione.

  Nel caso in cui, però, ciò non dovesse avvenire, e quindi non si dovesse trovare la soluzione, il processo viene ripetuto, attuando quindi una ricerca ricorsiva.

  Proseguendo con la ricorsione, ci si potrà rendere conto che la variabile presa in esame si è rivelata essere sbagliata, poichè da essa non si genera alcuna soluzione(o non è massima), e si

dovrà quindi fare *backtracking*, oppure essa farà parte della soluzione, e si potrà quindi proseguire ulteriormente.

Ci sono vari tipi di algoritmi, in base al funzionamento di questo paradigma:

- ***Forward checking***
  Avviene quando si attuano una ricerca interlacciata e la propagazione dei vincoli in maniera basilare, sui domini, come già descritto prima;
- ***Mantaining arc consistency***
  Come dice il nome, ad ogni livello di ricerca si cerca di mantenere la consistenza degli archi;
- ***Conventional backtracking***
  In questa definizione rientrano gli algoritmi che non conservano i domini e non rilevano la mancanza di un valore valido per la variabile, fintanto che non si cerca di fare un'assegnazione.

Per garantire delle buone prestazioni, è cruciale che l'algoritmo, incentrato sulla propagazione dei vincoli dei quali fa uso, deve essere capace non solo di capire quando propagare tali vincoli, ma anche quando deve non guardare a tale vincolo.

- ## Euristica

  L'idea alla base è che, quando si deve selezionare la prima variabile dalla quale partire, questa scelta ha un grande peso, poichè, se scelta con criterio e in modo giusto, può snellire i tempi di risoluzione del problema in esame. Se essa viene però scelta male, ciò può comportare dei tempi di risoluzione parecchio allungati. Esempi di questo tipo di programmazione si possono ritrovare in Haralick and Elliott [13], che andavano a considerare, e a scegliere, come primo dominio, quello con i valori più piccoli, e in McSplit, poichè in esso vi è la possibilità di usare, quando si fa la scelta del dominio, una tra due differenti euristiche. Viene definita tale perchè il risultato che ne si ottiene è più dettato dai risultati empirici, è non è quindi detto che la scelta fatta sia la più efficiente a livello scientifico.

- ## Branch and Bound

  Nel 1982 McGregor [14] propose di applicare al problema del massimo comune sottografo un procedimento basato sulla logica branch and bound. Esso si basa su una funzione di bound, che permette 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 metodo è quello che al giorno d'oggi ha ottenuto il maggiore successo ed è stato poi raffinato nei lavori di Krissinel e Henrick [15] e in particolar modo nell'algoritmo McSplit di McCreesh, Prosser e Trimble [16], frutto anche di numerose modifiche negli ultimi anni.

  Questo approccio è poi anche stato esteso nella versione k-down, dove il valore del bound è impostato, all'inizio, esattamente al numero di nodi del grafo più piccolo dei due. Viene poi gradualmente decrementato ad ogni iterazione in cui non riesce a trovare una soluzione che soddisfi quel valore del bound, fino a raggiungere eventualmente la soluzione massima possibile.

  Essendo tale valore impostato in base alla grandezza del grafo più piccolo, questa variante permette di scartare più velocemente quei rami che, seppur di dimensione non massima, possiedono un bound che può superare la dimensione richiesta, ma senza poterla raggiungere veramente.

3 Stato dell'arte

Come già citato in precedenza, McGregor [14] è stato uno dei primi a proporre un algoritmo che si basasse sulla tecnica del branch e bound. Il ramo rappresenta un possibile abbinamento di nodi, che viene esteso fintanto che viene soddisfatta una specifica condizione, calcolata dalla funzione di bound: il valore ritornato da quel ramo deve essere maggiore della massima dimensione raggiunta fino a quel momento. Questo approccio ha subito varie modifiche, volte a rendere più efficiente l'algoritmo: nel 2004, grazie a Krissinel ed Henrick [15], la funzione di ricerca degli abbinamenti è stata migliorata. Con Vismara e Valery [17], invece, si fanno i primi passi verso la programmazione a vincoli.
Presi due grafi G ed H, il modello che si ritrova qui va ad associare una variabile D_v con ogni vertice v di G. Tale variabile, inoltre, contiene tutti i nodi del grafo H, più una variabile aggiuntiva $\perp$, alla quale D_v viene assegnata se v non viene accoppiato a nessun vertice di H.
In caso contrario, D_v viene assegnato al vertice di H con il quale vi è stata la corrispondenza.
Ci sono anche dei vincoli sugli archi, in maniera tale da conservare gli archi e non-archi tra i vari vertici che trovano corrispondenza.
Con Ndiaye e Solnon [18] si fanno ulteriori passi in avanti con la programmazione a vincoli, poichè tutti i vincoli che sono stati introdotti nel lavoro di Vismara e Valery [17] sono stati sostituiti da un vincolo globale di completa differenza. Si è operato in questo modo per massimizzare i nodi di G(primo grafo) assegnati ad H(secondo grafo), a patto che gli abbinamenti siano completamente diversi quando non assegnati alla variabile speciale $\perp$.
Inoltre, si è visto che, in base alle situazioni poste in esame, diversi vincoli e tecniche possono essere impiegati per ovviare ad esse, come l'utilizzo di algoritmo di forward-checking per grafi etichettati.
Uno degli sviluppi recenti, e che ha anche subito una serie di modifiche, è il cosiddetto algoritmo McSplit, lavoro presentato da McCreesh, Trimble e Prosser [16], che riesce ad ottimizzare la replicazione dei dati e presenta un migliore sistema di backtracking.

4 Background

Nel capitolo precedente sono stati evidenziati alcuni lavori che hanno avuto come obiettivo quello di risolvere il problema del massimo comune sottografo, cercando di avere grafi più grandi, di risolverlo nel minor tempo possibile, e di ottenere la soluzione più completa possibile. Tra questi, l'algoritmo di McCreesh, Trimble e Prosser [16] è quello che, nel corso del tempo, da quando è stato pubblicato, ha ricevuto maggiore attenzione e successo, ricevendo svariate modifiche e nuove implementazioni, nel tentativo di rendere la ricerca del MCS più efficiente. Di seguito, si avrà una descrizione dell'algoritmo originale e di quelli successivi, trattando però in maniera più corposa solamente RRSplit [19], argomento di interesse centrale in questa tesi. Tale implementazione, seguendo le linee base del McSplit, ha introdotto altri elementi in modo da ovviare a quelli che possono essere i problemi incontrati nell'algoritmo precedenti, e da migliorarne le prestazioni.

4.1 McSplit

Questo algoritmo segue i principi della programmazione a vincoli, seguendo l'approccio branch and bound. McSplit [16] è stato sviluppato da Ciaran Mccreesh, Patrick Prosser, e James Trimble, nel 2017, all'università di Glasgow. Tale algoritmo, fonte di studio da parte di vari ricercatori, così come anche oggetto di varie modifiche e nuove implementazioni da parte degli stessi, è oggetto di studio, insieme ad RRSplit [19], della tesi in questione.

Algoritmo 1 : McSplit, versione seriale

```
McSplit(G, H, M, M_best, domains)
if |M| > |M_best| then
    M_best ← |M|
end if
bound = calc_bound(domains, M)
if bound ≤ |M| then
    return
end if
bd ← select_bidomain(domains)
v = find_min_value(bd.left)
remove_vtx_from_left_domain(v, bd.left)
for w ∈ bd.right do
    remove_vtx(w, bd.right)
    M.insert(v, w)
    new_domains ← filter_bidomains(domains, G, H)
    McSplit(G, H, M, M_best, new_domains)
    M.pop_back()
end for
McSplit(G, H, M, M_best, domains)
```

Per rendere più facile la comprensione, però, verranno fornite ulteriori definizioni a riguardo:

Definizione 1 : Preso un nodo in un grafo, viene definita **classe di adiacenza** di quel nodo l'insieme dei vertici che sono direttamente connessi ad esso mediante arco.

Definizione 2 : Nell'andare a parlare di questo problema, si ricorrerà all'utilizzo dell'espressione **dominio**. Tale termine verrà impiegato per andare a parlare di un insieme di nodi associati fra loro dalla stessa classe di adiacenza. Andando a considerare il caso del problema con due grafi, si verrà a parlare di bidominio, poichè si avranno due domini(uno per ogni grafo) per ogni classe. Nel caso di n grafi, si avranno n domini per ogni classe(e.g. n = 3, si avrà un tridominio);

Definizione 3 : I nodi di due grafi che sono associati alla medesima classe di adiacenza sono definiti **compatibili**;

Definizione 4 : viene definita **soluzione** un insieme di n-ple di nodi che siano compatibili, appartenenti a n grafi diversi, e che possono essere abbinati per creare un sottografo comune. Nel caso di due grafi, le n-ple si riducono a coppie, appartenenti a 2 grafi differenti.

Per poter meglio entrare nel merito dell'algoritmo, verrà fornito un esempio a scopo esemplificativo:

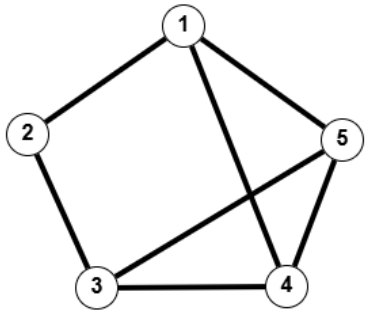

Figura 10: Grafo di esempio

Sia dato il seguente grafo. Nell'andare ad utilizzare la definizione di adiacenza sopra data, è possibile andare a costruire un bidominio per ogni vertice, dove uno dei domini rappresenta i vertici ad esso adiacenti, mentre l'altro tiene conto dei nodi che invece non lo sono. Seguendo questo principio, andando a costruire tale bidominio, ed andando a contrassegnare con 1 i vertici adiacenti e con 0 quelli non adiacenti, per il vertice 1 si avrà tale situazione: essendo adiacente ai vertici 2, 4 e 5, tale dominio di adiacenza, denominato come D_1, conterrà tali nodi, quindi $D_1 = \{2,4,5\}$, mentre D_0, che contiene i nodi che non sono adiacenti al nodo 1, conterrà il nodo 3, portando quindi ad avere $D_0 = \{3\}$. Questo processo, però, può essere espanso, e quindi, tenendo sempre conto del vertice 1, possiamo proseguire a definire ulteriori classi di adiacenza, ma andando ad aggiungere un altro vertice. Se quindi, in questo caso, noi scegliamo il vertice 2, le classi di adiacenza e di non adiacenza saranno aggiornate, ma avremo bisogno di due cifre binarie per andare ad indicare gli ulteriori casi nei quali possiamo incorrere:

- **Classe 11**: i vertici presenti in tale dominio sono adiacenti sia al vertice 1 che al vertice 2. In questo caso avremo che $D_{11} = \{\emptyset\}$;
- **Classe 10**: tali vertici sono adiacenti al nodo 1, ma non al nodo 2. Nel grafo G, in questo caso, tale dominio è $D_{10}= \{4,5\}$;
- **Classe 01**: questa classe va ad indicare tutti i nodi adiacenti al nodo 2, ma non al vertice 1, ed è indicato con D_{01}, che comprende il vertice 3 ($D_{01} = \{3\}$);
- **Classe 00**: include i vertici che non sono adiacenti nè a al vertice 1, nè al vertice 2 ($D_{00} = \{\emptyset\}$).

È quindi chiaro che questo procedimento può continuare finchè sarà possibile aggiungere a tale analisi un altro nodo, ottenendo sempre nuove classi di adiacenza(2^N, con N il numero di nodi presi in esame).

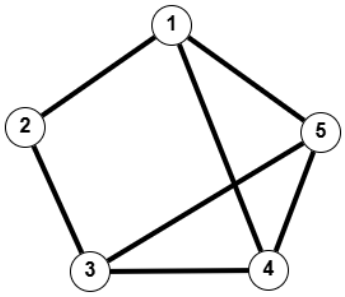

Figura 11: Grafo G

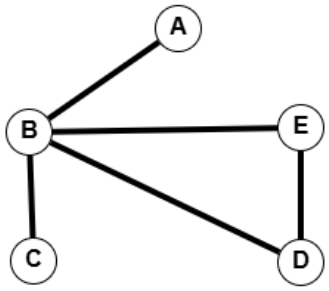

Figura 12: Grafo H

Riprendendo lo stesso grafo di prima, ed accostandolo ad un altro grafo, possiamo eseguire ciò che è stato visto poco prima, ma eseguendo tale analisi in parallelo. Vengono scelti, come vertici di partenza i nodi 1 per il grafo G, ed A per il grafo H. Come primi domini di adiacenza abbiamo:

- Per il grafo G, $D^G{}_1 = \{2,4,5\}$ e $D^G{}_0 = \{3\}$;
- Per il grafo H, $D^H{}_1 = \{B\}$ e $D^H{}_0 = \{C,D,E\}$.

Ci si rende subito conto che i vertici {2,4,5}, del grafo G, e {B}, del grafo H, sono compatibili, poichè sono adiacenti, rispettivamente, ai nodi 1 ed A dei loro rispettivi grafi, e quindi appartengono alla classe di adiacenza D_1.
Proseguendo con la mappatura, si decide di andare a prendere un'altra coppia di vertici, ovvero 3 e C, rispettivamente per i grafi G ed H.
Tale scelta viene fatta perchè il vertice 3 appartiene al dominio di cardinalità minore, ovvero D_0, mentre il nodo C viene selezionato poichè compatibile con il nodo 3 ed avente l'etichetta minore.
Andando ad estendere la mappatura, i risultati che si ottengono sono i seguenti:

- $D^G{}_{11}$:{2,4,5}, $D^G{}_{10}$:{∅}, $D^G{}_{01}$:{∅}, $D^G{}_{00}$:{∅};
- $D^H{}_{11}$:{B}, $D^H{}_{10}$:{∅}, $D^H{}_{01}$:{∅}, $D^H{}_{00}$:{D,E}.

In questo caso è possibile notare come, considerando i domini di adiacenza qui sopra, gli unici vertici che possono essere presi per espandere la mappatura sono quelli appartenenti a D_{11}, ovvero, per G: {2,4,5}, e per H: {B}, e questo perchè essi sono compatibili.
Si procede ulteriormente, prendendo la coppia di vertici {2B}, che andranno a far parte della nuova mappatura, e quindi espandere l'eventuale soluzione, che in quell'iterazione è la seguente : {1A,3C,2B}.
Se si vuole ulteriormente proseguire con le analisi, servirebbe esaminare otto classi di adiacenza, contrassegnate da tre cifre binarie, come 001, 000, etc.
Tale sviluppo si può estendere fintanto che vi sono domini compatibili, e che permettono quindi di estendere la mappatura, così come è anche possibile esplorare i nodi non inizialmente presi in considerazione, magari scegliendo un altro abbinamento di vertici e cercando di espandere di lì la soluzione.
Come già anticipato, McSplit è un algoritmo basato sulla programmazione a vincoli, che sfrutta il branch-and-bound per risolvere il problema del MCS, e tale algoritmo è anche di natura ricorsiva.
L'algoritmo sfrutta un calcolo del bound per poter determinare se, procedendo ulteriormente, ovvero prendendo in esame un determinato nodo, esso possa risultare o meno ottimale. Ciò viene fatto sfruttando un sistema di etichette che va a specificare, preso un nodo, la sua adiacenza rispetto ad altri vertici(un esempio di etichetta può essere 11, il che andrebbe ad indicare che, presa una determinata coppia di vertici, appartenente alla soluzione, essi sarebbero entrambi adiacenti ad un insieme di nodi).
Ciascun nodo, appartenenente alla soluzione presa in considerazione, possiede la propria etichetta.

L'impiego di tale sistema permette una più veloce convergenza del bound a valori vicini a quello della soluzione massima che si otterrebbe nell'esplorazione del ramo, poichè accelera/migliora il processo di pruning e backtracking nel momento in cui si sta procedendo ad esplorare una soluzione non ottimale.
Di seguito una spiegazione, passo per passo, dell'algoritmo, in maniera più dettagliata, riprendendo gli stessi grafi di prima:

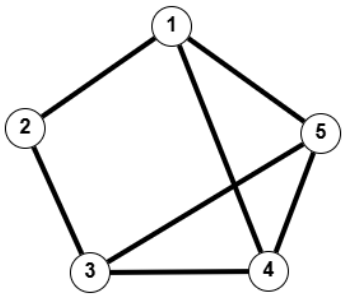

Figura 13: Grafo G

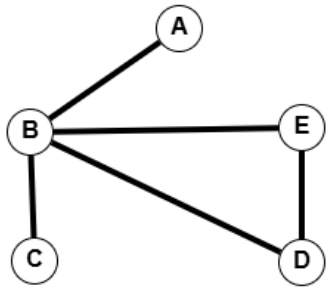

Figura 14: Grafo H

1	2	3	4	5
5	2	3	4	1
2	4	5	3	1
2	4	5	3	1
4	5	2	3	1

A	B	C	D	E
E	B	C	D	A
B	C	D	E	A
B	D	E	C	A
D	E	B	C	A

Mediante ricerca in profondità, ciò che viene fatto è quello di ricercare, partendo da una coppia di nodi, un nuovo nodo nel grafo di sinistra tale che, sfruttando le classi di adiacenza, esso trovi un abbinamento con un nodo del grafo di destra.
Sono stati usati diversi colori per andare ad indicare le diverse etichette binarie incontrate in questo caso.
Come prima coppia di vertici, presi in modo arbitrario, vengono selezionati i nodi 1 ed A per i loro rispettivi grafi.
La scelta della prima coppia di nodi avviene perchè:

- Non essendo i nodi ancora etichettati, sono tutti compatibili;
- L'algoritmo di McSplit, alla luce di ciò, seleziona i nodi con le etichette minori;

Un'ulteriore considerazione che si può fare all'inizio è che il bound dei due grafi coincide con il numero di vertici del grafo più piccolo, e questo perchè, senza aver esplorato i due grafi, è plausibile che il massimo comune sottografo possa coincidere con il grafo con meno nodi.
Una volta selezionata la coppia di vertici, {1A}, si procede con la definizione dei loro domini, ovvero le classi di adiacenza.
I nodi 1 ed A, nel passaggio successivo, sono stati scambiati di posto con l'ultimo elemento ed evidenziati di verde, poichè facenti parte della soluzione parziale, e quindi, per ora, non devono essere più presi in considerazione.
I domini contenenti i vertici adiacenti ai nodi "di partenza" sono stati evidenziati con il colore blu, mentre quelli non adiacenti con il colore rosso. Da ciò emerge che, nel grafo G, il dominio più piccolo è composto dal singolo nodo non adiacente al nodo 1, ovvero 3, mentre nel grafo H ciò coincide con l'unico vertice che, invece, è adiacente ad A.

Arrivati a questo punto, viene fatto il calcolo del bound, che viene così eseguito: si tiene conto del numero di vertici che sono già stati selezionati e che fanno parte della soluzione parziale(1, per ora). Ad esso si sommano le cardinalità dei domini più piccoli, presi dai rispettivi grafi G ed H($D^G{}_0$ e $D^H{}_1$, poichè entrambi hanno un solo elemento).
In questo caso, quindi, il bound ora è uguale a 3, e questo significa che la soluzione massima può avere 3 nodi.
Si procede ulteriormente con l'analisi, e si seleziona il dominio più piccolo del grafo di sinistra, che in questo caso, essendo costituito da un solo elemento, corrisponde con lo scegliere il nodo 3.
Vi sono dei vertici che sono compatibili con il nodo 3, e tra questi viene preso quello con l'etichetta minore, e da qui si ha, come nuova coppia di vertici da aggiungere alla mappatura, la coppia {3,C}.
Vengono quindi a definite nuovamente le classi di adiacenza(che ora sono quattro), e con il colore arancione sono indicati gli elementi appartenenti a D_{11}(elementi adiacenti ad entrambi i vertici della mappatura), mentre con il rosa gli elementi appartenenti a D_{00}(che non sono adiacenti nè all'uno nè all'altro nodo).
È da notare come, in questo specifico caso, si è arrivati ad un punto dove le classi di adiacenza D_{01} e D_{10} non presentano alcun elemento, anche se, con altri grafi presi in considerazione, tale cosa potrebbe non avvenire.
Inoltre, ci si accorge anche che, sebbene il grafo H presenti il dominio D_{00} non vuoto, esso non trova corrispondenza nel grafo G. Questo significa che, nella selezione della prossima coppia di nodi per estendere la mappatura, i nodi D ed E sono automaticamente scartati, lasciando come unica opzione valida il nodo B, poichè compatibile con i nodi del grafo G, ovvero 2,4,5.
Andando nuovamente a calcolare il bound, esso rimane ancora uguale a 3, e questo vuol dire che la soluzione massima sarà composta da 3 nodi.
Selezionando quindi la coppia di vertici 2 e B, ci si rende conto che la soluzione non può più espandersi, poichè si vengono a formare dei domini incompatibili tra loro, portando quindi il bound ad avere lo stesso valore della dimensione della soluzione.
Non ha più senso ricorrere ulteriormente, ma è possibile, facendo backtracking, cercare di scartare una parte della soluzione parziale, nel tentativo di trovare una soluzione più grande. È possibile, per esempio, considerare una coppia di vertici diversa da {1A} come coppia di partenza.
L'algoritmo, essendo ricorsivo, permette, ad ogni chiamata ricorsiva, di aggiornare, o comunque conservare, dei dati che sono cruciali per procedere ulteriormente con la ricerca della soluzione. Tali dati, presenti nella struct Bidomain, sono:

- La posizione iniziale del dominio del grafo di destra;
- La posizione iniziale del dominio del grafo di sinistra;
- La lunghezza del dominio del grafo di sinistra;
- Le lunghezza del dominio del grafo di destra;
- Adiacenza o meno rispetto all'ultimo nodo selezionato.

Vengono utilizzate "solo" queste informazioni perchè, come mostrato nell'immagine di sopra, ciò che accade è che, prendendo i nodi all'interno dei vettori, essi, appartenendo ad un dominio, vengono ordinati in modo tale da avere tutti i vertici della stessa classe di adiacenza adiacenti tra di loro.
Ogni riordinamento successivo ad esso verrà fatto solo con l'intento di riordinare i nodi all'interno delle stesse classi di adiacenza.
Tale scelta, quindi, ha il vantaggio di usare meno memoria ed una rappresentazione più compatta dei vari domini.
Tale algoritmo, inoltre, può supportare anche il multi-threading.
Questo viene fatto perchè McSplit, di per sè, si presta bene al parallelismo, seppur con un caveat: l'algoritmo, arrivato ad un certo punto dell'esplorazione di un ramo, fa una singola copia delle strutture dati, salvando quindi lo stato attuale, in quella ricorsione, e le passa ad un helper thread, così che,

mentre il main thread prosegue nell'esplorazione del ramo, l'helper thread, invece, esplora un differente ramo.

Algoritmo 2 : McSplit, versione parallela

```
McSplit(G, H, M, M_best, domains)
if |M| > |M_best| then
    M_best ← |M|
end if
bound = calc_bound(domains, M)
if bound ≤ |M| then
    return
end if
bd ← select_bidomain(domains)
if M.size() < 5 then
    McSplit_par(G, H, M, M_best, domains)
end if
v = find_min_value(bd.left)
remove_vtx_from_left_domain(v, bd.left)
for w ∈ bd.right do
    remove_vtx(w, bd.right)
    M.insert(v, w)
    new_domains ← filter_bidomains(domains, G, H)
    McSplit(G, H, M, M_best, new_domains)
    M.pop_back()
end for
McSplit(G, H, M, M_best, domains)
```

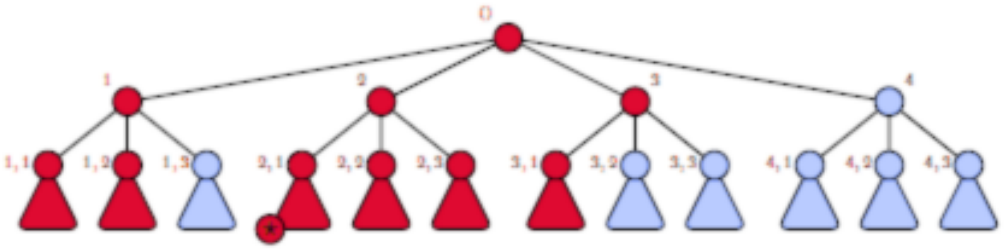

Figura 15: Esempio di ramificazione di McSplit

Prendendo in esame la figura 15, rappresentante una possibile ramificazione ottenuta dall'algoritmo, vengono evidenziati con il colore rosso gli elementi necessari per garantire che la soluzione trovata sia massima. Tra questi, l'elemento che presenta il simbolo $\star$ va ad indicare che, esplorando tale ramo, è possibile trovare la soluzione in maniera ottimale.
Con l'azzurro, invece, vengono indicati tutti gli elementi che non devono essere esplorati se la soluzione massima è già stata trovata, e che quindi è desiderabile evitare per quanto possibile.
Per rendere efficace il parallelismo, ogni volta che l'helper thread sta per processare dei dati futuri nel loop **foreach**, fa una copia della copia di tali dati, e riparte con il loop **foreach**, evitando però le ricorsioni che sono già state fatte. Tale parallelismo è stato limitato ad una profondità di 5, oltre la quale si procede con una ricerca senza andare più a copiare i dati.
Essendo questo algoritmo molto incentrato sul backtracking, ed usando strutture dati locali, l'andare a copiare i dati senza alcun criterio porterebbe ad un significativo rallentamento dell'algoritmo, risultando, quindi, più in un peggioramento delle prestazioni, che in un loro miglioramento.

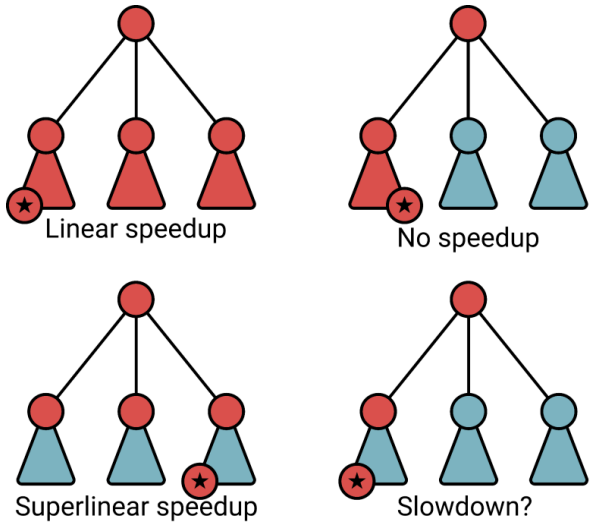

Figura 16: Vantaggi e svantaggi della parallelizzazione

Tuttavia, non è sempre garantito che, utilizzando tale paradigma di programmazione, si ottenga sempre un miglioramento delle prestazioni, ed in genere, i casi che si possono presentare sono i seguenti:

- ***Linear speedup***: Avviene generalmente quando si va incontro a dei rami non eliminabili che devono essere necessariamente esplorati, per poter così garantire la massimalità della soluzione, e il lavoro viene diviso in parti più o meno uguali;
- ***No speedup***: Succede quando i thread esplorano dei rami che non conducono alla soluzione ottimale, e quindi evitabili;
- ***Superlinear speedup***: Accade quando un thread trova la soluzione ottima prima che sia trovata nella run sequenziale, portando vari rami evitabili ad essere scartati nelle run parallele;
- ***Slowdown***: Può succedere, come caso limite, quando, cercando di esplorare i vari rami, ciò che accade è che il main thread, nel trovare la soluzione ottima, va a replicare e passare i dati agli helper thread. Tuttavia, questi non fanno altro che scartare immediatamente il ramo, portando quindi ad uno spreco di tempo.

Un'ulteriore precisazione che va fatta è che generalmente questo tipo di problema porta ad avere dei rami sbilanciati, condizione che può comportare il rallentamento, anche significativo, dell'intero processo. Questo accade perchè, se è vero che alcuni thread hanno già terminato l'esplorazione dei rami ai quali erano stati incaricati, è sufficiente che uno di questi sia ancora in esecuzione, perchè sta ancora esplorando un ramo particolarmente grande, per far "stallare" il tutto.

4.2 Versioni successive

In questa sezione verranno introdotti, seppur in maniera abbastanza concisa, i lavori che hanno portato alla creazione di algoritmi che, seppur poggianti sul codice di McCreesh, presentano delle migliorie.
Da notare: in questi lavori, eccettuato RRSplit, è sempre stato presente un gruppo di persone che ha lavorato dietro a ciascuno di questi algoritmi.

4.2.1 McSplit-RL

Il primo lavoro, volto a modificare l'algoritmo sviluppato da McCresh, preservandone però le sue funzionalità di base, è il cosiddetto McSplit-RL [20].
Tale implementazione, prendendo le basi di McSplit, ed in particolar modo le euristiche che vengono utilizzate, si pone come obbiettivo quello di raggiungere, il più velocemente possibile, le foglie di un albero(ovvero, quei nodi sui quali non è più possible ricorrere ulteriormente), e lo fa sfruttando i principi del reinforcement learning.
L'algoritmo di branch-and-bound viene considerato una sorta di agente, ed ogni volta che esso compie una scelta di branching, riceve una ricompensa commisurata ai benefici ottenuti da tale scelta, con lo scopo di ridurre quanto più possibile lo spazio di ricerca.
Tale scelta, inoltre, avrà un punteggio(*score*) che dipende dalla somma delle ricompense ottenute con le scelte precedenti.
È dunque chiaro che l'agente, nel momento in cui si trova ad un punto di branching, andrà a compiere la scelta associata al maggior punteggio.

4.2.2 McSplit-LL

Uscito nel 2022, McSplit-LL [21] si presenta come il successore di McSplit-RL, poichè prende dei concetti in esso già presenti, come quello di *score*, seppur vengano qui migliorati.
Vengono introdotti due parametri, ovvero l'**LSM**(*Long-Short Memory*) e il **LUM**(*Leaf vertex Union Match*). Il primo parametro mantiene un punteggio per il nodo dal quale parte la ramificazione, usando la ricompensa a breve termine di ogni nodo presente nel primo grafo e la ricompensa a lungo termine di ogni coppia di nodo di entrambi i vertici. In questo modo è possibile attuare un pruning decisamente più efficace rispetto agli algoritmi precedenti.
Il LUM, invece, viene impiegato per accoppiare i nodi *leaf* connessi con quelli attualmente già accoppiati, portando anche qui un miglioramento delle prestazioni.
Uno dei motivi che ha portato alla creazione di tale algoritmo risiede in un "difetto" presente in McSplit-RL: con l'aumentare delle ricorsioni, i punteggi accumulati soffrivano di *bias* a causa di differenti configurazioni attuali rispetto a quelli "storici", ovvero parecchio precedenti, che vengono qui risolti tramite l'**LSM**.

4.2.3 McSplit-DAL

Un altro lavoro da annoverare è quello che ha portato alla creazione di McSplit-DAL [22].
Il fulcro di questo lavoro è che, mentre McSplit-RL e McSplit-LL concentrano i loro sforzi unicamente sulla riduzione dell'upper bound via branching, quest'algoritmo tiene in considerazione anche quanto un grafo venga realmente esemplificato per via di quel branching.
Inoltre, viene anche proposta una strategia di selezione del vertice ibrida, che si basa sul valore ottenuto da una funzione, sì da diversificare la ricerca. Facendo così, è quindi possibile applicare il branching in maniera alternata, basandosi sui valori ritornati da queste funzioni.

4.2.4 RRSplit

Sviluppato nel 2025, tale algoritmo, oggetto di speciali attenzioni in questa tesi, introduce dei concetti volti a scartare dei rami di ricerca che o porterebbero a soluzioni duplicate, e quindi inutili da esplorare, oppure a soluzioni non ottimali, e quindi scartate a priori, prima di un'ulteriore ricorsione ridondante.
Nel corso della trattazione verranno usati delle notazioni che, per dovere di chiarezza, verranno qui enunciati:

B: Si riferisce ad un ramo della ricorsione, che presenterà una configurazione determinata dagli elementi (S, C, D);

S: Fa riferimento al sottografo comune che, nello specifico ramo B, costituisce la soluzione parziale;

C: Con tale lettera vengono denominate le coppie di vertici che fanno parte del *Candidate set*, ovvero possono essere ancora prese in considerazione per poter essere aggiunte alla soluzione parziale S;

D: Denominata anche *Exclusion set*, è la struttura dati incaricata di salvare, ad ogni ramo della ricorsione B, le coppie di vertici che, a differenza di quelle $\in C$, non possono essere incluse in S;

P(C): Fa riferimento a $\{X_\mathrm{i} \times Y_\mathrm{i} | 1 \leq i \leq c\}$, ovvero ai sottoinsiemi del *Candidate set*, al quale è stata applicata tale criterio di riduzione, già presente in McSplit [16]:

Sia dato un ramo (S, C). Una coppia di vertici $< v, w > \in C$ non può formare alcun sottografo comune con S se esiste una coppia di nodi
$< v', w' > \in S$ tale che v e w siano, allo stesso tempo, adiacenti o non adiacenti rispettivamente a v' e w'.

Le riduzioni applicate nel lavoro di RRSplit sono le seguenti:

Vertex-Equivalence-based Reduction: L'idea alla base di tale riduzione si basa sul seguente caso: presi due massimi comuni sottografi$< g, h, \phi >$ e $< g', h', \phi' >$ dei due grafi G ed H, vengono definiti *cs-isomorfi* se g è isomorfo a g'(o, in maniera equivalente, h è isomorfo ad h').

Poichè tali sottografi comuni, definiti cs-isomorfi, condividono le stesse informazioni strutturali, è chiaro che andare ad esplorarli tutti porta solo ad uno spreco di tempo. Quindi, se è stato trovato, esplorando un ramo, uno di questi sottografi $< g, h, \phi >$, esso potrà essere ignorato se esiste $< g', h', \phi' >$ che soddisfa tali condizioni:

Condizione 1 $< g', h', \phi' >$ è cs-isomorfo a $< g, h, \phi >$;

Condizione 2 $< g', h', \phi' >$ è stato già trovato prima;

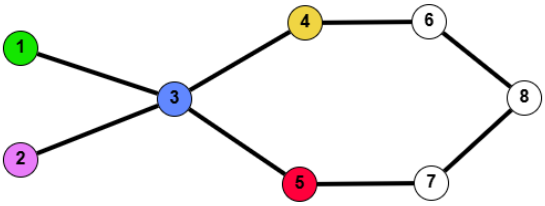

Figura 17: Sottografo G

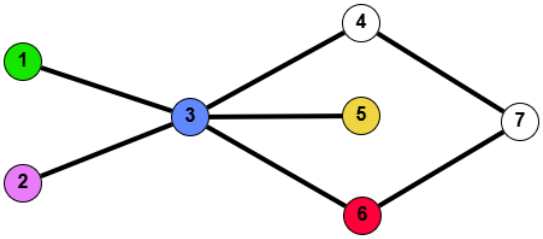

Figura 18: Sottografo H

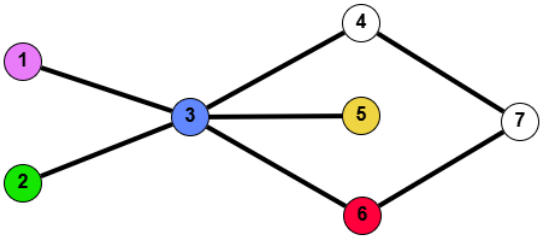

Figura 19: Sottografo H_1

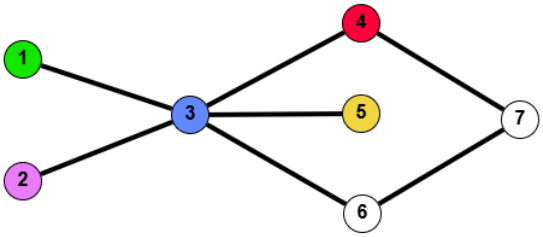

Figura 20: Sottografo H_2

Sia dato il seguente esempio: siano presi i grafi in figura 17 18 19 e 20. Alcuni dei loro vertici sono evidenziati come facenti parte della soluzione, e sono distinti con colori diversi. È possibile notare che alcune di queste configurazioni, riguardante il grafo H, presentano dei vertici che sono strutturalmente equivalenti. Esplorare ciascuna di queste soluzioni è quindi ridondante e porta solo a consumo inutile di tempo.

Presa una coppia di vertici $< v, w >$, appartenenti al grafo G, essi vengono definiti strutturalmente equivalenti, e vengono indicati come $v \sim w$, se $\forall v' \in V_{\mathrm{G}}, (v, v') \in E_{\mathrm{G}} \leftrightarrow (w, v') \in E_{\mathrm{G}}$.

Vengano esaminati, per esempio, le figure 18 e 19, rappresentanti, rispettivamente, $< g, h, \phi >$ e $< g, h_1, \phi_1 >$, ovvero due delle possibili configurazioni isomorfe che si possono ottenere dal grafo G e dal grafo H qui considerati. È possibile notare che, esaminando i vertici $\{1, 2\}$ dei sottografi in figura 18 e 19, colorati in maniera alterna con i colori verde e viola, essi sono strutturalmente equivalenti. Nell'andare quindi ad esplorare lo spazio delle soluzioni, è sufficiente considerare solo una tra le due mappature qui presenti.

Lo stesso vale per i sottografi in figura 19 e 20 poichè, anche in questo caso, presi i vertici $\{4, 5, 6\}$, essi sono strutturalmente equivalenti, rendendo quindi le mappature $< g, h_1, \phi_1 >$ e $< g, h_2, \phi_2 >$ cs-isomorfe.

Basandoci dunque su questa definizione di vertici strutturalmente equivalenti, è quindi possibile andare a partizionare il grafo G in classi di equivalenza, dove essa, per il vertice $v \in V_{\mathrm{G}}$, viene definita come tale: $\Psi(v) := \{v' \in V_{\mathrm{G}} | v' \sim v\}$.

Basandoci quindi sulla *Vertex Equivalence*, è possibile andare a riconoscere diversi sottografi comuni che sono cs-isomorfi a quello dato da g, h, andando seplicemente a sostituire un vertice $\in V_{\mathrm{G}}$ con uno che è strutturalmente equivalente.

Considerando $v \in V_{\mathrm{G}}$ e un vertice che ne è strutturalmente equivalente $v_{\mathrm{eq}} \in \Psi(v)$, è possibile ottenere un sottografo comune cs-isomorfo in due casi:

- Se $v_{\text{eq}} \in V_{\text{G}}$, diventa possibile scambiare i vertici mappati di v e v_{eq}, sostituendo quindi $< v, \phi(v) >$ e $< v_{\text{eq}}, \phi(v_{\text{eq}}) >$ con $< v, \phi(v_{\text{eq}}) >$ e $< v_{\text{eq}}, \phi(v) >$;
- Altrimenti, si rimpiazza $< v, \phi(v) >$ con $< v_{\text{eq}}, \phi(v) >$, ovvero si sostituisce v con v_{eq};

Da qui, si ha la formulazione del seguente lemma:

Lemma 1 *Sia $S =< g, h, \phi >$ un sottografo comune dei grafi G ed H, con $v \in V_G$ e $v' \in \Psi(v)$. Si verificherà uno dei seguenti casi:*

Caso 1: $v' \in V_G . S' = S \setminus \{< v, phi(v) >, < v', \phi(v') >\} \cup \{< v, \phi(v') >, < v', \phi(v) >\}$ *è un sottografo comune cs-isomorfo a S;*

Caso 2: $u' \notin V_G . S' = S \setminus \{< v, \phi(v) >\} \cup \{< v', \phi(v) >\}$ *è un sottografo comune cs-isomorfo a S.*

Per verificare che un sottografo comune cs-isomorfo è già stato trovato in precedenza, viene introdotta una nuova struttura dati, che viene definita exclusion set D. Essa viene aggiornata ad ogni ricorsione, man mano che si procede nei vari rami di ricerca, così che ogni ramo viene così indicato con (S, C, D). Ad essere specifici, in D sono presenti tutte le coppie di vertici che sono già state selezionate come facenti parte della soluzione parziale, e che quindi non devono essere inclusi nei sottografi all'interno del ramo in considerazione.

Inizializzato vuoto, (dati (S, C, D), all'inizio esso sarà $(\emptyset, V_{\text{G}} \times V_{\text{H}}, \emptyset)$) ad ogni ricorsione, preso il sotto-ramo $B^{\text{i}} = (S^{\text{i}}, C^{\text{i}}, D^{\text{i}})$ che si forma con l'inclusione di $< v, w^{\text{i}} >$ in S, per il primo gruppo si avrà $D^{\text{i}} = D \cup \{< v, w^1 >, < v, w^2 >, ... < v, w^{\text{i-1}} >\}$.

Per il secondo gruppo, invece, dove si forma il sotto-ramo (S', C', D'), si avrà $D' = D$.

Sia preso in considerazione un ramo (S, C, D) ed una coppia di vertici $< v', w' > \in D$. Esiste un antenato di tale ramo, indicato come $(S_{\text{anc}}, C_{\text{anc}}, D_{\text{anc}})$, dove v' è stato preso come vertice di branching. Questo porta quindi $< v', w' >$ a non essere ancora presente in D_{anc} fintanto che non si crea B'_{anc}, portando così ad avere $D'_{\text{anc}} = D_{\text{anc}} \cup \{v', w'\}$.

In questo modo, ciascun sottografo comune all'interno di B'_{anc}, che contengono $< v', w' >$, sono già stati prima di giungere al ramo (S, C, D).

Da qui è ora possibile sviluppare le riduzioni: si consideri il processo di ramificazione di un ramo definito come $(S =< g, h, \phi >, C, D)$ con $X \times Y \in P(C)$, e con v e X presi come vertice e sottoinsieme di branching.

Riduzione al primo gruppo Si considera un sotto-ramo formato al primo gruppo includendo $< v, w >$: $w \in Y$, con ogni sottografo S_{sub} che dovrà includere tale coppia di vertici.

Tuttavia, se esiste una coppia di vertici $< v_{\text{eq}}, w > \in D$ tale che $v_{\text{eq}} \in \Psi(v)$, il ramo può essere scartato, poichè le condizioni 1 e 2 sono state soddisfatte.

Riduzione al secondo gruppo In questo caso, il vertice v viene escluso dalla selezione, portando quindi ogni $S_{\text{sub}} = (g_{\text{sub}}, h_{\text{sub}}, \phi_{\text{sub}})$ che si trova nel sotto-ramo ad escludere tale vertice.

Se in S_{sub} si trova un vertice $v_{\text{eq}} \in C \setminus v$, ma è comunque strutturalmente equivalente a v le condizioni 1 e 2 sono soddisfatte, e quindi si può scartare tale sotto-ramo.

Maximality-based Reduction: Come il nome già lascia intendere, questo tipo di riduzione si occupa di rimuovere tutti i rami che, se esplorati, porterebbero ad una soluzione non massima. L'idea sulla quale poggia questa riduzione è la seguente: andando ad esplorare un determinato ramo di una ricorsione, esisterà, per quello stesso ramo, una coppia di nodi $< v, w >$ che deve essere contenuta affinchè la massimalità della soluzione sia garantita.

Da qui, è possibile quindi proseguire con l'esplorazione del sotto-ramo $S \cup \{< v, w >\}, C \setminus v \setminus w$, che va ad escludere tale coppia di vertici v, w dal candidate set, aggiungendola quindi alla soluzione

parziale, per garantire la massimalità della soluzione. Andare invece ad esplorare tutti i rami che non includono tale coppia di vertici nè nella soluzione parziale, nè nel candidate set, significa automaticamente esplorare delle soluzioni non massime per quel ramo di ricorsione, sprecando quindi tempo.

È possibile quindi dire che esiste un sottografo comune, e che è il più grande, $S_{\text{opt}} \in B$ tale per cui S_{opt} contenga necessariamente una coppia di vertici $< v, w > \in C$, se per ogni sottoinsieme $X \times Y$ all'interno di $P(C)$ v e w sono, allo stesso tempo, adiacenti o non adiacenti a tutti gli altri vertici presenti in X e Y rispettivamente.

$$\forall X \times Y \in P(C) : (N(v, X) = X \setminus \{v\} \wedge N(w, Y) = Y \setminus \{w\}) \vee (N(v, X) = \emptyset \wedge N(w, Y) = \emptyset) \quad (1)$$

Di qui il seguente lemma:

Lemma 2 *Sia $B = (S, C, D)$ un ramo e $< v, w >$ una coppia di vertici $\in C$ che soddisfa l'equazione 1. Esiste un sottografo comune più grande, denominato come S_{opt} nel ramo B tale da contenere $< v, w >$.*

Vertex-Equivalence-based Upper Bound: Se si considera un ramo della ricorsione, ed il massimo comune sottografo in esso trovato(indicato con S'), è possibile terminare in anticipo la ricorsione se l'upper bound sulla dimensione del sottografo comune è minore o uguale a quella di S'.

Più l'upper bound è stringente, maggiore è il numero di rami che si possono eliminare dalla ricerca, risparmando quindi tempo.

Andando a considerare un sottografo comune S_{sub} trovato nel ramo $B = (S, C, D)$ ed un sottoinsieme $X \times Y \in P(C)$, si può andare a fare la seguente formulazione:

$$|S_{\text{sub}}| \cap X \times Y \leq ub_{\text{X,Y}} := min\{|X|, |Y|\} \quad (2)$$

Questo perchè, se fosse altrimenti, un sottografo comune andrebbe a contenere due coppie di vertici distinte $< v, w >$ e $< v', w' >$. Ciò porterebbe quindi uno dei due vertici, tra v e w, ad essere uguale a v' o w', rispettivamente, violando così la definizione di funzione biunivoca. Quindi, $ub_{\text{X,Y}}$ è l'upper bound del numero di coppie di candidati che rientrano in $X \times Y$ e sono presenti in un sottografo comune che si ritrova al ramo $B = (S, C, D)$.

Inoltre, poichè tutti i sottoinsiemi in $P(C)$ sono disgiunti, l'upper bound del ramo (S, C, D) sarà come segue:

$$|S_{\text{sub}}| \leq ub_{\text{S,C}} := |S| + \sum_{X \times Y \in P(C)} ub_{\text{X,Y}} \quad (3)$$

Le motivazioni che hanno indotto gli autori di RRSplit ad attuare tale riduzioni risiedono nel fatto che tale upper bound $ub_{\text{X,Y}}$ non sarebbe abbastanza stringente.

Inoltre, presa una coppia di vertici $< v, w > \in C$, se esiste una coppia di nodi $< v', w > \in D$: $v' = \Psi(v)$, ogni sottografo comune trovato in (S, C, D) non può includere $< v, w >$, portando tale coppia di vertici ad essere rimossa da C.

Da qui, quindi, il nuovo upper bound:

Si considera, per prima cosa, il sottoinsieme $X \times Y \in P(C)$ e un vertice, scelto in maniera arbitraria, $v \in X$, e si partizionano X e Y come segue:

$$X_{\text{L}} = X \cap \Psi(v), X_{\text{R}} = X \setminus X_{\text{L}} \quad (4)$$

$$Y_{\text{L}} = \{w | < v', w > \in D, v' \in \Psi(v)\}, Y_{\text{R}} = Y \setminus Y_{\text{L}} \quad (5)$$

In X_{L} appaiono tutti i vertici in X strutturalmente equivalenti a v, mentre Y_{L} contiene i vertici $w \in Y$ che appaiono in una coppia di nodi $< v', w > \in D : v' \in \Psi(v)$.

Da qui è possibile partizionare $X \times Y$ in $X_{\mathrm{L}} \times Y_{\mathrm{L}}$, $X_{\mathrm{L}} \times Y_{\mathrm{R}}$, $X_{\mathrm{R}} \times Y_{\mathrm{L}}$, $X_{\mathrm{R}} \times Y_{\mathrm{R}}$, ed è anche possibile scartare tutte le coppie di vertici che si ritrovano nel partizionamento $X_{\mathrm{L}} \times Y_{\mathrm{L}}$.

Inoltre, S_{sub} contiene al più $min\{|X_{\mathrm{R}}|, |Y|\}$ coppie di vertici nelle partizioni $X_{\mathrm{R}} \times Y_{\mathrm{L}}$ e $X_{\mathrm{R}} \times Y_{\mathrm{R}}$. Se fosse altrimenti ciò equivarrebbe ad ammettere l'esistenza di un vertice presente in $X_{\mathrm{R}} \cup Y$ che apparirebbe in due coppie di vertici distinti in s_{sub}, portando quindi S_{sub} a non essere un sottografo comune.

Procedendo allo stesso modo, è chiaro che S_{sub} contiene, al più,
$min\{|X_{\mathrm{L}}|, |Y_{\mathrm{R}}|, max\{|Y| - |X_{\mathrm{R}}|, 0\}\}$ coppie di nodi prese dalla partizione $X_{\mathrm{L}} \times Y_{\mathrm{R}}$.

S_{sub}, quindi, contiene non più di $ub_{(\mathrm{X,Y,D})}$ coppie di vertici da $X \times Y$, con

$$ub_{\mathrm{X,Y,D}} := min\{|X_{\mathrm{R}}|, |Y|\} + min\{|X_{\mathrm{L}}|, |Y_{\mathrm{R}}|, max\{|Y| - |X_{\mathrm{R}}|, 0\}\} \tag{6}$$

È possibile quindi andare a definire il nuovo upper bound, riguardante però un qualunque ramo (S, C, D), ed esso, indicato come $ub_{\mathrm{S,C,D}}$ viene definito in questo modo:

$$|S_{\mathrm{sub}}| \leq ub_{\mathrm{S,C,D}} := |S| + \sum_{X \times Y \in P(C)} ub_{\mathrm{X,Y,D}} \tag{7}$$

Prendendo in esame il codice originale di RRSplit, ci sono degli elementi aggiuntivi, rispetto a McSplit, che vengono adoperati per poter applicare le riduzioni viste nel capitolo precedente.
Essi sono:

EqClass È un vettore, che ha come dimensione il numero di nodi del grafo G, ed ha al suo interno gli indici dei nodi stessi di quest'ultimo.

Supponendo che G abbia dimensione 20, dentro EqClass avremo disposti, in maniera ordinata, dei numeri che vanno da 0 a 19, rappresentanti gli indici del grafo stesso.

Tale vettore viene utilizzato per la definizione di un upper bound più stringente rispetto a quello presente su McSplit, e rimane, nel corso delle iterazioni, invariato.

Inoltre, esso ha anche uno scope globale, ovvero può essere acceduto da qualunque funzione, senza la necessità di passarlo come argomento della funzione stessa;

index_right È anch'esso un vettore, che però ha come dimensione il numero dei nodi del grafo H, ma presenta anch'esso gli indici degli stessi.

Il suo scopo è quello di andare a definire delle nuove partizioni, nel momento in cui si debbano filtrare i vari bidomini, mettendo in campo la *vertex-equivalence-based reduction.*

A differenza di **EqClass**, questo vettore va incontro a modifiche, più nel preciso subisce una serie di scambi di posizione degli elementi che lo compongono.

Come per EqClass, anche index_right ha uno scope globale.

best_match È una variabile booleana che è utilizzata per implementare la *maximality-based-reduction.*

Algoritmo 3 : RRSplit, versione seriale

```
RRSplit(G, H, M, M_best, domains)
if |M| > |M_best| then
    M_best ← |M|
end if
if EqClass_new_upper_bound < |M_best| then
    return
end if
bd ← select_bidomain(domains)
v = find_min_value(bd.left)
remove_vtx_from_left_domain(v, bd.left)
best_match = false
for w ∈ bd.right do
    remove_vtx(w, bd.right)
    swap(index_right[w], index_right[right[bd.right + bd.right_len]])
    M.insert(v, w)
    new_domains ← filter_bidomains(domains, G, H, best_match)
    RRSplit(G, H, M, M_best, new_domains)
    M.pop_back()
    if best_match == true ∨ bound ≤ |M_best| then
        return
    end if
end for
Refine candidate set by removing Ψ_v
RRSplit(G, H, M, M_best, domains)
```

Come è già possibile evincere dall'algoritmo 3, tali modifiche vengono utilizzate principalmente per evitare che vengano fatte iterazioni inutili su rami che porterebbero o a soluzioni duplicate, oppure di non massima dimensione.

5 RRSplit parallelo, implementazione CPU

In questo capitolo verranno discusse tutte le modifiche e le aggiunte che sono state applicate ad RRSplit, su CPU, in modo tale che esso possa supportare il multi-threading.
Questa sezione è divisa in due sottosezioni: nella prima vengono mostrate le modifiche al codice che sono state fatte ad RRSplit, per poter supportare il multi-threading, nella seconda invece si metteranno in evidenza alcune problematiche riscontrate nell'andare a compiere questo lavoro.

5.1 Implementazione parallela

Nell'andare a sviluppare tale codice, sono stati adoperati gli schemi di parallelismo presenti nel lavoro parallelo di McCreesh, andando però a fare delle aggiunte necessarie per permettere l'utilizzo delle variabili che sono state aggiunte in RRSplit.

Algoritmo 4 : RRSplit, versione parallela

```
RRSplit(G, H, M, M_best, domains, index_right)
if |M| > |M_best| then
    M_best ← |M|
end if
if EqClass_new_upper_bound < |M_best| then
    return
end if
bd ← select_bidomain(domains)
if M.size() < 5 then
    RRSplit_par(G, H, M, M_best, domains, index_right)
end if
v = find_min_value(bd.left)
remove_vtx_from_left_domain(v, bd.left)
best_match = false
for w ∈ bd.right do
    remove_vtx(w, bd.right)
    swap(index_right[w], index_right[right[bd.right + bd.right_len]])
    M.insert(v, w)
    new_domains ← filter_bidomains(domains, G, H, best_match, index_right)
    RRSplit(G, H, M, M_best, new_domains)
    M.pop_back()
    if best_match == true ∨ bound ≤ |M_best| then
        return
    end if
end for
Refine candidate set by removing Ψ_v
RRSplit(G, H, M, M_best, domains, index_right)
```

Tra queste, la prima sostanziale modifica consiste nel non andare più a dichiarare index_right come vettore globale.
Tale scelta è stata fatta perchè inizialmente, come approccio risolutivo, per evitare che il vettore venisse modificato in maniera errata, si era tentato di usare i mutex per garantire l'atomicità degli swap.
Seppur formalmente corretto però, tale strategia risolutiva risultava essere estremamente inefficiente, portando quindi a valutare un differente tipo di strategia.
Una differente implementazione, che è risultata essere anche quella definitiva, è stata quella di andare a dichiarare ed inizializzare, all'interno del main, tale vettore, per poi procedere in tal modo:

- Tale vettore viene passato all'interno delle varie funzioni, ovvero mcs, solve e solve_nopar, filter_domains e le varie funzioni di partition, e questo perchè tali funzioni, che prima potevano accedervici senza che l'argomento venisse loro passato, ora non possono più farlo;
- Ciascun thread opera con la propria copia di index_right, che inizialmente è uguale per tutti, e prosegue nel proprio ramo della ricerca;
- Ogni volta che si deve proseguire in un sotto-ramo della ricerca, il vettore viene passato by reference. Questo permette non solo la modifica e la sua propagazione in avanti, ma anche che tali modifiche si possano propagare all'indietro, nel momento in cui fosse necessario fare backtracking, emulando così fedelmente il comportamento della versione seriale;
- Avendo ciascun thread la propria copia di index_right, non è più necessario applicare schemi di mutua esclusione, poichè ciascuno di essi modificherà la propria versione del vettore, evitando quindi problemi di concorrenza, e permettendo anche di mantenere il codice veloce.

Un'ulteriore modifica è stata anche implementata sulla porzione di codice atta alla definizione del nuovo upper bound: utilizzando una variabile *inc_size*, che viene inizializzata a zero nella funzione *mcs*, per poi venir propagata per ciascun ramo della ricerca, tale variabile, nel momento in cui si verifica la condizione $incumbent.size() < current.size()$, verrà modificata, e assumerà come valore $incumbent.size()$.
Essa viene usata come indice di partenza all'interno del primo loop, e questo perchè nell'implementazione originale vengono esaminate tutte le tuple della soluzione parziale, risultando in una perdita di tempo.
Usando invece *inc_size* come indice di partenza per andare a controllare solo le tuple che sono state recentemente inserite, è possibile ridurre il numero di cicli che tale loop deve eseguire, riducendo così anche il tempo impiegato.

```
for(VtxPair & a:current){
   if(EqClass[a.v]==EqClass[v]&&w<a.w) w=a.w;
}
```

Figura 21: Vecchio loop per il ritrovamento di vertici strutturalmente equivalenti, per definire il nuovo upper bound

```
for(int i = incumbent_size; i < current.size(); i++){
   if(EqClass[current[i].v] == EqClass[v] && w < current[i].w)
      w = current[i].w;
}
```

Figura 22: Nuovo loop per il ritrovamento di vertici strutturalmente equivalenti, per definire il nuovo upper bound

Per quanto riguarda la variabile *best_match*, non sono state apportate modifiche alla sua implementazione poichè, essendo tale variabile già dichiarata all'interno della funzione *solve*, la sua trasposizione in un ambiente parallelo non necessitava di modifiche.
A scopo di analisi, sono state utilizzate alcune variabili globali, che però non hanno alcuna finalità concernente il miglioramento delle prestazioni del codice, e sono:

solution_time viene utilizzata per tracciare il tempo necessario per il ritrovamento del primo massimo comune sottografo.

Ogni volta che $incumbent.size() < current.size()$, in tale variabile viene salvato il tempo che è stato impiegato per raggiungere tale soluzione parziale, e solamente nel momento in cui essa viene ingrandita potrà essere modificato;

max_sol_iter è l'equivalente di solution_time, ma per le iterazioni, e il suo funzionamento è analogo;

```
bool update(unsigned v, unsigned long long iterations){
   while(true){
      unsigned cur_v = value.load(std::memory_order_seq_cst);
      if(v > cur_v){
         if(value.compare_exchange_strong(cur_v, v,
         std::memory_order_seq_cst)){
            solution_time = steady_clock::now();
            max_sol_iter = iterations;
            return true;
         }
      }
      else
         return false;
   }
}
```

Figura 23: Esempio di aggiornamento del tempo e del numero di iterazioni per la prima soluzione massima

cuts questa variabile viene usata per evidenziare il numero di volte in cui la porzione di codice, che definisce il nuovo upper bound, innesca il pruning, portando all'uscita anticipata da quel ramo di ricerca, ed evitando quindi ricorsioni inutili.

```
if(w>0){
    int count_left=0, count_right=0;
    for (int i=bd.left_len; i>=0; --i)
       if(EqClass[left[bd.l+i]]==EqClass[v]) count_left++;
    for (int i=bd.right_len; i>=0; --i)
       if(right[bd.r+i]>w) count_right++;
    if(bd.left_len<=bd.right_len && count_left>count_right){
       if(bound+count_right-count_left<=incumbent.size()){
          cuts++;
          return;
       }
    }
    if(bd.left_len>bd.right_len &&
    (bd.right_len-count_right)>(bd.left_len-count_left)){
       if(bound+(bd.left_len-count_left)-
       (bd.right_len-count_right)<=incumbent.size()){
          cuts++;
          return;
       }
    }
}
```

Figura 24: Esempio di utilizzo della variabile cuts all'interno dei cicli di definizione del nuovo upper bound

5.2 Problematiche riscontrate

Nell'andare a fare le run per poter determinare quanto prestante fosse il codice, ci si è resi conto che RRSplit, già nella versione originale, ha manifestato dei bug in alcuni degli esperimenti compiuti(9 su 1050, per essere precisi).
In particolar modo, ci sono due caratteristiche, implementate nel codice, che hanno portato a trovare sempre una soluzione che fosse più piccola, rispetto a quella trovata da McSplit, di 1 di dimensione.
Le sezioni incriminate sono quelle relative alla $maximality - based - reduction$ e, citando l'algoritmo 3, al *Refine candidate set by removing* Ψ_v.

```
for (int i=bd.right_len; i>=0; --i) {
   idx = index_of_next_smallest(right, bd.r, bd.right_len+1, w);
   if(idx==-1)
      break;
   w = right[bd.r + idx];
   std::swap(index_right[w],index_right[right[bd.r + bd.right_len]]);
   right[bd.r + idx] = right[bd.r + bd.right_len];
   right[bd.r + bd.right_len] = w;
   auto new_domains = filter_domains(domains, left, right,
   g0, g1, v, w,best_match);
   current.emplace_back(VtxPair(v, w));
   solve(g0, g1, incumbent, current, new_domains, left, right,
   matching_size_goal,level+1, incumbent_size);
   current.pop_back();
   if(best_match||bound <= incumbent.size()) return;
}
```

Figura 25: Porzione del codice che crea bug. Nello specifico, è l'ultimo if, relativo alla riduzione per massimalità, a dare problemi

```
for(int i = 0; i<bd.left_len; ++i){
   if(EqClass[left[bd.l+i]]==EqClass[v]){
   std::swap(left[bd.l+i], left[bd.l+bd.left_len-1]);
   --bd.left_len; --i;
   }
}
```

Figura 26: Sezione di codice, relativo allo sfoltimento del candidate set, che crea bug

Ciò che può esser qui successo è che l'algoritmo, nell'andare a rimuovere dei rami di ricerca, da esso ritenuti inutili, abbia in realtà eliminato proprio quelli cruciali per andare a trovare con successo la soluzione di massima dimensione.
Non avendo trovato soluzione a ciò si è deciso, nella fase di running, di fare due run separate, testando sia il codice difettoso sia quello che non andasse ad utilizzare tali sezioni di codice, per capire quanta discrepanza potesse esserci nell'impiego o meno di quest'ultime.

Algoritmo 5 : RRSplit, versione parallela modificata

```
RRSplit(G, H, M, M_best, domains, index_right)
if |M| > |M_best| then
    M_best ← |M|
end if
if EqClass_new_upper_bound < |M_best| then
    return
end if
bd ← select_bidomain(domains)
if M.size() < 5 then
    RRSplit_par(G, H, M, M_best, domains, index_right)
end if
v = find_min_value(bd.left)
remove_vtx_from_left_domain(v, bd.left)
for w ∈ bd.right do
    remove_vtx(w, bd.right)
    swap(index_right[w], index_right[right[bd.right + bd.right_len]])
    M.insert(v, w)
    new_domains ← filter_bidomains(domains, G, H, index_right)
    RRSplit(G, H, M, M_best, new_domains)
    M.pop_back()
end for
RRSplit(G, H, M, M_best, domains, index_right)
```

6 Architettura GPU

Prima di esporre l'implementazione di RRSplit su GPU, però, è doveroso fare un'introduzione sommaria sul funzionamento stesso di una scheda grafica, comunemente detta GPU. Verranno esposti brevemente tutti i suoi livelli/tipi di memoria, mettendo in luce gli vantaggi/svantaggi che si ottengono nell'andare ad usare quel tipo di memoria, il tutto per rendere di più facile comprensione ciò che sarà materia di discussione nel capitolo seguente.
Tuttavia, è necessario specificare che, andando ad ampliare un lavoro precedente, che è stato svolto seguendo l'architettura CUDA, utilizzata nelle schede NVIDIA, si è proseguito in questa direzione, sfruttando quindi la medesima architettura.

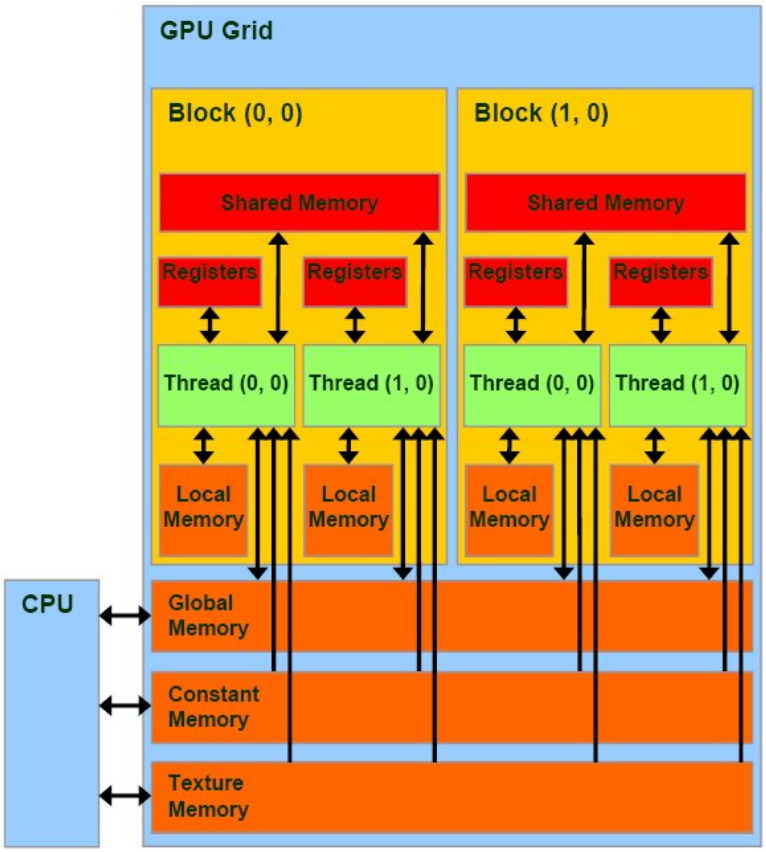

Figura 27: Architettura GPU NVIDIA
https://www.3dgep.com/cuda-memory-model/

Global Memory

Tutte le variabili dichiarate in questa memoria possono essere modificate da qualunque kernel e sono anche accessibili all'host attraverso specifiche funzioni implementate in CUDA.
È chiaro capire, però, che tale tipo di memoria è anche il più lento tra tutti, e generalmente viene adoperato solamente quando vi devono essere delle operazioni di copia di dati tra host e device.
Per andare ad incapsulare un dato nella memoria globale, è sufficiente che essa sia preceduta da __**device**__(ad esempio, __**device**__ **int n;** sta ad indicare che la variabile intera n risiede nella memoria globale, è ed quindi accessibile da chiunque).

Shared Memory

Questo livello di memoria, invece, viene usata per condividere una variabile che dovrà essere usata all'interno di un blocco, è può quindi essere letta e modificata solo dai thread che sono all'interno di quello stesso blocco.
Questa memoria, on-chip, è già più veloce rispetto a quella globale, e viene spesso usata quando si vuole che la variabile abbia una visibilità ridotta al blocco di thread.
Per andare a dichiarare una variabile come tale, la si fa precedere da **__shared__**.

Constant Memory

Questa memoria è particolare perchè, come il nome già anticipa, qui dentro risiedono i dati che non possono essere modificati in alcun modo, e che possono quindi essere soggetti solo a lettura.
È tuttavia simile alla *global memory*, perchè i dati qui presenti sono accessibili da chiunque. Ma, proprio per via della sua peculiarità, essa è comunque molto veloce, poichè presenta anche la memoria cache che velocizza le operazioni di lettura, risultando quindi più efficiente.
Per dichiarare una variabile come costante, bisogna farla precedere da **__constant__**.

Texture Memory

È una memoria progettata per essere read-only, ottimizzando gli accessi a dati presentanti un pattern di accesso specifico, quale la località spaziale.
È anche provvista di memoria cache, portandolo ad essere più veloce della *global memory*.

Local Memory

Fa riferimento ad una regione di memoria privata ad ogni singolo thread, la cui vita di un dato locale coincide con quella del thread, e tale memoria è fisicamente localizzata nella DRAM. Essendo off-chip, l'accesso a tale memoria è molto più lento rispetto all'accesso alla *shared memory* o ai *registers* on-chip.
Per dichiarare una variabile locale, basta farla precedere da **__local__**.

Registers

È il tipo di memoria più veloce presente sulla GPU, e ciascun thread può salvarvi all'interno dei dati, che però non sono accessibili agli altri thread.
Il numero di registri, però, è parecchio limitato, e nel momento in cui il numero di variabili dovessero eccedere quello dei registri, i dati in eccesso vengono salvati nella *local memory*

7 RRSplit, implementazione GPU

Come già accennato, per l'implementazione GPU del codice RRSplit è stato adoperato un framework già esistente, che andava ad eseguire l'algoritmo di McSplit su GPU usando CUDA.
Anche questa sezione presenterà una suddivisione in più sottosezioni: la prima consistente l'esposizione del framework esistente, che è stato utilizzato per implementare RRSplit su GPU, il secondo per evidenziare le modifiche ad esso apportate.

7.1 Framework esistente

Il lavoro dal quale si è partiti è quello di Gabriele Mosca [23], nel quale il codice di McSplit è stato riscritto per permetterne il suo funzionamento all'interno delle GPU moderne. Tale framework sfrutta un paradigma iterativo e non ricorsivo, e questo perchè tali GPU, avendo una memoria dedicata alle ricorsioni piuttosto limitata, non riuscirebbero a sostenere un numero così alto di ricorsioni come quello che si può riscontrare in McSplit.
Da questa premessa è quindi sorta la necessità di creare delle strutture dati che salvassero le informazioni importanti, senza doversi così affidare allo stack.
Tale struttura corrisponde ad un vettore di strutture dati, per permettere di utilizzare quanto meno memoria possibile, e segue ciò che è stato utilizzato in McSplit, salvando quindi tutte le informazioni cruciali per ciascun dominio. Essa è paerò anche ottimizzata in modo da utilizzare, per ciascuna di queste informazioni, un solo byte di memoria.
Questa ottimizzazione non viene però senza un "costo" poichè, operando in questo modo, la dimensione massima del grafo ne risulta significativamente ridotta(massimo ipotetico pari a 256). Nel framework, tuttavia, tale valore era stato impostato a 64, valore oltre il quale, molto probabilmente, le prestazioni del codice sarebbero state parecchio limitate.
La struttura dati implementata da Mosca, contenente 8 dati riguardanti i domini presi in esame, e che quindi occupa complessivamente 8 byte di memoria, è composta da:

- indice di inizio del dominio sul grafo di sinistra(L);
- indice di inizio del dominio sul grafo di destra(R);
- la lunghezza del dominio sul grafo di sinistra(LL);
- la lunghezza del dominio sul grafo di destra(RL);
- adiacenza o meno del dominio all'ultimo nodo selezionato, usato per i grafi connessi(ADJ);
- lunghezza della soluzione parziale corrente(P);
- ultimo nodo selezionato sul secondo grafo(W);
- lunghezza iniziale del grafo di destra, utilizzato per accertarsi che tutte le possibili combinazioni siano già state provate(IRL).

Tali domini vengono poi salvati in un vettore di dati e, potendo una soluzione includere molteplici domini, gli stessi vengono anche salvati in un secondo vettore di indici, che memorizza al suo interno le posizioni iniziale e finale dei vari domini che appartengono alla medesima soluzione.

7.2 Aggiunte e modifiche al codice

Le principali aggiunte che sono state fatte al codice hanno avuto lo scopo di implementare le caratteristiche chiave di RRSplit, già esaminate nella sezione 4.2.4, ad eccezione delle sezioni difettose, e sono:

- In graph.h sono state inserite le strutture dati presenti in RRSplit, ovvero il vettore *degree* e la matrice *adjlist*, che vengono utilizzate per le nuove funzioni implementate;
- L'aggiunta dei parametri che sono presenti in RRSplit, ma non in McSplit(EqClass e index_right);

```
__constant__ ui d_EqClass[MAX_GRAPH_SIZE];
__constant__ unsigned int d_degrees_g1[MAX_GRAPH_SIZE];
__constant__ unsigned int d_adjlist[MAX_GRAPH_SIZE][MAX_GRAPH_SIZE];
ui *EqClass;
unsigned int degrees[MAX_GRAPH_SIZE];
unsigned int adjlist[MAX_GRAPH_SIZE][MAX_GRAPH_SIZE];
```

Figura 28: Variabili aggiunte al framework

- L'aggiunta delle nuove funzioni di partizionamento del dominio del grafo di destra utilizzate da RRSplit, ma non presenti in McSplit;

```
__host__  __device__ uchar partition_right(uchar *arr, uchar start, uchar len,
                const uchar *adjrow, int *index_right) {
    uchar i=0;
    for (uchar j=0; j<len; j++) {
        if (adjrow[arr[start+j]]) {
            int_swap(&index_right[arr[start+i]],&index_right[arr[start+j]]);
            uchar_swap(&arr[start+i], &arr[start+j]);
            i++;
        }
    }
    return i;
}

__host__  __device__ uchar partition_sparse(uchar *arr, uchar start, uchar len,
        unsigned int degree, const ui * adjlist, int *index_right){
    uchar pos, j=0;
    for(uchar i=0;i<degree;++i){
        pos=index_right[adjlist[i]];
        if(pos>=start && pos<start+len){
            int_swap(&index_right[arr[start+j]],&index_right[arr[pos]]);
            uchar_swap(&arr[start+j], &arr[pos]);
            j++;
        }
    }
    return j;
}
```

Figura 29: Funzioni utilizzate per le partizioni del dominio del grafo di destra

- La sezione di codice che implementa l'upper più stringente rispetto a quello già sfruttato da McSplit;

```
for(uchar i = incumbent_size; i < domains[bd_pos - 1][P]; i++){
   if(d_EqClass[cur[i][L]] == d_EqClass[v] && w < cur[i][R])
      w = cur[i][R];
   }
if(w > 0){
   uint count_left = 0, count_right = 0;
   for (int i = bd[LL]; i >= 0; i--)
      if(d_EqClass[left[bd[L]+i]] == d_EqClass[v])
         count_left++;
   for (int i = bd[RL]; i >= 0; i--)
      if(right[bd[R]+i] > w)
         count_right++;
   if(bd[LL] <= bd[RL] && count_left > count_right){
      if(bound + count_right - count_left
      <= (uint)(inc_pos + 1)){
         bd_pos--;
         continue;
      }
   }
   if(bd[LL] > bd[RL] && (bd[RL] - count_right) >
   (bd[LL] - count_left)){
      if(bound + (bd[LL] - count_left) - (bd[RL] - count_right)
      <= (uint)(inc_pos + 1)){
         bd_pos--;
         continue;
      }
   }

}
```

Figura 30: Funzione di calcolo dell'upper bound usato in RRSplit_GPU

- lo swap che avviene all'interno del loop della funzione di risoluzione del mcs, sfruttando il vettore index_right, anche qui implementato;

Le modifiche attuate al codice, invece, sono le seguenti:

- Il limite della dimensione massima dei grafi è stato aumentato da 64 ad 80, poichè gli esperimenti qui condotti coinvolgevano grafi di tale dimensione;
- All'interno della funzione *generate_next_domains*, per il calcolo di r_len non viene più adoperata la funzione *partition* ma, dipendentemente dal soddisfacimento o meno di una condizione, *partition_sparse* o *partition_right*;

Tutte le implementazioni qui enunciate sono state applicate sia sulla porzione di codice relativo all'*host* che sul codice eseguito dal *device*.

8 Risultati

Di seguito, verranno presentati i risultati ottenuti andando ad utilizzare i codici su CPU, andando a confrontare RRSplit_parallelo con RRSplit_seriale e McSplit_parallelo.

8.1 Considerazioni sul dataset

È doveroso fare, tuttavia, delle considerazioni sul dataset ottenuto, prima di proseguire con ulteriori analisi.
Il dataset utilizzato è quello disponibile sul sito http://mivia.unisa.it/datasets/graph-database/arg-database/ [24] [25], e sono stati usati i grafi generati in maniera casuale, con due caratteristiche specifiche:

mcsXX fa riferimento alla percentuale del grafo che è identico al kernel di partenza, dal quale poi vengono generati tutti i grafi. Ciò significa che, preso un kernel dal quale sviluppare tutti grafi, mcs10 vuol dire che il 10% di quel grafo è identico al kernel di partenza;

rXXX Indica la densità di archi del grafo, ovvero: preso un grafo(in questo esempio connesso) con N vertici, il numero totale di archi che tale grafo può avere, al massimo, sarà uguale a N*(N-1). Con tale parametro, il numero totale di archi possibile sarà r*N(N-1)(con un caveat: se il numero di archi non è sufficiente per ottenere un grafo connesso, ne vengono aggiunti altri). Quindi, se si ha un grafo con 20 nodi ed r = 0.1, il numero massimo di archi che si potrà avere dovrebbe essere 0.1*20*19 = 38.

Partendo da questi presupposti, è stata fatta una run preliminare, con il semplice intento di vedere quali grafi andassero incontro al pruning presente in RRSplit, portando a queste considerazioni:

- Una fetta molto ampia dei grafi presente nel dataset *r005* ha effettivamente innescato il pruning, mentre lo stesso non si può dire per quelli presenti in *r01* ed *r02*. Infatti, se in *r01* vi erano alcuni casi, ma comunque significativamente minori rispetto a *r005*, in *r02* i grafi che innescano tale pruning sono ancora minori. Da qui si è potuto evincere, quindi, che con dei grafi più densi diviene più diffcile trovare strutture isomorfe, a meno che non si parli di grafi estremamente densi, anche se non è questo il caso;

- Il numero di grafi che innesca il pruning di RRsplit varia in base alla grandezza stessa del grafo e, più nello specifico, con l'aumentare dei nodi del grafo il numero totale diminuisce;

- Andando a variare il parametro "*mcsXX*" il numero di grafi che presentava strutture isomorfe è rimasto molto simile per *mcs10*, *mcs70* ed *mcs90*, con l'ultimo presentante il valore più alto. Questo suggerisce che, usando una struttura più simile al kernel di partenza, è più facile andare incontro a strutture isomorfe.

  I valori più bassi si sono riscontrati in *mcs30* e *mcs50*, ed il motivo per cui essi sono anche più bassi rispetto ad *mcs10* potrebbe essere il seguente: essendo la rimanente parte generata casualmente, è più probabile che in *mcs10* la parte generata casualmente porti a strutture isomorfe, così come è plausibile che in *mcs30* e *mcs50* la percentuale di kernel "in comune" non garantisca necessariamente la presenza di strutture isomorfe.

Fatte le dovute considerazioni per il dataset da utilizzare negli esperimenti, si è deciso di proseguire in questo modo:

- Sono stati presi 10 grafi che hanno innescato, nella run preliminare, il pruning di RRSplit;

- Tali grafi sono stati poi affiancati dalla loro controparte(ad esempio, A58 affiancato con B58 e viceversa), creando così il singolo *esperimento*;

- Per il grafo di "sinistra", che verrà denominato come grafo *pattern*, le dimensioni prese erano di 20, 25 e 30 nodi;
- Per il grafo di "destra", denominato grafo *target*, la dimensione andava dai 20 agli 80 nodi, ma facendo sempre in modo che non fosse più piccolo del grafo *pattern*(ad esempio, si può avere 30_A27 e 30_B27, 25_B12 e 50_A12, ma non 30_A00 e 20_B00);
- Per ciascuno di questi esperimenti sono state fatte 10 run, per verificare che i risultati fossero sempre consistenti;
- Tali procedure sono state ripetute per *mcs10*, *mcs30*, *mcs50*, *mcs70* e *mcs90*;
- Non essendovici abbastanza elementi in *r01* e *r02* per proseguire come sopra citato, tali run sono state eseguite solo su *r005*, prendendoli dal dataset *rand*.

8.2 Risultati CPU

In questa sezione verranno messi a confronto la prestazioni ottenute dagli algoritmi adoperati su CPU, e ci si soffermerà in particolar modo su:

- I tempi medi per run che gli algoritmi hanno impiegato per esplorare tutto lo spazio delle soluzioni;
- I tempi medi per run che essi hanno richiesto per risalire alla prima soluzione di massima dimensione;
- Il numero di iterazioni medio per run che ciascun algoritmo ha eseguito per esplorare tutte le posibili opzioni;
- Il numero di iterazioni medio per run che gli stessi hanno richiesto per raggiungere la prima soluzione di massima dimensione possibile;
- **Solo per RRSplit**: la percentuale di volte che il pruning è stato innescato rispetto al numero di iterazioni per run;

8.2.1 RRSplit parallelo e RRSplit seriale

Andando a confrontare le prestazioni ottenute da RRSplit, e dalla sua versione parallela, si può riscontrare un netto miglioramento delle prestazioni.

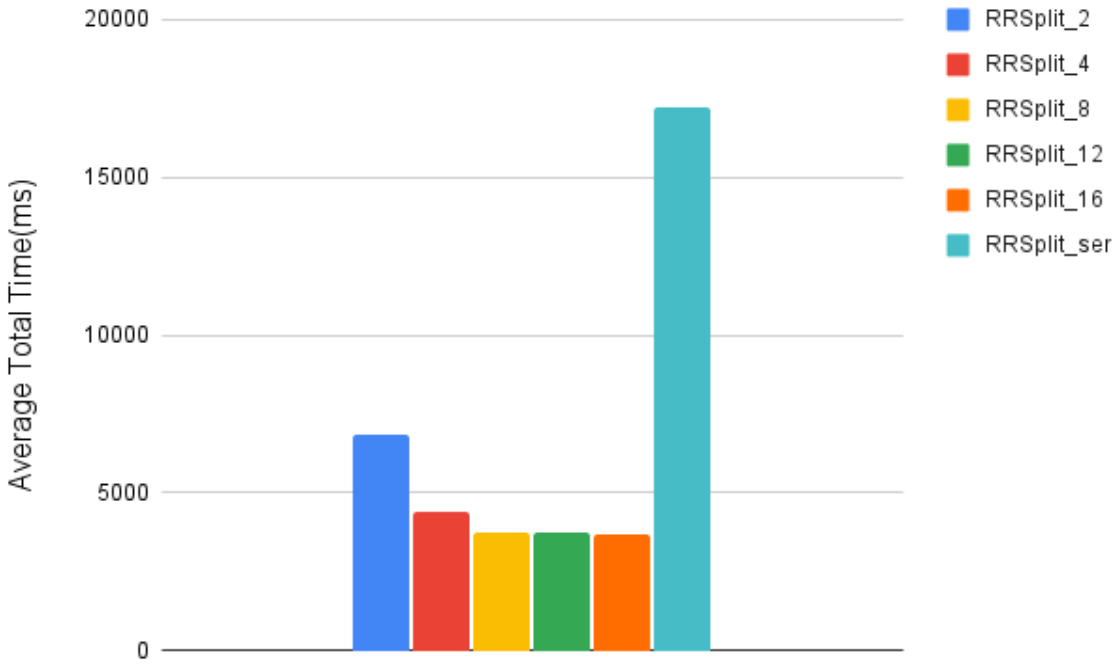

Figura 31: Tempo medio per run per esplorare tutte le possibili soluzioni

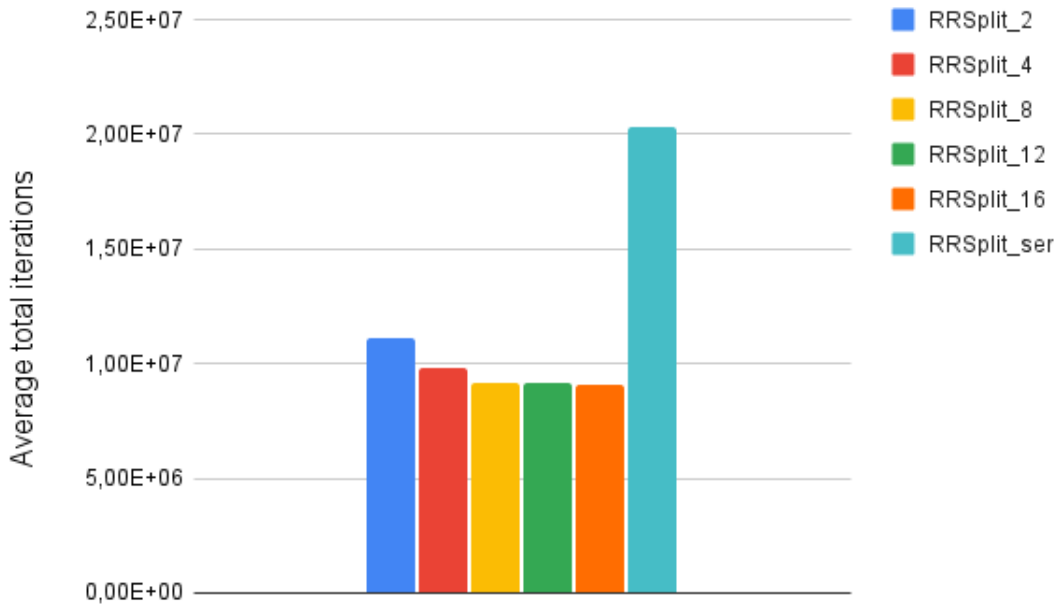

Figura 32: Iterazioni medie per run per esplorare tutte le possibili soluzioni

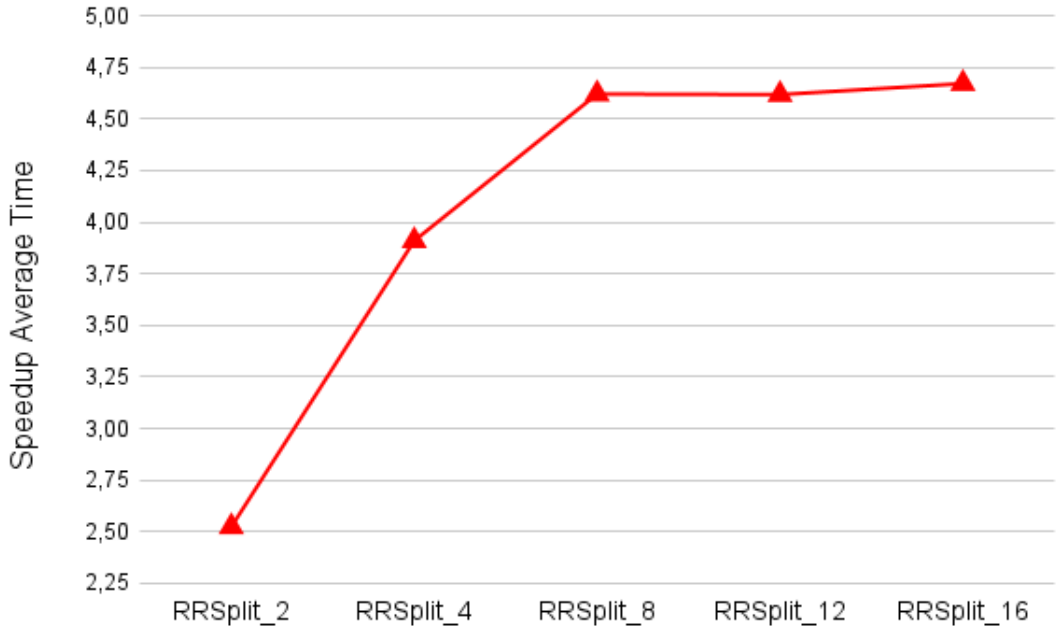

Figura 33: Speedup, rispetto ad RRSplit seriale, relativo al tempo medio impiegato per l'esplorazione di tutte le possibili soluzioni

Già andando a considerare i seguenti grafici, si può notare un miglioramento significativo per quanto concerne sia i tempi che le iterazioni richiesti per esplorare tutte le possibili soluzioni. I tempi vengono infatti ridotti fino a quasi 5 volte, e le iterazioni per un valore che, escluso il caso 2 thread, è sempre maggiore di 2, presentando quindi una consistente diminuzione delle iterazioni richieste per poter esplorare tutto lo spazio delle soluzioni.
Da questi grafici, inoltre, è possibile notare come prendere un numero di thread maggiore di 8 sembra non portare più a dei miglioramenti significativi, poichè i valori ottenuti in figura 33 risultano essere parecchio simili. Questo, molto probabilmente, è dovuto alle dimensioni dei grafi *pattern* che, non superando i 30 nodi di grandezza, portano l'algoritmo, quando usa un numero elevato di thread, ad esplorare rami non importanti per la definizione della soluzione.

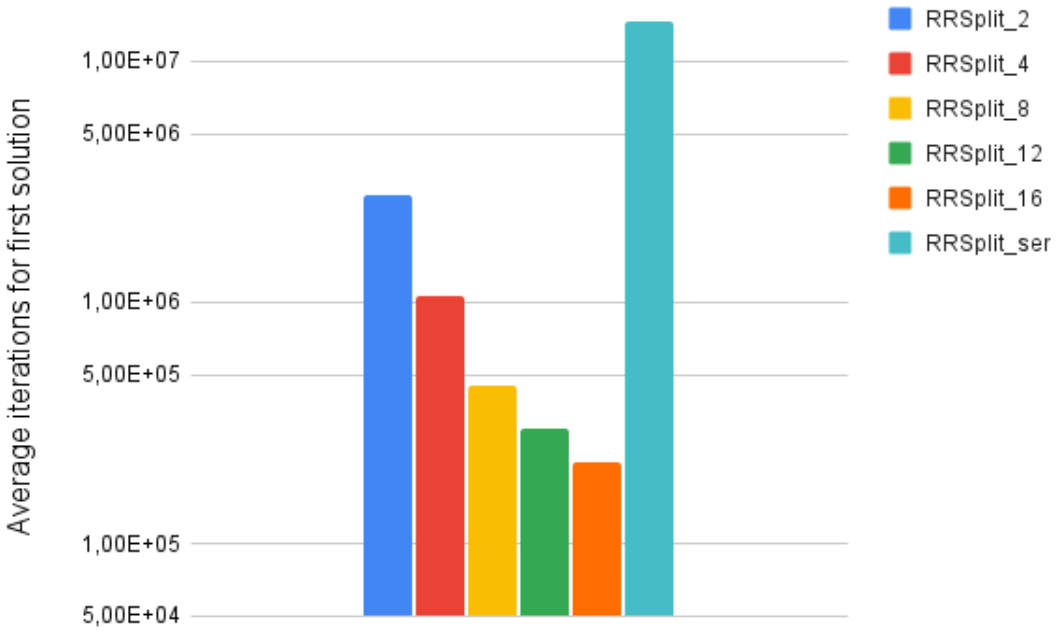

Figura 34: Iterazioni medie per run per trovare la prima soluzione di massima dimensione

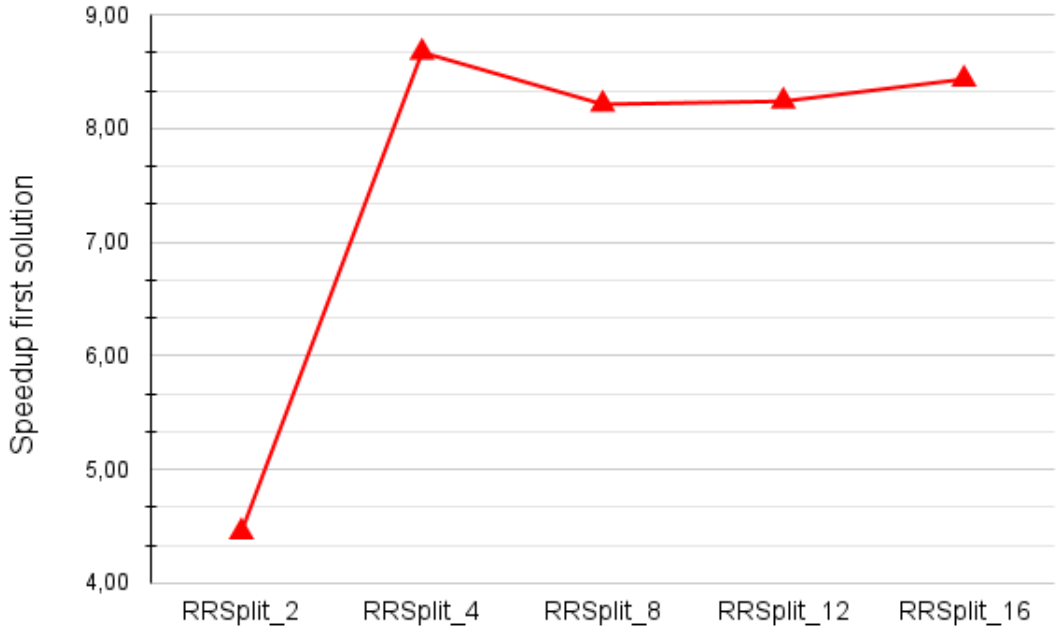

Figura 35: Speedup, rispetto a RRSplit seriale, relativo al tempo medio impiegato per trovare la prima soluzione di massima dimensione

Prendendo in esame il grafico 34, si può evidenziare una differenza con i risultati espressi in precedenza: è possibile notare che, per un numero di thread maggiore o uguale a 8, i tempi e le iterazioni richieste per l'esplorazione di tutte le possibili soluzioni non cambiano. Lo stesso non si può tuttavia dire per le iterazioni richieste per trovare la prima soluzione di dimensione massima.
È anzi possibile notare una diminuzione delle iterazioni qui richieste per un valore che è ben oltre 50 nel caso $n\_threads = 16$, e che si attesta ad un valore tra 10 e 50 per i casi 4, 8, 12 thread.
Inoltre, confrontando i grafici nelle immagini 32 e 34, si nota come per RRSplit seriale la riduzione delle iterazioni sia pari solo a poco più di 2 circa, mentre negli altri casi è possibile anche superare il fattore di riduzione 10. Questo mette in evidenza che, con una soluzione multi-thread, la convergenza verso la prima soluzione diventa via via più rapida rispetto all'approccio single-thread.
È tuttavia possibile notare, in questo caso, come mostrato dal grafico 35, che per $n\_threads = 4$ si ha lo speedup migliore in termini di tempo richiesto per il ritrovamento della prima soluzione a grandezza massima, avvicinandosi ad un valore pari a 9. Questo potrebbe essere dovuto ad eventuali rallentamenti dovuti a carichi sbilanciati, nel caso in cui il numero di thread dovesse essere elevato, portando quindi uno o più thread ad esplorare rami inutili per la definizione della soluzione.
Andando con $n\_threads \geq 8$ invece, il fattore di speedup varia relativamente poco, un valore che quindi non sembrerebbe giustificare l'utilizzo di un numero di thread così elevato.

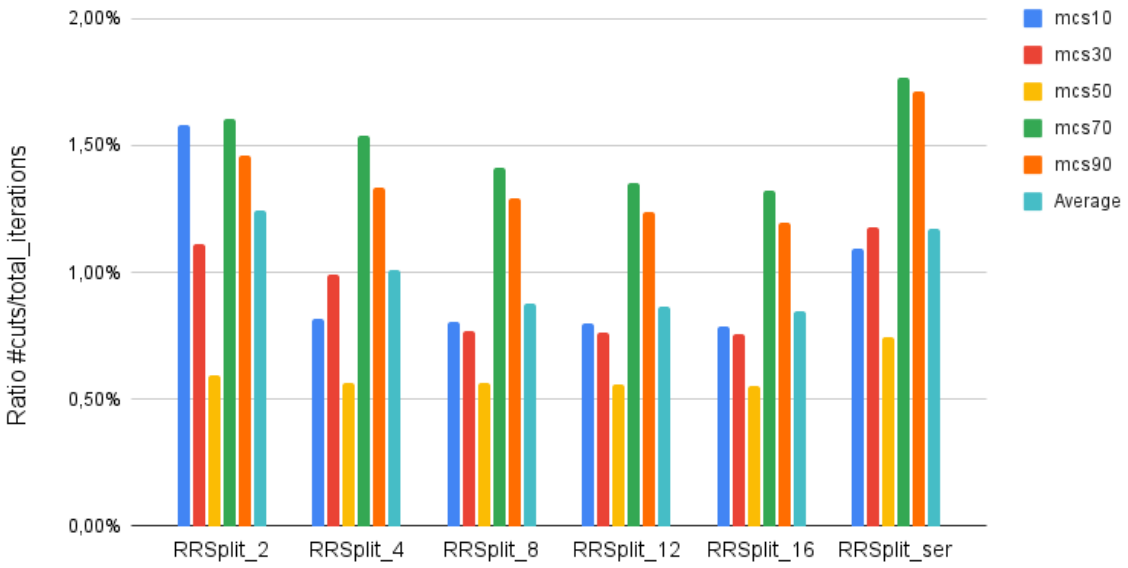

Figura 36: Rapporto percentile tra numero di inneschi del pruning(#cuts) e numero di iterazioni totali per run

Continuando con le analisi, ed esaminando il rapporto, espresso in percentuali, delle volte in cui il pruning di RRSplit è stato innescato rispetto al numero totale di iterazioni, è possibile fare delle considerazioni:

- L'andamento generale è che all'aumentare dei thread la precentuale diminuisce. Ciò può essere dovuto al fatto che, utilizzando la programmazione multi-thread, che però sfrutta alcune variabili globali per poter ricercare una soluzione ottima, esso comporti una riduzione delle iterazioni "inutili". Tuttavia, attivandosi prima queste condizioni rispetto al pruning, anche il numero degli inneschi viene ridotto;
- I valori, eccettuato per $RRSplit\_2\_mcs10$, rimangono coerenti con il variare del numero di thread;
- Arrivati a $n\_thread = 8$ i valori rimangono pressochè invariati.
- I valori seguono quello che era stato già anticipato in precedenza: siccome *mcs70* e *mcs90* condividono una percentuale di grafo in comune maggiore rispetto alle altre controparti, e siccome *mcs10* ha più probabilità di avere parti generate in modo casuale che siano isomorfe rispetto a *mcs30* e *mcs50*, i grafi che innescano più volte il pruning sono quelli appartenenti alla categoria *mcs70* e *mcs90*.

  Invece, *mcs50* è la categoria che presenta meno pruning rispetto agli altri, con *mcs10* e *mcs30* che presentano valori simili, un po' più alti per *mcs10*.

Da queste osservazioni, pertanto, è possibile quindi comprendere che l'implementazione multi-thread dell'algoritmo di RRSplit abbia beneficiato di tali modifiche, poichè esso ha permesso riduzioni significative di tempo richiesto sia per esplorare tutte le soluzioni, che per trovare la prima soluzione di dimensione massima, e lo stesso si può dire per le iterazioni totali e richieste per il ritrovamento della prima soluzione ottimale.

8.2.2 RRSplit difettoso

Per dovere di completezza, al fine di poter comprendere di quanto le ottimizzazioni scartate, seppur presentanti dei bug, possano aver inciso sulle prestazioni degli algoritmi di RRSplit, seriale e parallelo, ulteriori run sono state portate a compimento, e sono state fatte le valutazioni come si è già visto in precedenza. Da ciò è emerso che i risultati, pur essendo più ravvicinati rispetto a quelli precedenti, mostrano ancora come l'implementazione parallela, applicata su RRSplit, sia efficace. Vengono messe in evidenza delle prestazioni che, sia per la versione seriale, che per quella parallela, seppur di un margine minore, migliorano con l'utilizzo di tali sezioni di codice.

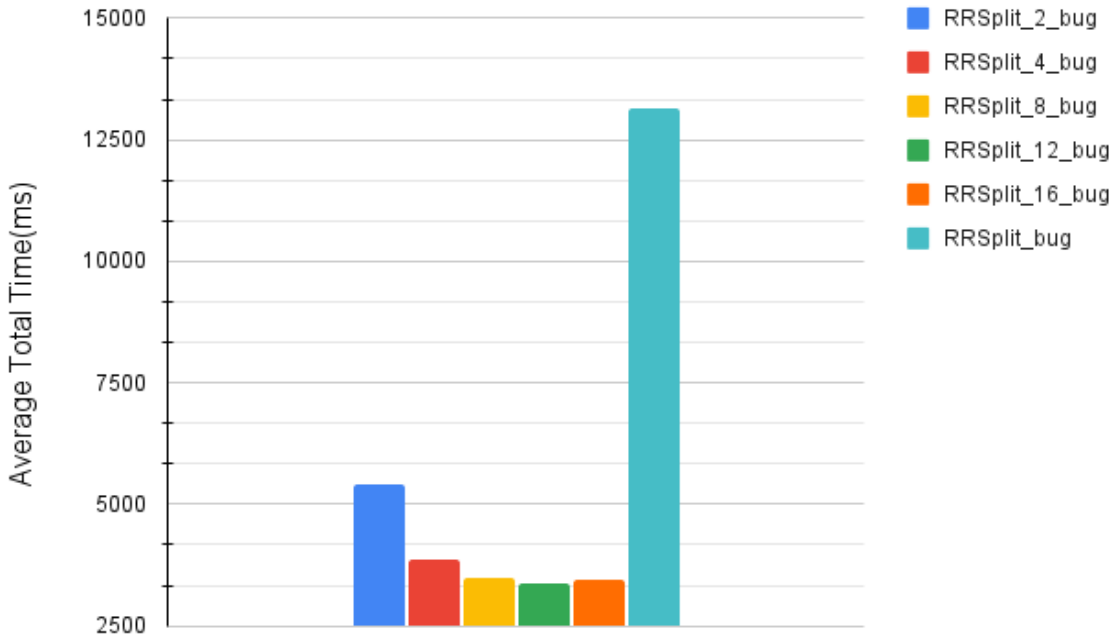

Figura 37: Tempo medio per run per esplorare tutte le possibili soluzioni

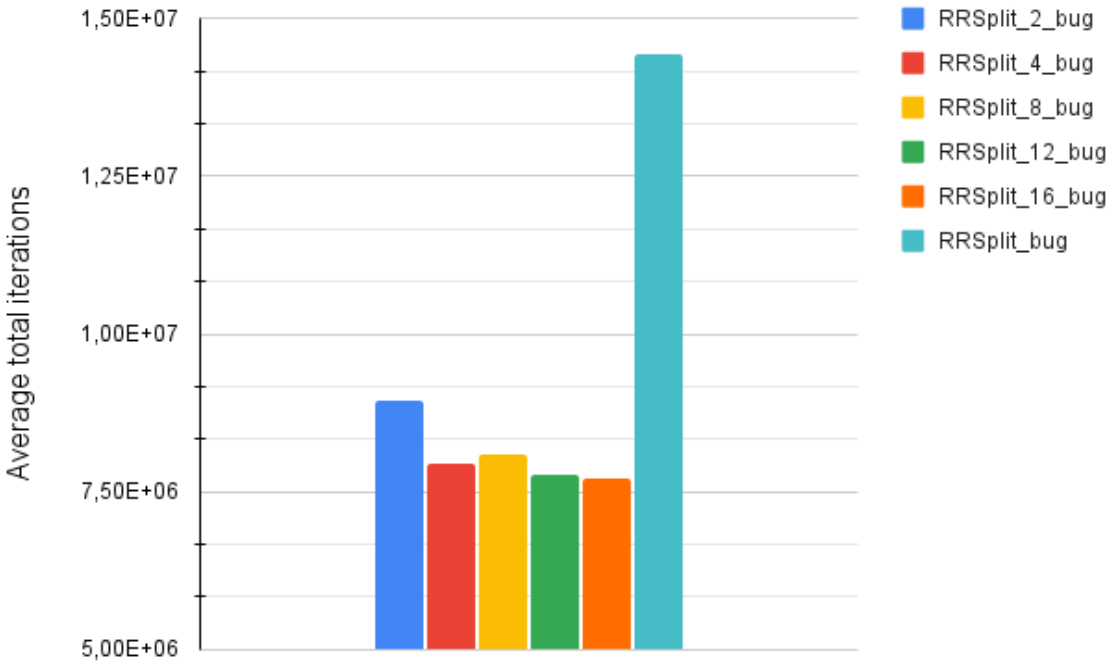

Figura 38: Iterazioni medie per run per esplorare tutte le possibili soluzioni

Come è possibile notare dai grafici in figura, l'impiego degli schemi di parallelismo ha permesso una riduzione significativa dei tempi e delle iterazioni richieste per poter esplorare tutte le possibili soluzioni.

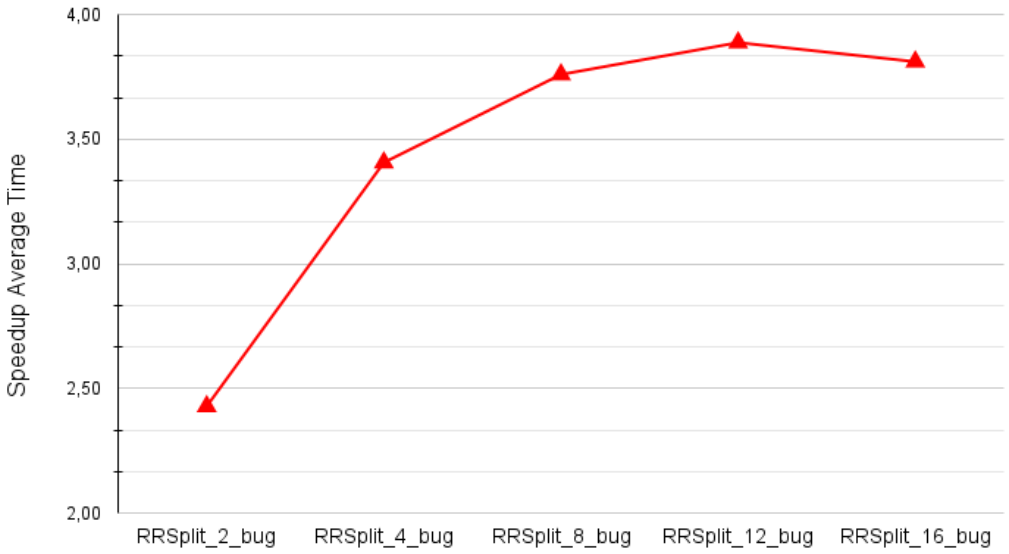

Figura 39: Speedup, rispetto a RRSplit seriale, sul tempo per esplorare tutte le possibili soluzioni

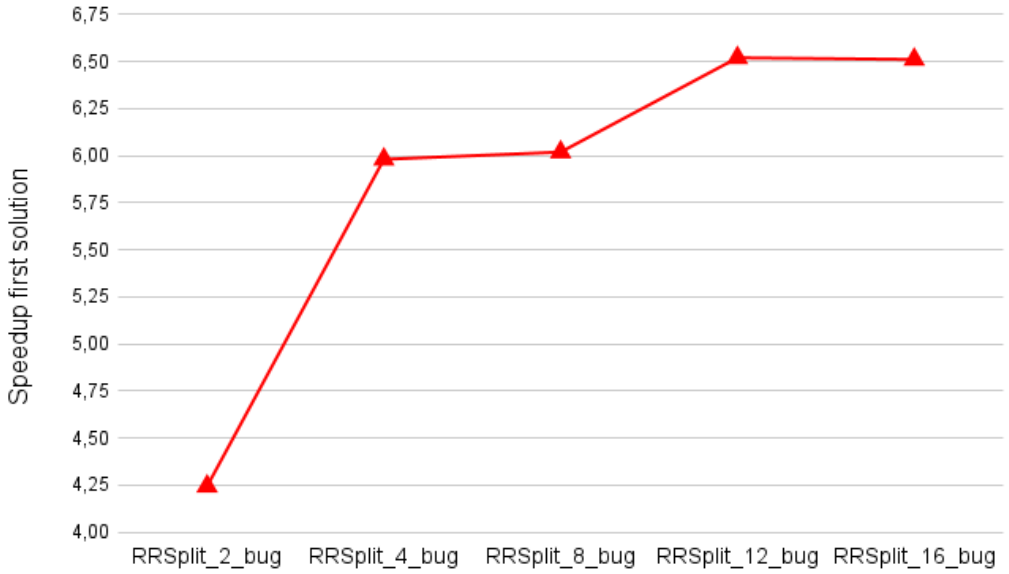

Figura 40: Speedup, rispetto ad RRSplit seriale, sul tempo per trovare la prima soluzione di dimensione massima

Esaminando i risultati presenti nelle figure 39 e 40, le prestazioni, per quanto minori rispetto a quelle citate nella sezione 8.2.1, rimangono comunque abbastanza elevate da giustificare un approccio multi-thread su RRSplit, poichè i tempi richiesti per esplorare tutte le possibili soluzioni e quelli per trovare la prima soluzione ottimale vengono significativamente ridotti.

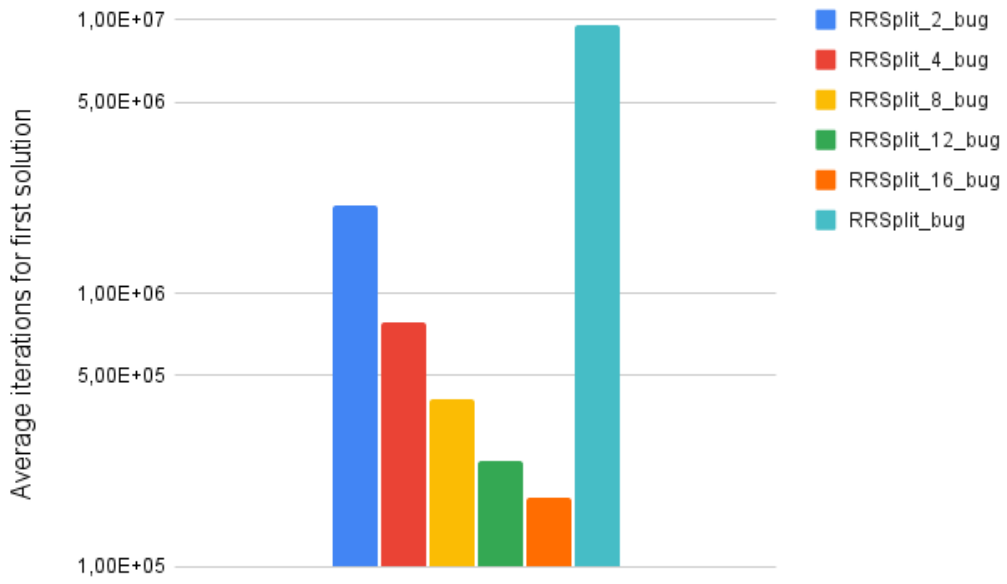

Figura 41: Iterazioni medie per run per trovare la prima soluzione di dimensione massima

Anche nel caso riguardante il numero totale di iterazioni necessarie per trovare la prima soluzione di dimensione massima, essi si riducono in maniera efficace, permettendo ad RRSplit parallelo di essere sempre più efficiente rispetto alla controparte seriale.

8.2.3 RRSplit parallelo e McSplit parallelo

Ciò che si può notare, andando ad esaminare i grafici qui presenti, invece, è che le prestazioni di RRSplit parallelo, messe a confronto con quelle di McSplit parallelo, per quanto migliori, siano comunque abbastanza ravvicinate.

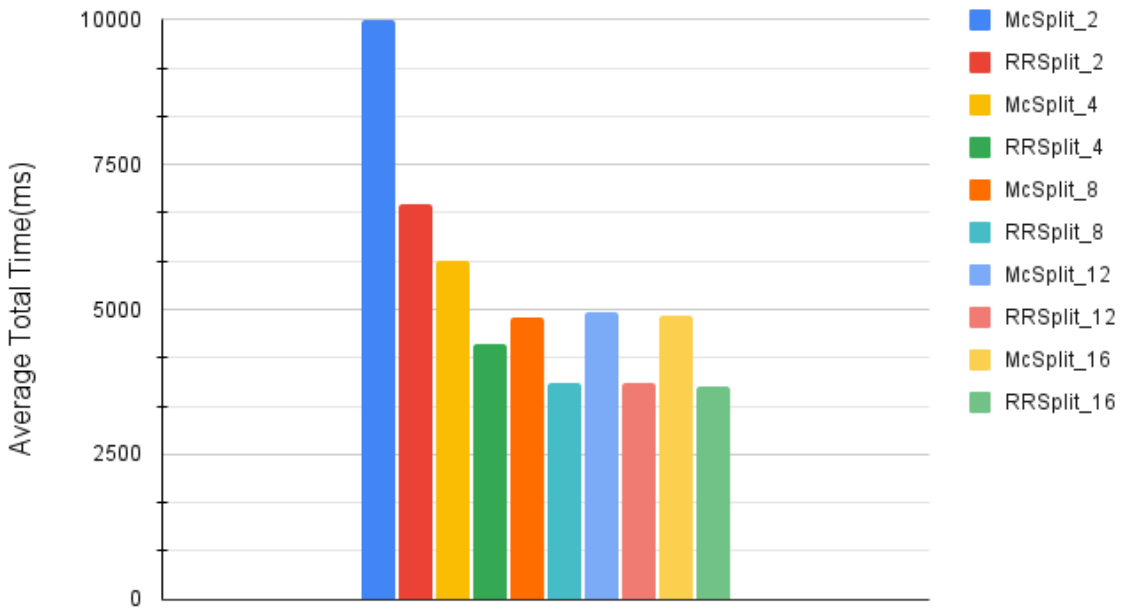

Figura 42: Tempo medio per run per esplorare tutte le possibili soluzioni *(ms)*

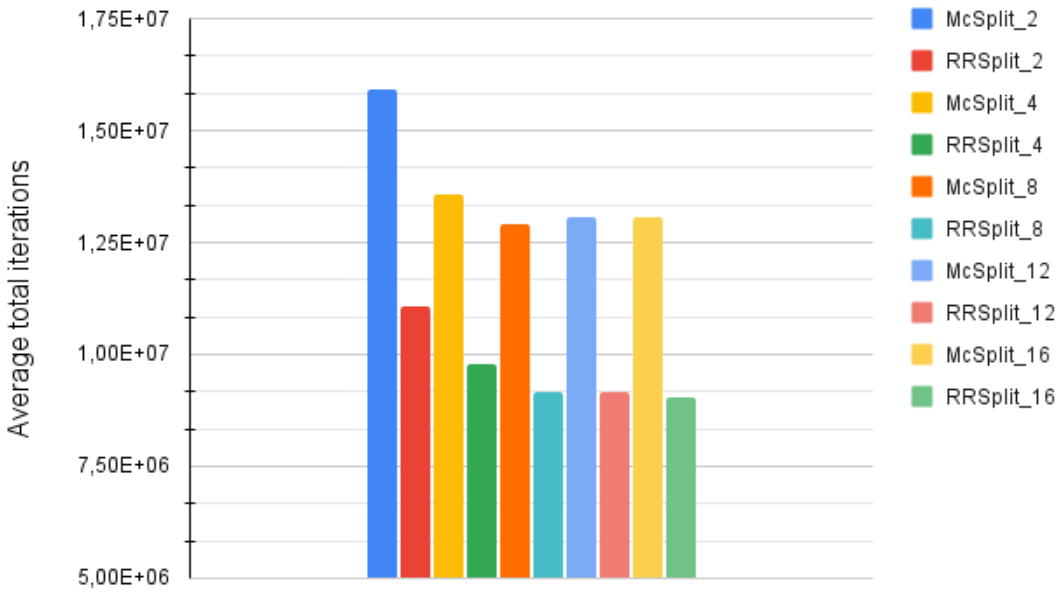

Figura 43: Iterazioni medie per run per esplorare tutte le possibili soluzioni

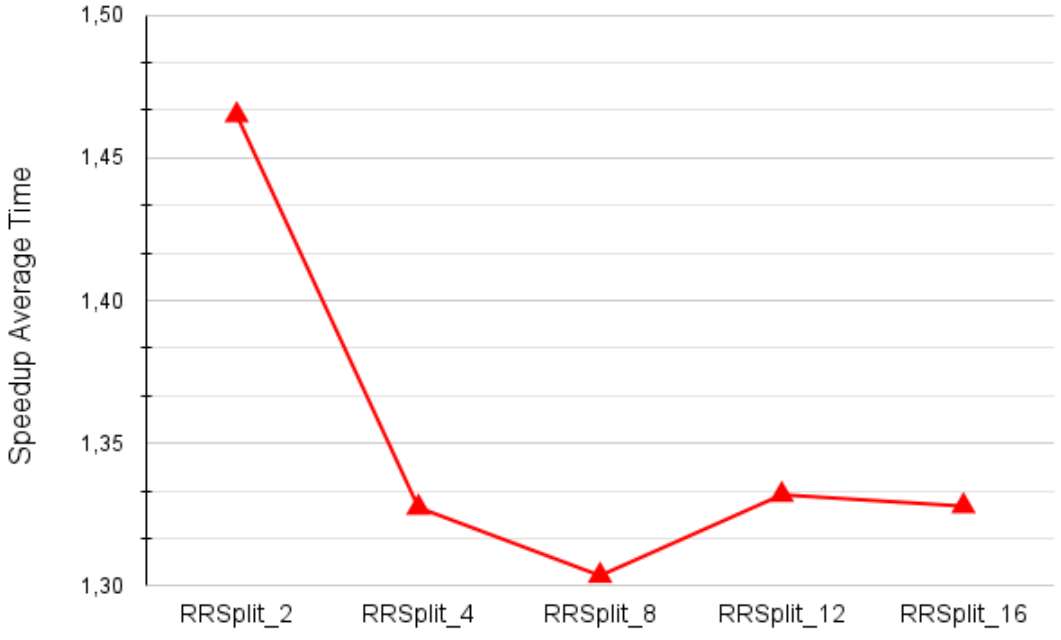

Figura 44: Speedup, rispetto a McSplit parallelo, relativo al tempo medio impiegato per l'esplorazione di tutte le possibili soluzioni

Osservando i risultati riportati dai grafici nelle figure 43 e 44, infatti, si può notare che, escludendo il caso $n\_thread = 2$, che ha beneficiato in misura leggermente maggiore rispetto agli altri casi, i tempi e le iterazioni richiesti per esplorare tutto lo spazio delle soluzioni migliora di un fattore pressochè simile, inferiore a 1.5.

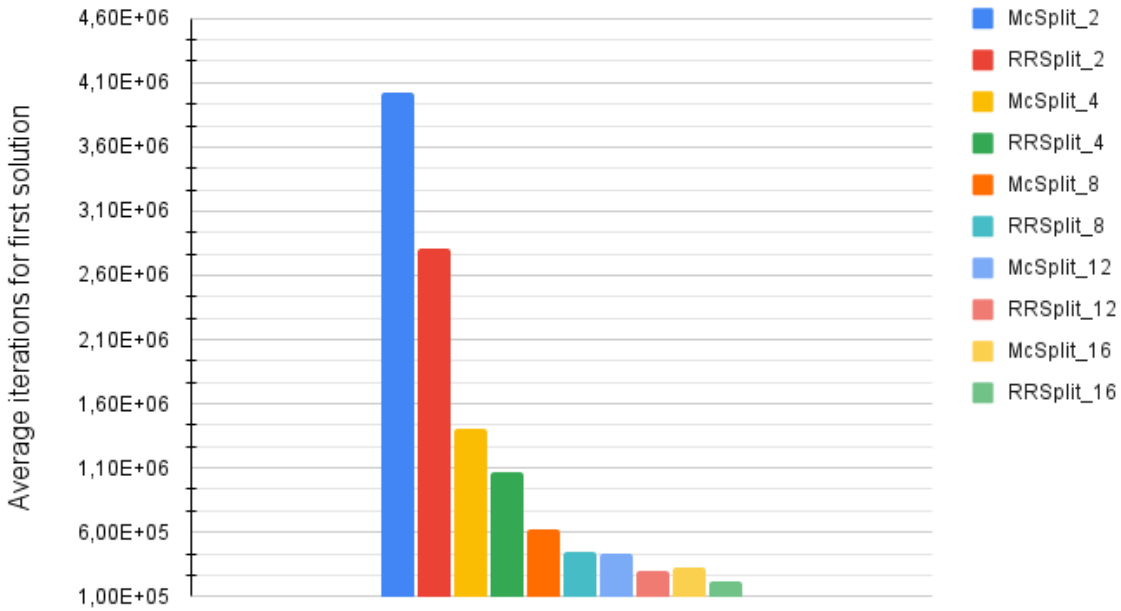

Figura 45: Iterazioni medie per run per trovare la prima soluzione di massima dimensione

Figura 46: Speedup, rispetto a McSplit parallelo, relativo al tempo medio impiegato per trovare la prima soluzione di massima dimensione

Osservando le figure 45 e 46, ciò che si nota è che lo speedup relativo al tempo medio impiegato per trovare la prima soluzione di dimensione massima, così come anche il fattore di riduzione delle iterazioni richieste per andarla a ritrovare, rimangono consistentemente attorno ai valori di prima, con RRSplit che mantiene sempre delle prestazioni migliori rispetto a McSplit.
È pertanto possibile concludere che, sebbene non nella stessa misura che si è potuta riscontrare in RRSplit seriale, l'utilizzo delle caratteristiche presenti in RRSplit ha permesso un ottenimento di prestazioni comunque superiori a McSplit, mantenendo sempre un discreto distacco per quanto riguarda sia i tempi che le iterazioni richieste per la risoluzione del problema, rendendo quindi valido l'approccio utilizzato.

8.2.4 Osservazioni aggiuntive

È degno di nota, però fare delle considerazioni aggiuntive riguardo a come la percentuale di somiglianza del grafo, rispetto ad un kernel di partenza, possa aver influito sulle prestazioni.

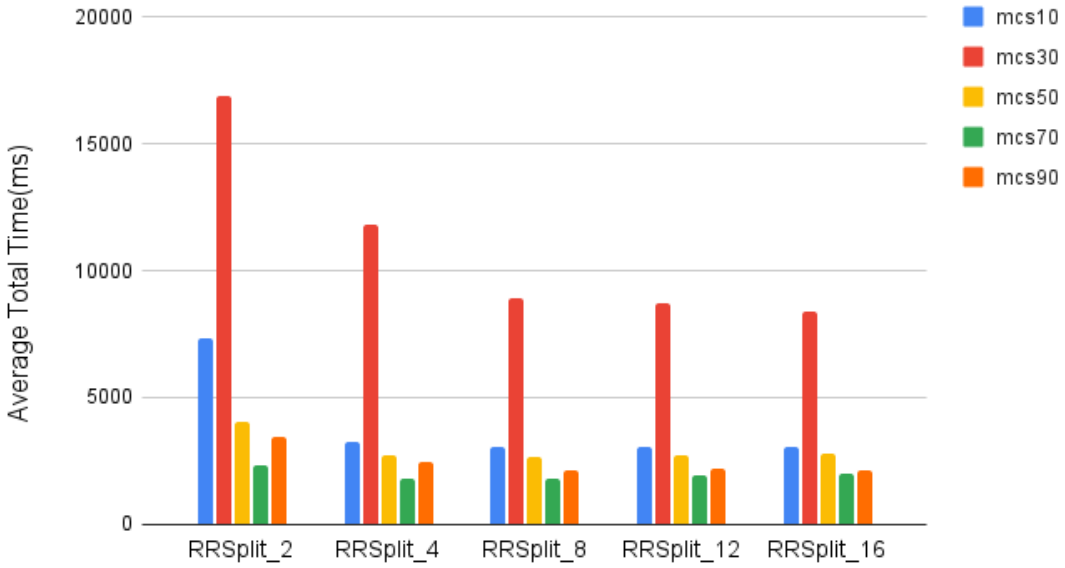

Figura 47: Tempi medi totali per run, in base al fattore *mcs*, RRSplit parallelo

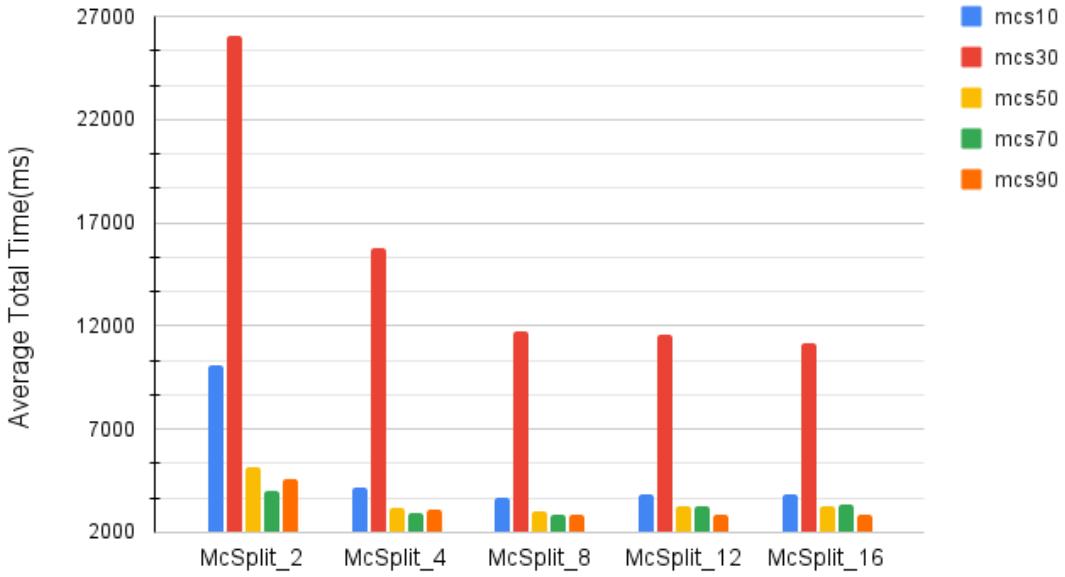

Figura 48: Tempi medi totali per run, in base al fattore *mcs*, McSplit parallelo

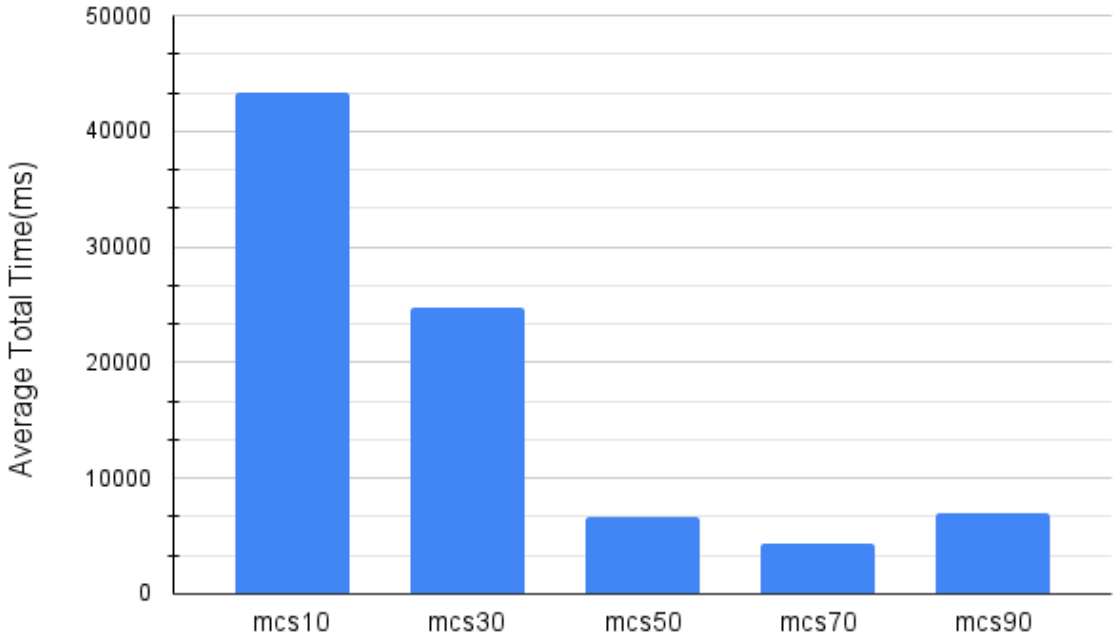

Figura 49: Tempi medi totali per run, in base al fattore *mcs*, RRSplit seriale

Come già anticipato, sono state fatte delle run preliminari per andare a vedere quali grafi, essendo isomorfi, andassero ad innescare il pruning.
Da ciò è emerso che quelli in *mcs30* e *mcs50* hanno avuto i risultati peggiori, così come anche, esaminando i dati della figura 36, si fosse notato come *mcs30* fosse il secondo gruppo di grafi con la percentuale più bassa.
Esaminando anche i tempi medi totali ottenuti dalle run, andando però a differenziare in base a questa caratteristica, si può notare come, a parte che per RRSplit seriale, che presenta dei tempi molto più lunghi in *mcs10*, *mcs30* è stato il sottogruppo che ha presentato, in maniera consistente, le prestazioni peggiori.
Ciò è dovuto, con buona probabilità, al fatto che i grafi, tolta la percentuale di kernel che li accomuna, sono generati in maniera casuale. Se la percentuale di kernel in comune è già molto alta di base, vi è più probabilità che i grafi siano isomorfi mentre, al contrario, se la percentuale è molto bassa(come il 10% per l'appunto), può succedere, con più probabilità, che la parte generata in modo casuale possa essere isomorfa. Questo dunque permetterebbe, nel caso di RRSplit, di attivare più spesso il pruning ed eliminare i rami inutili e, nel caso di McSplit, di trovare più facilmente la soluzione ottimale, portando anch'esso a scartare rami che non amplierebbero la soluzione, ma che anzi rallenterebbero l'intero processo.
Tuttavia, l'andamento che si riscontra in RRSplit seriale, e che si differenzia dai risultati ottenuti dai codici paralleli, può essere dovuto dal seguente fatto: essendo in *mcs10* la percentuale di kernel in comune minore rispetto agli altri casi l'algoritmo, a differenza delle implementazioni parallele, che possono esplorare contemporaneamente più sotto-rami di ricerca, abbia trovato difficoltà nel trovare strutture isomorfe. Tale difficoltà può aver impedito l'algoritmo di ottimizzare, per questo caso, i tempi di risoluzione.

8.3 Risultati GPU

Per la sezione GPU, onde evitare che i tempi delle run diventassero spropositatamente lunghi, è stato deciso di porre come limite di tempo della run stessa, oltre la quale l'algoritmo cessa l'esecuzione, 20 minuti.
Inoltre, vi sono due parametri aggiuntivi, non presenti nei codici su CPU: ovvero:

BLOCK_SIZE Indica il numero di thread all'interno dello stesso blocco, e che quindi lavorano contemporaneamente su un kernel GPU;

N_BLOCKS Questo parametro specifica quanti blocchi, invece, sono presenti. È utile ricordare che i thread all'interno di diversi blocchi non possono condividere tutti i dati, e che quindi regolare tale valore equivale un po' come andare a specificare il numero di thread CPU che vengono utilizzati per il processo.

Per $BLOCK\_SIZE$ si è deciso di impostarlo a 512, mentre N_BLOCKS ha assunto come valore, per questi esperimenti, 64.
La scelta del valore di tali parametri è stata attuata dopo una run preliminare dove, avendo anche qui impostato un timeout di 20 minuti, per evitare tempi di risoluzione troppo tediosi, sono emerse le seguenti considerazioni:

- Con il diminuire di $BLOCK\_SIZE$ le prestazioni generali delle run peggioravano gradualmente
- Per valori più piccoli di N_BLOCKS ciò che si riscontrava era un miglioramento delle prestazioni sulle run aventi, come grafo *pattern*, un grafo di 20 nodi, così come anche su alcune run con il grafo *pattern* con dimensione 25. Questo però andava a peggiorare lievemente alcune run che, presentando il grafo *pattern* di dimensione 25, avevano, come grafo *target*, dei grafi con almeno 50-60 nodi. Allo stesso modo venivano anche rallentati, anche in maniera significativa, tutte le run che presentassero, come grafo *pattern*, un grafo di 30 nodi.

Da queste premesse, le run sono state eseguite in maniera analoga a come già spiegato in precedenza nella Sezione 8.1.
Prendendo però solo le run che sono state portate a pieno compimento, e che quindi non sono state interrotte dal timeout, e dalle quali si è riusciti a trovare la soluzione di massima dimensione in modo corretto, è possibile constatare che le prestazioni di RRSplit_GPU, per quanto molto vicina a quella di McSplit_GPU, siano sempre leggermente peggiori.

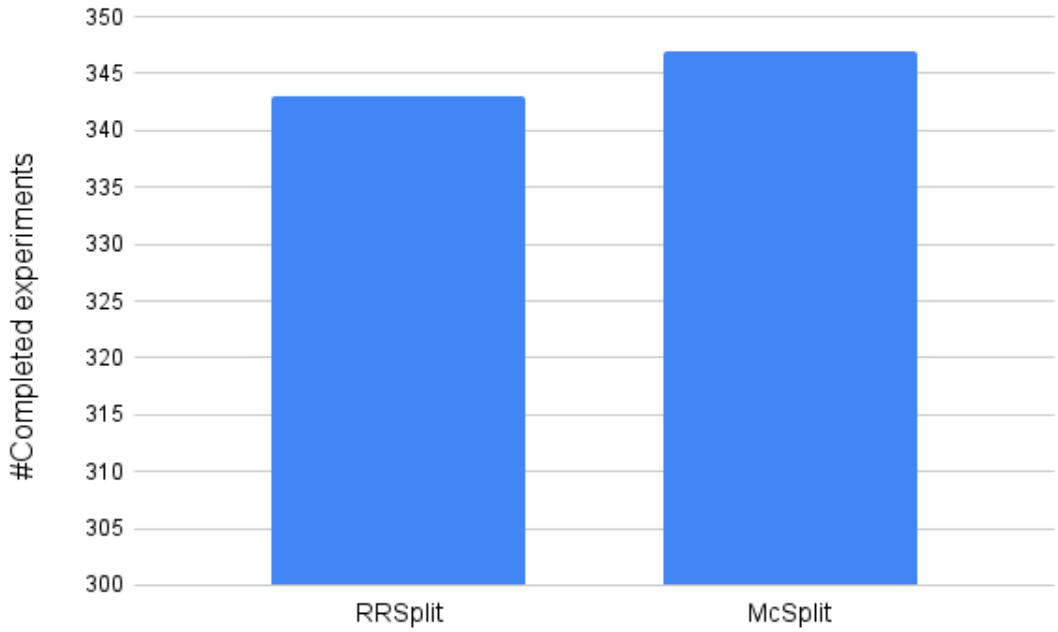

Figura 50: Numero di esperimenti, su GPU, completati

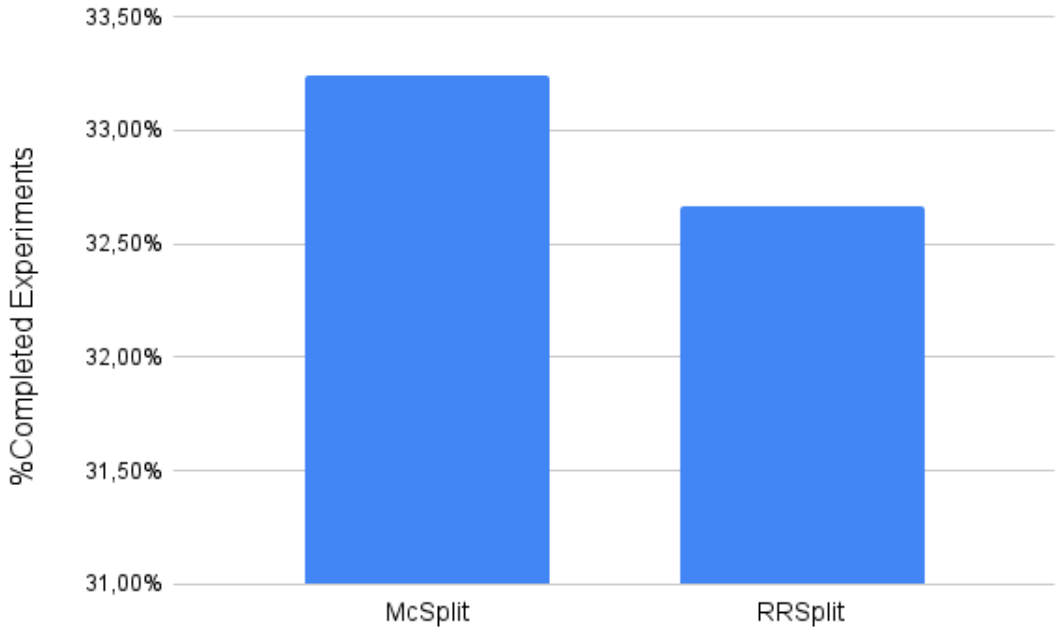

Figura 51: Percentuale di esperimenti, su GPU, completati

Come è già possibile notare dal grafico 50, RRSplit riesce a completare un numero leggermente inferiore di esperimenti rispetto a McSplit, 343 per RRSplit e 347 per McSplit, per essere precisi, su 1050 totali, facendo sì che le percentuali ad esse relative si attestassero sul 32-33% per entrambi i casi, come si evince dalla figura 51.

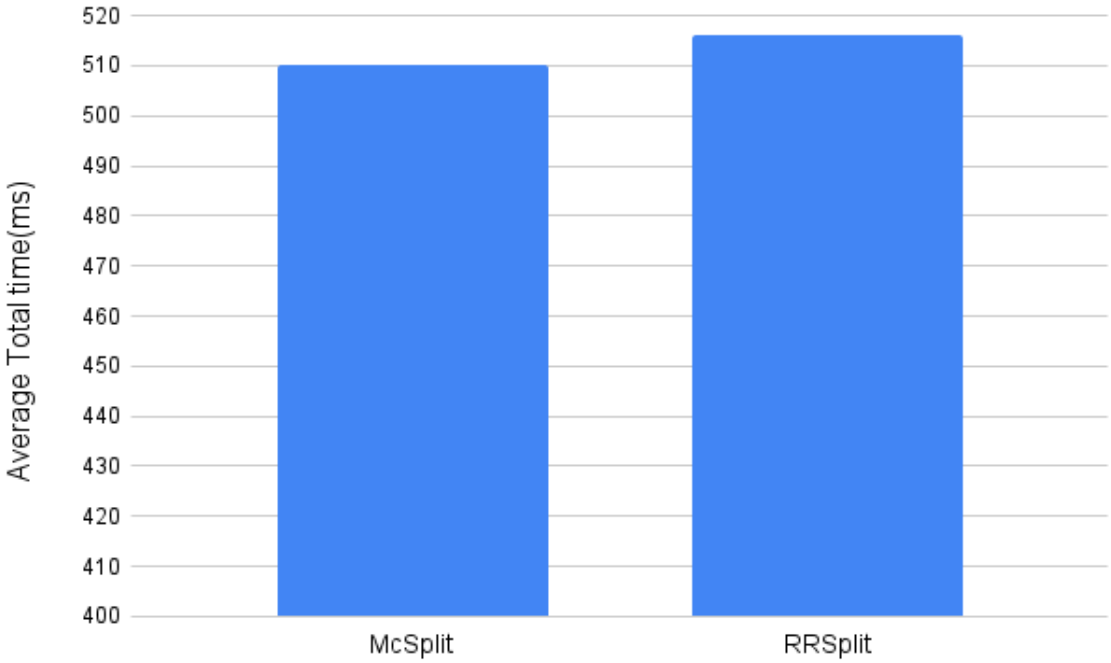

Figura 52: Tempi medi totali per run completate, su GPU

Inoltre, è possibile anche constatare che, se su CPU le migliorie presenti su RRSplit abbiano apportato dei benefici, permettendo di ottenere delle prestazioni migliori a tutto tondo rispetto alla controparte McSplit, su GPU tal cosa non si verifica, come si può evincere dal grafico 52. Prendendo infatti le run che sono andate a buon fine, è possibile riscontrare un peggioramento, seppur estremamante lieve, dei tempi di risoluzione.
Da questi risultati sembra quindi emergere l'idea che RRSplit, seppur molto valido su CPU, riuscendo anche ad ottenere grandi benefici dall'implementazione multi-thread, non sembri particolarmente adatto in un ambiente GPU. Ciò potrebbe essere dato proprio da come l'ambiente CUDA/GPU funzioni, portando tutte le ottimizzazioni, funzionanti su CPU, ad essere un dispendio fin troppo esoso su GPU, sulla quale gran parte degli sforzi e del tempo possono essere devoluti alla sincronizzazione dei thread nei vari blocchi, così come anche dei blocchi stessi.

9 Conclusioni

Prendendo in considerazione i risultati ottenuti nel capitolo precedente, verranno qui esposti eventuali sviluppi futuri, ma che in questa tesi non hanno trovato spazio di realizzazione.

9.1 Implementazione CPU di RRSplit

È stato possibile verificare che, utilizzando gli schemi di parallelismo adottati da McCreesh su RRSplit, esso abbia apportato dei miglioramenti consistenti, avendo esso delle prestazioni migliori sia rispetto a McSplit parallelo, sia rispetto a RRSplit seriale.
Tuttavia, come già accennato in precedenza, tale algoritmo presenta ancora dei bug che non sono ancora stati risolti, e che, nel caso in cui lo fossero, permetterebbe all'algoritmo di RRSplit, seriale e parallelo, di essere più veloce.
Dovessero tali bug venire risolti, sarebbero necessari nuovi esperimenti, con la consapevolezza però che il divario di prestazioni ottenuto tra RRSPlit e McSplit aumenterebbe ulteriormente.

9.2 Implementazione GPU di RRSplit

Si è potuto evincere che, nell'ambiente GPU, tale algoritmo risulti avere prestazioni peggiori rispetto alla controparte McSplit, mettendo in luce come esso non segua la stessa tendenza mostrata nel caso CPU.
Tuttavia, per poter arrivare alla conclusione definitiva che RRSplit_GPU risulti essere sempre e consistentemente peggiore di McSplit_GPU, ulteriori esperimenti, condotti su grafi con diverse caratteristiche rispetto a quelle enunciate nella sezione 8.1, devono essere condotti.
Un'altra direzione che si potrebbe prendere è, invece, quella di modificare il framework che è stato utilizzato, per vedere se ciò possa migliorare le prestazioni.

Ringraziamenti

Voglio ringraziare i miei relatori per avermi guidato in questo percorso di tesi, per aver dimostrato una disponibilità ed una pazienza fuori dal comune, permettendomi di vivere questo momento importante, comunque difficile e non privo di insidie, con molta calma e serenità.
Voglio qui infine dar spazio ad una riflessione: questo traguardo, così come tanti altri che ho raggiunto, o che raggiungerò nel mio futuro, immediato o meno che esso sia, non è merito esclusivamente mio. Al contrario, molti vi hanno apportato un grosso contributo, non solo per permettermi di giungere fino a questo punto, ma anche perchè, grazie al sostegno che ho ricevuto nel mio lungo cammino, chiamasi vita, se ora io sono la persona che oggi conoscete, è per merito vostro.
Durante il periodo universitario ho avuto modo di creare amicizie fantastiche, e con esse creare dei momenti indimenticabili e dei legami molto forti. Non so come descrivere la fortuna che ho avuto nell'incontrarvi, però posso dire una cosa: grazie di tutto cuore.
Una buona fetta dei meriti va a voi, che mi avete accolto nel vostro gruppo, permettendomi di essere me stesso, e al contempo stesso dandomi modo di maturare, pur rimanendo il solito scemo, ma non solo. Così nei momenti belli, siete stati presenti anche in quelle giornate nelle quali ero un po' giù, sempre stando presenti e dandomi modo di superare tutte le difficoltà incontrate nel mio percorso.
Se ora mi trovo qui, è anche grazie a voi, non dimenticatelo mai.
Va tuttavia messa in evidenza la grande importanza, però, che la mia famiglia ha avuto nel raggiungimento non solo di questo traguardo, ma anche di questa mia forma, chiamiamola così, ovvero dell'educazione che ho da loro ricevuto per poter essere, prima di tutto, una buona persona con dei saldi principi morali. Siete stati voi, infatti, i primi che mi hanno impartito le lezioni più importanti della vita, estremamente preziose e che tuttora caratterizzano la mia persona, quali l'umiltà e la condivisione, così come anche il rispetto per gli altri e la costanza.
Perchè è inutile girarci attorno: non sarei potuto diventare la persona che ora conoscete se non avessi avuto una famiglia con dei sani valori morali, e di questo vi ringrazio molto.
Rivolgo questo ulteriore ringraziamento a te, mamma, che sei sempre stata la prima a gioire per ogni mio singolo successo, piccolo o grande che esso sia stato, così come anche la prima alla quale ho rivolto i miei dubbi, i miei crucci, sempre ricevendo consigli, e anche ammonimenti, di modo da guidarmi nel modo migliore possibile, affinchè potessi realizzarmi, nonostante la distanza che, in questi anni di università, ci ha separati.
Voglio infine ringraziare anche le amicizie che ho avuto modo di creare al di là del contesto universitario, primo tra questi un amico di vecchia data, Francesco, che ho avuto modo di conoscere a nove anni, e che ormai è per me un fratello, avendone praticamente combinate di tutte i colori.
A tutti voi, grazie.

Indice delle immagini

Indice degli algoritmi

Riferimenti bibliografici

[1] John W Raymond e Peter Willett. "Maximum common subgraph isomorphism algorithms for the matching of chemical structures". In: *Journal of computer-aided molecular design* 16.7 (2002), pp. 521–533.

[2] Péter Englert e Péter Kovács. "Efficient heuristics for maximum common substructure search". In: *Journal of chemical information and modeling* 55.5 (2015), pp. 941–955.

[3] Imen Boukhris, Zied Elouedi, Thomas Fober, Marco Mernberger e Eyke Hüllermeier. "Similarity analysis of protein binding sites: A generalization of the maximum common subgraph measure based on quasi-clique detection". In: *2009 Ninth International Conference on Intelligent Systems Design and Applications*. IEEE. 2009, pp. 1245–1250.

[4] Hans-Christian Ehrlich e Matthias Rarey. "Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review". In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 1.1 (2011), pp. 68–79.

[5] Yao Lu, Horst Bunke e Cheng-Lin Liu. "An algorithm for maximum common subgraph of planar triangulation graphs". In: *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer. 2013, pp. 162–171.

[6] Younghee Park, Douglas S Reeves e Mark Stamp. "Deriving common malware behavior through graph clustering". In: *computers & security* 39.419 (2013), e430.

[7] Jon M Kleinberg. "Authoritative sources in a hyperlinked environment". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 604–632.

[8] Andrew Dalke e Janna Hastings. "FMCS: a novel algorithm for the multiple MCS problem". In: *Journal of cheminformatics* 5.Suppl 1 (2013), O6.

[9] Debin Gao, Michael K Reiter, Dawn Song et al. "Automatically finding semantic differences in binary programs [C]". In: *Proc of the Int Conf on Information and Communications Security*. 2008, pp. 238–255.

[10] Jochem H. Rutgers, Pascal T. Wolkotte, Philip K.F. Hölzenspies, Jan Kuper e Gerard J.M. Smit. "An Approximate Maximum Common Subgraph Algorithm for Large Digital Circuits". In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 2010, pp. 699–705. DOI: 10.1109/DSD.2010.29.

[11] Ciaran McCreesh. "Solving hard subgraph problems in parallel". Tesi di dott. University of Glasgow, 2017.

[12] Viggo Kann. "On the approximability of the maximum common subgraph problem". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 1992, pp. 375–388.

[13] Robert M Haralick e Gordon L Elliott. "Increasing tree search efficiency for constraint satisfaction problems". In: *Artificial intelligence* 14.3 (1980), pp. 263–313.

[14] James J McGregor. "Backtrack search algorithms and the maximal common subgraph problem". In: *Software: Practice and Experience* 12.1 (1982), pp. 23–34.

[15] Evgeny B Krissinel e Kim Henrick. "Common subgraph isomorphism detection by backtracking search". In: *Software: Practice and Experience* 34.6 (2004), pp. 591–607.

[16] Ciaran McCreesh, Patrick Prosser e James Trimble. "A partitioning algorithm for maximum common subgraph problems". In: (2017).

[17] Philippe Vismara e Benoıt Valery. "Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms". In: *International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences*. Springer. 2008, pp. 358–368.

[18] Samba Ndojh Ndiaye e Christine Solnon. "CP models for maximum common subgraph problems". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2011, pp. 637–644.

[19] Kaiqiang Yu, Kaixin Wang, Cheng Long, Laks Lakshmanan e Reynold Cheng. "Fast Maximum Common Subgraph Search: A Redundancy-Reduced Backtracking Approach". In: *arXiv preprint arXiv:2502.11557* (2025).

[20] Yanli Liu, Chu-Min Li, Hua Jiang e Kun He. "A learning based branch and bound for maximum common subgraph related problems". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 03. 2020, pp. 2392–2399.

[21] Jianrong Zhou, Kun He, Jiongzhi Zheng, Chu-Min Li e Yanli Liu. "A strengthened branch and bound algorithm for the maximum common (connected) subgraph problem". In: *arXiv preprint arXiv:2201.06252* (2022).

[22] Yanli Liu, Jiming Zhao, Chu-Min Li, Hua Jiang e Kun He. "Hybrid learning with new value function for the maximum common subgraph problem". In: *arXiv preprint arXiv:2208.08620* (2022).

[23] Gabriele Mosca. "Solving the maximum common subgraph problem on many-cores architectures". Tesi di dott. Politecnico di Torino, 2019.

[24] P. Foggia, C. Sansone e M. Vento. "A Database of Graphs for Isomorphism and Sub-Graph Isomorphism Benchmarking". In: -. 1 Gen. 2001, pp. 176–187.

[25] M. De Santo, P. Foggia, C. Sansone e M. Vento. "A large database of graphs and its use for benchmarking graph isomorphism algorithms". In: *Pattern Recogn. Lett.* 24.8 (mag. 2003), pp. 1067–1079. ISSN: 0167-8655. DOI: 10.1016/S0167-8655(02)00253-2. URL: http://dx.doi.org/10.1016/S0167-8655(02)00253-2.