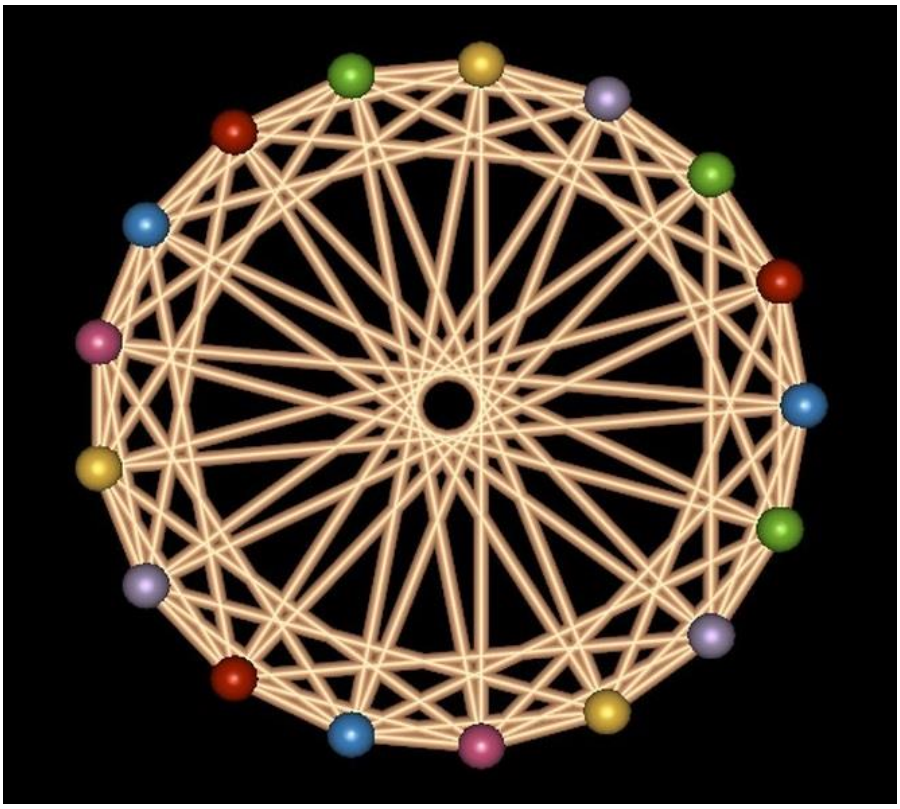
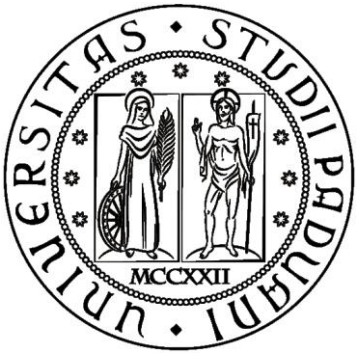
UNIVERSITÁ DEGLI STUDI DI PADOVA

ALGORITMI EURISTICI PER LA RISOLUZIONE DI PROBLEMI DI GRAPH COLORING



UNIVERSITÁ DEGLI STUDI DI PADOVA



FACOLTÁ DI INGEGNERIA
CORSO DI LAUREA MAGISTRALE
IN
INGEGNERIA INFORMATICA

LAUREANDO:

Massimo Crivellari

RELATORE:

Prof. Michele Monaci

ANNO ACCADEMICO 2012 – 2013

INDICE

CAPITOLO 1

CAPITOLO 2

CAPITOLO 3

CAPITOLO 4

CAPITOLO 1

INTRODUZIONE

1.1 GRAPH COLORING PROBLEM

Il problema della colorazione dei grafi nasce come congettura nel 1852, mentre uno studente dell'Università di Londra, Francis Guthrie, cercava di colorare una cartina delle contee della capitale inglese utilizzando, come di consueto, colori differenti per contee confinanti. Accorgendosi che solo quattro colori bastavano al suo scopo e non riuscendo ad individuare mappe per le quali un tale numero non fosse sufficiente, si propose di dimostrare la sua congettura, successivamente divenuta nota con il nome di congettura o teorema dei quattro colori. Molti illustri matematici vennero coinvolti nel tentativo di dimostrarla e dedicarono anni della loro vita alla risoluzione del problema. Molte furono le dimostrazioni proposte, alcune contenenti errori, altre contestate in quanto facenti uso del calcolatore; solo infine negli anni 2000 si può ritenere definitivamente risolto il problema della dimostrazione del teorema.

I numerosi sforzi compiuti per individuare una dimostrazione portarono a grandi risultati nel campo dello studio del problema della colorazione dei grafi e più in generale in altre aree della teoria dei grafi.

Oggigiorno sono numerosi i problemi che possono essere risolti utilizzando algoritmi e strategie propri del problema della colorazione dei vertici di un grafo:

essi spaziano dalla colorazione di mappe, come si è già potuto vedere, a problemi nel settore dei trasporti; dall'organizzazione degli orari all'interno di un'università, al problema di assegnamento di radio frequenze alle diverse stazioni emittenti in modo tale da evitare interferenze di segnale.

Essi sono tutti caratterizzati dalla condivisione di una struttura comune: una risorsa è a disposizione di più utenti, alcuni dei quali, però, non possono accedervi in modo simultaneo; il problema richiede di raggruppare gli utenti che utilizzano la risorsa nello stesso istante in modo tale da minimizzare il numero di copie della risorsa che devono essere disponibili contemporaneamente.

Formalmente, dato un grafo non orientato $G = (V, E)$, con V insieme dei vertici ed E insieme degli archi di G, con cardinalità rispettivamente n ed m, il problema della colorazione dei vertici richiede che sia assegnato un colore a ciascuno dei nodi del grafo, in modo che due vertici adiacenti vengano sempre colorati diversamente e che il numero di colori sia minimizzato.

1.2 OBIETTIVI

Il problema della colorazione dei nodi di un grafo è un ben noto problema NP-difficile e nonostante abbia ricevuto grandi attenzioni nel tempo vista la sua importanza in molteplici applicazioni pratiche, gli algoritmi esistenti in letteratura sono in grado di risolvere in maniera esatta solo istanze di piccola taglia. Tuttavia, molto spesso, le applicazioni del mondo reale sono rappresentabili attraverso grafi con molti più nodi e archi. Ecco quindi l'importanza degli algoritmi euristici e meta-euristici, che benché non esibiscano comunque una soluzione ottima del problema, ne danno un buon bound e sembrano essere molto promettenti nel trattare grafi con centinaia e migliaia di vertici [13].

Lo scopo di questa tesi è quello di realizzare e successivamente analizzare algoritmi euristici e/o strategie che possano in qualche modo apportare miglioramenti a quelli già noti, in termini di risultati conseguiti nell'individuare una colorazione ottima per i grafi proposti dalla letteratura.

Il problema risolto è un problema di programmazione mista-intera (Mixed Integer Programming – MIP) in cui i valori assunti dalle variabili sono confinati al campo intero. Più nello specifico il problema richiederà di risolvere uno 0-1 MIP dato che le variabili saranno tutte binarie. La programmazione intera-mista è una delle più importanti tecniche per risolvere problemi d'ottimizzazione complessi e offre un potente strumento per modellare e risolvere un'ampia varietà di problemi.

Un MIP problem è definito da un insieme di variabili (x), un insieme di vincoli lineari sulle variabili definite ($Ax = b$), un insieme di vincoli di interezza che specifica che alcune variabili devono assumere valori interi e una funzione obiettivo lineare da ottimizzare ($min\ c^T x$), come di seguito descritto:

$$\begin{aligned} \min\ & c^T x \\ & Ax = b \\ & x \geq 0\ intere \end{aligned}$$

Generalmente questi problemi sono risolti attraverso algoritmi branch-and-bound o branch-and-cut. Un tale approccio si basa sull'esplorazione di un albero generato da rilassamenti continui del modello MIP originale, in cui, in corrispondenza a ciascun nodo dell'albero, lo spazio delle possibili soluzioni è suddiviso in due sottospazi disgiunti attraverso l'inserimento di vincoli complementari sui limiti di esistenza di una particolare variabile. Questa tecnica è particolarmente efficace quando i rilassamenti continui del problema sono una buona approssimazione del guscio convesso delle soluzioni ammissibili attorno alla soluzione ottima, o, in alternativa, quando il rilassamento può essere ristretto aggiungendo piani di taglio per ottenere questa proprietà. Molto lavoro è stato fatto per rendere le implementazioni commerciali robuste rispetto al problema della dimensione dell'istanza, ma alcuni modelli rimangono comunque molto difficili da risolvere all'ottimo. In tali casi, e a meno che non esistano altre tecniche ad hoc di programmazione a vincoli che si adattino bene al problema considerato, ci si deve accontentare di soluzioni fattibili di buona qualità. In alcuni casi, però, può essere difficile addirittura ottenere soluzioni non ottime in tempo utile.

In risposta a questi problemi si utilizzano altre tecniche come quelle di ricerca locale. Infatti varie forme di local search, operando su soluzioni individuali (simulated annealing e tabu search) o su una popolazione di soluzioni (algoritmi

genetici, scatter search e path relinking), sono note per fornire soluzioni ammissibili di eccellente qualità e in tempi veloci per molti problemi. Questo suggerisce che applicare i concetti della ricerca locale (intorno, intensificazione, diversificazione) può essere utile quando si devono affrontare modelli di programmazione mista-intera particolarmente complessi.

Questa è stata la strada seguita nell'implementazione degli algoritmi di questa tesi. Per integrare la strategia di local search con un modello MIP sono tre le domande cui bisogna dare risposta:

- com'è definito l'intorno di ricerca?
- com'è attuato il concetto di intensificazione, cioè com'è esplorato l'intorno precedentemente definito?
- come viene garantita la diversificazione nella ricerca della soluzione?

L'intorno in cui effettuare la ricerca verrà definito attraverso la nozione di distanza di Hamming tra la soluzione corrente e la nuova soluzione da individuare; lo spazio di ricerca verrà esplorato con una semplice enumerazione delle soluzioni e quando possibile attraverso RINS, cioè l'algoritmo euristico interno al solver utilizzato, che si basa sulla creazione e risoluzione di un sub-MIP, attraverso la definizione ed esplorazione di un altro intorno, definito internamente ad esso e fondato sull'istanziamento di alcune precise variabili; infine la diversificazione è garantita dal meccanismo di recentering che consiste nell'aggiornamento della miglior soluzione correntemente individuata con la nuova soluzione migliorante restituita dall'ottimizzatore [4].

Più nello specifico, con questo lavoro, si vogliono indagare gli effetti del principio di prossimità tra soluzioni, il quale permette di limitare lo spazio di ricerca a intorni della soluzione ammissibile di partenza, spostandosi, nel processo di ottimizzazione, da una soluzione ammissibile ad un'altra migliore e vicina alla precedente. L'obiettivo è anche quello di migliorare il comportamento di un solver utilizzato come black-box nella risoluzione del problema e comprendere meglio le conseguenze della scelta di modificare i vincoli del modello per la riduzione dello spazio da esplorare, piuttosto che scegliere di apportare modifiche alla funzione obiettivo con lo scopo di semplificare l'esplorazione.

CAPITOLO 2

CENNI STORICI

2.1 IL TEOREMA DEI QUATTRO COLORI

La teoria su cui si fonda la colorazione dei grafi trae origine dal problema di dover colorare con differenti colori i diversi paesi presenti sulle cartine geografiche. La colorazione degli stati richiede l'utilizzo di colori diversi per paesi confinanti, dimodoché possano essere distinguibili facilmente i loro confini.

Se ogni stato viene denotato da un punto nel piano e se ogni coppia di punti corrispondenti a paesi con un confine in comune viene collegata da una linea, ciò che si ottiene è un grafo planare.

Infatti i primi risultati concernenti il problema della colorazione dei grafi riguardano, quasi esclusivamente, la classe dei grafi planari, essendo questi la rappresentazione matematica delle mappe politiche sulle quali, per l'appunto, si concentrarono i primi passi in questo senso.

FIGURA 2.1 – FRANCIS GUTHRIE

L'origine del problema della colorazione dei grafi può essere fatta risalire al 1852, anno in cui Francis Guthrie postulò la nota congettura dei quattro colori.

FIGURA 2.1 – CONTEE INGLESI

Nel 1852, Francis Guthrie, studente in legge all'Università di Londra, mentre cercava di colorare la cartina delle contee britanniche, si accorse che quattro colori erano sufficienti allo scopo. Il fatto di non riuscire a trovare una carta che richiedesse necessariamente più di quattro colori, spinse Guthrie a chiedersi se fosse vero che ogni mappa potesse essere colorata utilizzando solo quattro colori in modo che stati adiacenti avessero colori distinti. Tentò quindi di dimostrare questa sua congettura e poiché suo fratello, Frederick Guthrie, era uno studente di Augustus De Morgan, gli chiese di comunicare i dettagli del problema al professore (che precedentemente insegnò anche a Francis).

De Morgan, compiaciuto dei risultati e allo stesso tempo incapace di dare una risposta alla questione sollevata, scrisse una lettera al matematico e fisico William Rowan Hamilton a Dublino, nella speranza di coinvolgerlo nella risoluzione del problema della colorazione della mappa, ma subito gli rispose che probabilmente non si sarebbe interessato al suo problema molto presto [15].

De Morgan portò il problema all'attenzione di altri matematici e propose la congettura anche alla London Mathematical Society; moltissimi matematici dedicarono anni della loro vita nel tentativo di dimostrare il teorema e, nonostante nessuno riuscisse a provare la congettura, tali tentativi portarono a molti risultati inerenti il problema della colorazione dei grafi e contribuirono allo sviluppo di altre aree della teoria dei grafi.

Arthur Cayley risollevò il problema al meeting della London Mathematical Society nel 1879; è dello stesso anno una sua pubblicazione alla Royal Geographical Society in cui spiegava dove si riscontravano le difficoltà nei tentativi di arrivare ad una dimostrazione.

La prima dimostrazione si deve ad Alfred Kempe, sempre nel 1879, e per molto tempo il problema dei quattro colori fu considerato risolto.

Nel 1890 il teorema dei quattro colori tornò ad essere solo la congettura dei quattro colori, quando Percy John Heawood individuò un errore nel lavoro di Kempe confutandone la dimostrazione.

Negli anni a seguire molteplici sono stati i risultati ottenuti e i concetti introdotti nella teoria dei grafi, ma bisognerà attendere il 1977 quando Kenneth Appel e Wolfgang Haken, due matematici dell'Università dell'Illinois, pubblicarono la loro dimostrazione del teorema dei quattro colori. Tuttavia la dimostrazione fu accolta con molto scetticismo e grandi polemiche dal mondo scientifico in quanto si trattava di una dimostrazione rivoluzionaria, ottenuta grazie all'ausilio del computer, e non verificabile manualmente.

Del 1997 è invece un miglioramento della dimostrazione che riduce significativamente l'utilizzo del computer ed è dovuta a Neil Robertson, Daniel P. Sanders, Paul Seymour e Robin Thomas, ma in ogni caso non è una dimostrazione tradizionale, perché contiene passi che non possono essere verificati manualmente.

Infine, nel 2000, un matematico indiano, Ashay Dharwadker, propose una nuova ed elegante dimostrazione del teorema che richiede l'utilizzo della teoria dei gruppi e non più l'utilizzo del calcolatore [11, 18, 19, 20].

CAPITOLO 3

SOFTWARE UTILIZZATO

3.1 SCELTE E MOTIVAZIONI

Per la risoluzione dei modelli matematici del problema della colorazione dei vertici di un grafo, si è scelto di utilizzare uno strumento dell'azienda IBM di riconosciute capacità di ottimizzazione e tra i migliori in commercio attualmente, e cioè l'applicativo software IBM ILOG CPLEX Optimization Studio nella sua versione 12.4.

Poiché si è scelto di lavorare in un ambiente Microsoft su cui è installato il sistema operativo Windows 7 si è valutato di utilizzare lo strumento di sviluppo Microsoft Visual Studio 2010, la penultima versione dell'IDE di Microsoft Visual Studio. La scelta è ricaduta su questo prodotto anche per la semplicità di interfacciamento con l'altro importante software utilizzato per la realizzazione di questa tesi e cioè CPLEX: è bastato infatti impostare alcuni semplici riferimenti alle librerie rese disponibili dal software dell'IBM per usufruire appieno delle sue funzionalità all'interno del codice realizzato.

Il linguaggio prescelto per la scrittura del codice del programma è stato il linguaggio C, supportato da Visual Studio 2010 e contemporaneamente da CPLEX

attraverso il pacchetto di funzioni messe a disposizione dalla libreria CPLEX Callable Library, che ne permette la gestione delle funzionalità all'interno di altri software realizzati dall'utente.

3.2 MICROSOFT VISUAL STUDIO 2010

FIGURA 3.1 – LOGO MICROSOFT VISUAL STUDIO 2010

Visual Studio è un insieme completo di strumenti di sviluppo per la compilazione di applicazioni Web ASP.NET, Servizi Web XML, applicazioni desktop e applicazioni per dispositivi mobili. Visual Basic, Visual C# e Visual C++ utilizzano lo stesso ambiente di sviluppo integrato (Integrated Development Environment – IDE) che consente la condivisione degli strumenti e semplifica la creazione di soluzioni con linguaggi misti; attualmente, Visual Studio 2010, supporta diversi tipi di linguaggio, quali C, C++, C#, F#, Visual Basic, .NET e ASP.NET. Questi linguaggi sfruttano inoltre le funzionalità di .NET Framework, che fornisce l'accesso a tecnologie chiave in grado di semplificare lo sviluppo di applicazioni Web ASP e Servizi Web XML. Microsoft Visual C++ è l'implementazione Microsoft del compilatore C e C++ e dei correlati servizi di linguaggio e comprende strumenti specifici per l'integrazione con l'IDE di Visual Studio. Quindi esso permette la compilazione sia in modalità C che in modalità C++ e per il linguaggio C, linguaggio utilizzato per lo sviluppo del programma argomento di questa tesi, segue le indicazioni dello standard ISO C e in particolare le specifiche C99 in supplemento alle specifiche Microsoft aggiunte in forma di librerie [17].

3.3 IBM ILOG CPLEX OPTIMIZATION STUDIO 12.4

L'IBM ILOG CPLEX Optimization Studio (o più semplicemente CPLEX) è un software commerciale per la soluzione di alcune classi di problemi di ottimizzazione, tra cui la PL e la PLI.

CPLEX deve il suo nome al metodo del simplesso (simplex method) e al linguaggio in cui è implementato, cioè il linguaggio C, anche se oggi comprende algoritmi addizionali nel campo della programmazione matematica ed offre interfacce verso altri ambienti e linguaggi.

Originariamente sviluppato da Robert E. Bixby, è stato commercializzato a partire dal 1988 dalla CPLEX Optimization Inc., acquisita da ILOG nel 1997. ILOG, riconosciuta azienda di software internazionale che si occupa principalmente dello sviluppo, commercializzazione, vendita e supporto di BRMS (Business Rule Management Systems), componenti software di ottimizzazione e visualizzazione, in aggiunta ad applicazioni di supply chain e decision management, è passata a sua volta sotto il controllo dell'IBM nel gennaio 2009. Questo coincise con il rilascio di nuove release dei prodotti ILOG, ora brandizzati IBM ILOG.

FIGURA 3.2 – LOGO IBM ILOG

Il pacchetto IBM ILOG CPLEX Optimization Studio [9], nella sua versione 12.4 utilizzata per questo progetto, è costituito da:

- l'ottimizzatore matematico CPLEX;
- l'ottimizzatore per il constraint programming IBM ILOG CPLEX;
- il linguaggio di programmazione OPL (Optimization Programming Language);
- un ambiente di sviluppo integrato.

3.3.1 TIPOLOGIE DI PROBLEMI RISOLTI DA CPLEX

CPLEX, che continua quindi ad essere sviluppato attivamente sotto IBM [8], rappresenta una delle più efficienti applicazioni software, tra quelle che si possono trovare in commercio oggigiorno, creato per la risoluzione di problemi di ottimizzazione lineare (Linear Problems – LP) della forma:

Maximize (or Minimize) $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$

subject to
$$a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n \sim b_1$$
$$a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n \sim b_2$$
$$\ldots$$
$$a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n \sim b_m$$

con i seguenti limiti
$$l_1 \leq x_1 \leq u_1$$
$$\ldots$$
$$l_n \leq x_n \leq u_n$$

dove ~ può essere $\leq, \geq$ o $=$, e i limiti superiori u_i e i limiti inferiori l_i possono essere positivi infiniti, negativi infiniti, un numero reale qualsiasi. Si consideri la possibilità di eventuali vincoli aggiuntivi di interezza per le variabili nel caso MIP.

I dati passati come input per questi problemi sono:

coefficienti della funzione obiettivo $c_1, c_2, \ldots, c_n$

coefficienti dei vincoli
$$a_{11}, a_{21}, \ldots, a_{n1}$$
$$\ldots$$
$$a_{m1}, a_{m2}, \ldots, a_{mn}$$

termini noti $b_1, b_2, \ldots, b_m$

limiti superiori e inferiori $u_1, u_2, \ldots, u_n$ e $l_1, l_2, \ldots, l_n$

La soluzione ottima che CPLEX calcola e restituisce è:

$$x_1, x_2, \ldots, x_n$$

Più precisamente questo software permette all'utente, attraverso l'algoritmo del simplesso (nelle sue varianti primale o duale), attraverso metodi di punto interno, o ancora, attraverso procedure basate sull'enumerazione implicita di tipo branch and bound e altre tecniche particolari, di risolvere problemi di programmazione lineare intera, anche di notevoli dimensioni, e diverse estensioni di questi problemi, cioè:

- problemi di flusso in una rete, un caso speciale di programmazione lineare che CPLEX può risolvere molto velocemente sfruttando la struttura del problema;
- problemi di programmazione quadratica (Quadratic Programming – QP), in cui la funzione obiettivo comprende anche termini quadratici;
- problemi di programmazione con vincoli quadratici (Quadratic Constrained Programming – QCP), che includono quei problemi che tra i vincoli esibiscono termini quadratici; infatti CPLEX può risolvere problemi utilizzando tecniche di Second Order Cone Programming (SOCP);
- problemi di programmazione mista-intera (Mixed Integer Programming – MIP), in cui i valori che possono assumere alcune o tutte le variabili, nella funzione obiettivo, sono ristretti al campo intero.

3.3.2 COMPONENTI DI CPLEX

CPLEX, per venire incontro alle differenti esigenze degli utenti, mette a disposizione diverse componenti e, di conseguenza, diverse modalità d'utilizzo ed interazione:

- CPLEX Interactive Optimizer è un programma esecutivo con il quale è possibile interagire tramite riga di comando. Attraverso l'applicazione si può

descrivere il modello del problema, che può essere letto interattivamente, o lo si può importare e leggere da file esterni predisposti in determinati formati standard. È possibile configurare il risolutore tramite il settaggio di opportuni parametri, quindi richiedere l'ottimizzazione del modello specificato, ed infine ottenere la restituzione della soluzione interattivamente o attraverso la scrittura su file;

- Concert Technology è un insieme di librerie per i linguaggi C++, Java e .NET, linkabili in modo dinamico, che mettono a disposizione un'API che include strumenti di modellazione che permettono al programmatore di rendere le funzionalità di CPLEX Optimizer parte integrante delle applicazioni scritte in C++, Java o .NET;

- CPLEX Callable Library è una libreria di funzioni scritta utilizzando il linguaggio C che permette al programmatore di integrare CPLEX Optimizer, comandarlo e gestirne le funzionalità, in applicazioni scritte in C, Visual Basic, FORTRAN, o qualsiasi altro linguaggio che sia in grado di richiamare funzioni C;

- Python API per CPLEX è una interfaccia di programmazione multifunzione per programmatori Python che supporta tutti i diversi aspetti dell'ottimizzazione di CPLEX;

- Il connettore per The MathWorks MATLAB che permette ad un utente di definire problemi di ottimizzazione e risolverli all'interno dell'ambiente di MATLAB sfruttando il MATLAB Toolbox o una classe di CPLEX nel linguaggio di MATLAB.

Tranne la prima modalità d'uso, le altre, in sintesi, sono una serie di librerie, scritte in linguaggi differenti, che permettono di interfacciarsi col risolutore all'interno di un altro software più ampio, lasciando ad esso il controllo del processo ed eliminando la necessità di una interazione real-time da parte di un utente.

Dato che la scelta è stata quella di sviluppare un'applicazione grazie all'ausilio dello strumento di sviluppo Microsoft Visual Studio 2010 e attraverso l'utilizzo del linguaggio C, la modalità d'uso prescelta è stata quella che prevede l'integrazione di CPLEX e delle sue funzionalità tramite le librerie del CPLEX Callable Library, che permettono al programmatore di scrivere il proprio applicativo in modo indipendente e di richiamare le funzioni di ottimizzazione di CPLEX come una black-box nel momento in cui servono e senza l'esigenza di una stretta interazione in tempo reale con il risolutore, come sarebbe avvenuto nel caso in cui si fosse optato di ottimizzare il modello del problema direttamente tramite CPLEX Interactive Optimizer [8].

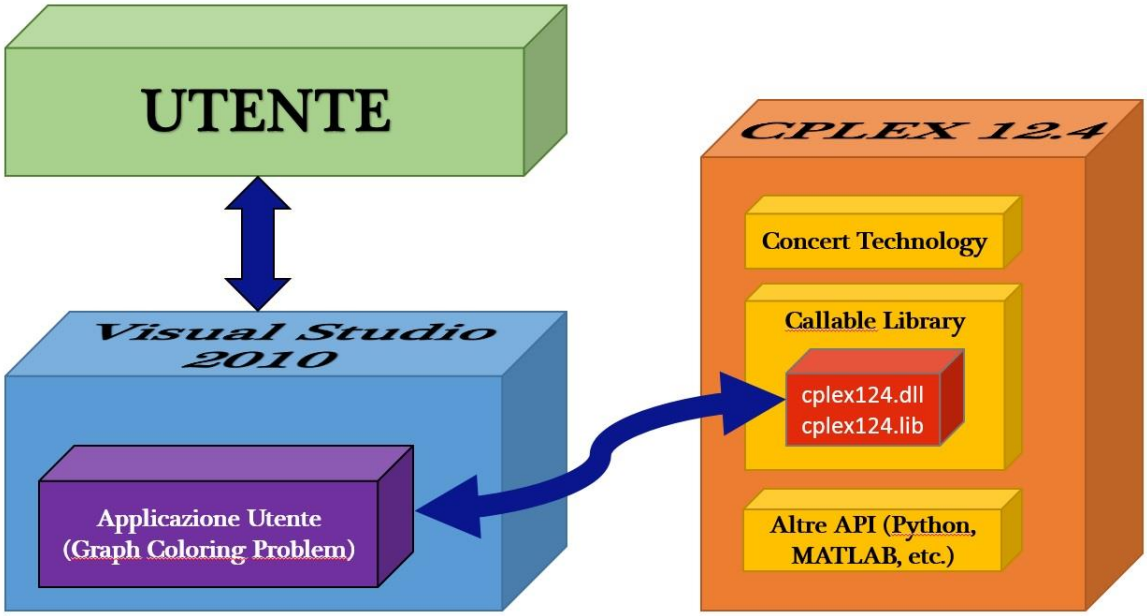


FIGURA 3.3 – INTERFACCIAMENTO DI CPLEX CON L'APPLICAZIONE UTENTE

3.3.3 DIVERSI OTTIMIZZATORI IN CPLEX

CPLEX dispone al suo interno di una batteria di risolutori, esatti ed euristici: le impostazioni di default hanno come conseguenza la chiamata dell'ottimizzatore che è più appropriato per la classe di problema che si vuole andare a risolvere. Ad ogni modo è possibile scegliere un diverso ottimizzatore, e cioè quello che più si adatta al particolare scopo che si vuole raggiungere.

Un problema di LP o QP può essere risolto chiamando in causa uno qualsiasi dei seguenti ottimizzatori di CPLEX: ottimizzatore che fa uso del simplesso duale o del simplesso primale, quello che utilizza metodi di barriera e, nel caso si riuscisse ad estrapolare dal problema una ben precisa struttura a rete, l'ottimizzatore di rete.

I modelli di rete puri sono obbligatoriamente risolti dal relativo ottimizzatore di rete.

I modelli QCP, includendo il caso speciale dei modelli SOCP, sono risolti con l'utilizzo dell'ottimizzatore che si basa su metodi di barriera.

I modelli MIP sono risolti tutti dal corrispettivo ottimizzatore, il quale, a sua volta, durante l'elaborazione, può invocare ciascuno degli ottimizzatori previsti per i problemi di LP o QP.

La tabella 3.1 riassume i vari tipi di problemi e i relativi ottimizzatori che possono essere utilizzati [8].

	LP	Network	QP	QCP	MIP
Dual Optimizer	X		X		
Primal Optimizer	X		X		
Barrier Optimizer	X		X	X	
Mixed Integer Optimizer					X
Network Optimizer	Nota	X	Nota		
Nota: dal problema deve essere possibile estrapolare una struttura a rete.					

TABELLA 3.1 – OTTIMIZZATORI

3.3.4 FORMATI DI FILE SUPPORTATI

Molti sono i formati riconosciuti e supportati da CPLEX, ognuno dei quali utile per un determinato scopo, come ad esempio la rappresentazione in formati diversi della soluzione trovata, la descrizione di informazioni di vario genere circa l'evoluzione del programma nella generazione della soluzione, l'elencazione di

parametri e delle loro impostazioni che devono essere rispettati e che indicano come deve essere eseguita la ricerca della soluzione, la rappresentazione in differenti formati del modello matematico del problema che si vuole risolvere, eccetera.

Di seguito saranno descritti solamente le tipologie di file utilizzati dall'applicativo sviluppato, e cioè file con estensione .lp, per la descrizione del modello del problema, file con estensione .sol, per la descrizione della soluzione trovata e file con estensione .prm per l'impostazione di parametri non di default attraverso la scrittura e lettura di file anziché attraverso la scrittura di istruzioni nel codice del programma [3].

3.3.4.1 FILE CON ESTENSIONE LP

CPLEX fornisce diverse opzioni per l'inserimento dei dati di input del problema. Quando si deve interagire con l'Interactive Optimizer, la maggior parte degli utenti, utilizza file formattati per l'inserimento dell'input. La tipologia di file propria di CPLEX è il file con estensione LP, un formato row-oriented che molti utenti considerano più naturale, in quanto simile all'usuale formulazione dei problemi LP e MIP, rispetto ad altri formati, come quello, per esempio, dei file MPS (Mathematical Programming System), in cui i dati vengono inseriti in un formato column-oriented molto meno intuitivo e pratico da utilizzare.

Naturalmente, se ci si orienta verso un'interazione con CPLEX di tipo interattivo è possibile passare i dati di input attraverso una modalità in linea e in tempo reale, usando però sempre il formato di file LP, ma questo è evidentemente possibile solo per problemi di piccola entità.

Le componenti fondamentali di un file LP sono:

- DEFINIZIONE DELLA FUNZIONE OBIETTIVO (obbligatoria): la dichiarazione del problema deve iniziare con la parola chiave

MINIMIZE

o

MAXIMIZE

che introduce la sezione della funzione obiettivo che deve essere ottimizzata; alla funzione obiettivo può essere assegnato un nome, altrimenti CPLEX assegna il nome di default *obj*.

- DEFINIZIONE DEI VINCOLI (obbligatoria): la dichiarazione dei vincoli è introdotta dalla parola chiave

SUBJECT TO

seguita dalla descrizione dei vari vincoli del problema. Ciascun vincolo deve essere inserito in una nuova riga, che inizia con il nome assegnatogli dall'utente o da quello di default *c1*, *c2*, *c3*, eccetera, attribuitogli invece da CPLEX; esso può estendersi anche su più righe e termina con l'indicazione del verso del vincolo ($\leq$, $\geq$ o $=$) e il termine noto.

- DEFINIZIONE DEI LIMITI DELLE VARIABILI (opzionale): la descrizione degli intervalli di esistenza delle diverse variabili inizia con la parola chiave

BOUNDS

seguita dalla dichiarazione, uno per riga, dei vari limiti. Se non presente, le variabili sono assunte essere non negative.

- SPECIFICA DELLE VARIABILI (opzionale): indica la natura delle variabili e quindi il campo dei valori che può assumere; è introdotta dalla parola chiave

GENERAL

o

BINARY

o

INTEGER

seguita dalla lista delle corrispettive variabili.

- TERMINAZIONE (obbligatoria): il file LP si chiude con la parola chiave

END

In figura 3.4 un esempio della formulazione di un problema da cui si può vedere come vengono utilizzate le diverse componenti di un file LP appena descritte [3].

```
Maximize
 obj: x1 + 2 x2 + 3 x3 + x4
Subject To
 c1: - x1 + x2 + x3 + 10 x4 <= 20
 c2: x1 - 3 x2 + x3 <= 30
 c3: x2 - 3.5 x4 = 0
Bounds
 0 <= x1 <= 40
 2 <= x4 <= 3
General
 x4
End
```

FIGURA 3.4 – ESEMPIO DI FILE LP

3.3.4.2 FILE CON ESTENSIONE SOL

Questo è il formato di file prodotto in output da CPLEX. È un file formattato in XML che rende possibile la visualizzazione dei file, con le soluzioni prodotte dall'ottimizzatore, direttamente nella maggior parte dei browser e il trattamento delle soluzioni da parte di tutte le applicazioni predisposte per la lettura di file XML.

Esso contiene molte informazioni relative alla soluzione del problema, alcune delle quali visibili in figura 3.5, come il valore della funzione obiettivo e il valore all'ottimo delle variabili, il numero di iterazioni effettuate da CPLEX, la qualità della soluzione, il numero di nodi dell'albero generato durante la risoluzione del problema, eccetera [3].

```
<?xml version = "1.0" standalone="yes"?>
<?xml-stylesheet href="https://www.ilog.com/products/cplex/xmlv1.0/solution.xsl" type="text/xsl"?>
<CPLEXSolution version="1.1">
 <header
   problemName="../../../examples/data/mexample.mps"
   solutionName="incumbent"
   solutionIndex="-1"
   objectiveValue="-122.5"
   solutionTypeValue="3"
   solutionTypeString="primal"
   solutionStatusValue="101"
   solutionStatusString="integer optimal solution"
   MIPNodes="0"
   MIPIterations="3"/>
 <quality
   epInt="1e-05"
   epRHS="1e-06"
   maxIntInfeas="0"
   maxPrimalInfeas="0"
   maxX="40"
   maxSlack="2"/>
 <linearConstraints>
  <constraint name="c1" index="0" slack="0"/>
  <constraint name="c2" index="1" slack="2"/>
  <constraint name="c3" index="2" slack="0"/>
 </linearConstraints>
 <variables>
  <variable name="x1" index="0" value="40"/>
  <variable name="x2" index="1" value="10.5"/>
  <variable name="x3" index="2" value="19.5"/>
  <variable name="x4" index="3" value="3"/>
 </variables>
</CPLEXSolution>
```

FIGURA 3.5 – ESEMPIO DI FILE SOL

3.3.4.3 FILE CON ESTENSIONE PRM

Per modificare i vari parametri che influiscono durante l'esecuzione dell'ottimizzazione e per impostarli a valori non di default, CPLEX fornisce la possibilità di scrivere e leggere file con estensione .prm.

Di seguito la formattazione prevista per questo tipo di file:

CPLEX Parameter File *Version number*
Parameter_name parameter_value

CPLEX legge l'intero file prima di apportare qualsiasi cambiamento alle impostazioni standard dei parametri. Dopo aver letto il file, inizialmente il Callable Library imposta tutti i parametri al loro valore di default, per modificarli solamente nel caso in cui non contenga errori, come valori mancanti o non consentiti. Non esiste una procedura per la verifica di valori duplicati per uno stesso parametro

all'interno di un file .prm e, nel caso fossero presenti, viene considerata valida l'ultima impostazione assegnata [3].

A seguire, in figura 3.6, un esempio di file .prm con alcuni parametri utilizzati.

```
CPLEX Parameter File Version 12.2.0
CPX_PARAM_EPPER                 3.45000000000000e-06
CPX_PARAM_OBJULIM               1.23456789012345e+05
CPX_PARAM_PERIND                1
CPX_PARAM_SCRIND                1
CPX_PARAM_WORKDIR               "tmp"
```

FIGURA 3.6 – ESEMPIO DI FILE PRM

CAPITOLO 4

STANDARD PER LA RAPPRESENTAZIONE DEI GRAFI

4.1 IL FORMATO DIMACS

DIMACS, acronimo per Centre for Discrete Mathematics and Theoretical Computer Science, sta ad indicare una collaborazione tra la Rutgers University e la Princeton University e le aziende di ricerca AT&T, Bell Labs, Applied Communication Sciences e NEC. Il centro si dedica sia a sviluppi teorici che ad applicazioni pratiche della matematica discreta e dell'informatica teorica, cercando di incoraggiare, stimolare e facilitare i ricercatori di questi settori e sponsorizzando conferenze e seminari.

Il DIMACS finanzia ricerche per la determinazione di algoritmi performanti in ambiti di rilevanza pratica e inerenti problemi di particolare interesse; tra questi figura anche il problema della colorazione dei grafi.

Un importante obiettivo del DIMACS è quello di semplificare gli sforzi richiesti per testare e comparare algoritmi ed euristiche fornendo un banco di prova comune per istanze e strumenti di analisi. Per facilitare questo compito, nell'ambito del vertex coloring problem, però adatto per molti altri tipi di problemi, è stato individuato un formato standard per la rappresentazione dei grafi.

Proprio con lo scopo di adottare uno standard che permettesse di utilizzare istanze comuni su cui già altri si sono adoperati e di poter disporre di risultati già noti per metterli in relazione con quelli ricavati durante la realizzazione di questo lavoro, valutarli e magari, sotto alcuni aspetti, anche migliorarli, si è scelto di utilizzare il formato di rappresentazione proposto dal DIMACS per la descrizione dei grafi acquisiti in fase di input dal programma realizzato.

Dato che sono varie le tipologie di problema di interesse del DIMACS, così come i diversi tipi di grafi utilizzati da questi problemi, esistono differenti standard ed indicazioni per la corretta formattazione dei file di input e di output secondo il formato DIMACS. Di seguito verrà descritto solamente lo standard previsto sia per i file di input che descrivono un grafo che fa riferimento ad un problema di vertex coloring, sia per i file di output che ne descrivono la soluzione [2].

4.2 FILE DI INPUT

Un file di input, formattato secondo le indicazioni DIMACS, contiene tutte le informazioni di un grafo necessarie per definire il corrispettivo problema.

Il formato impone la numerazione dei nodi da 1 ad n e un file si dice essere ben formato e consistente se i valori identificativi dei vari nodi sono valori validi, se i nodi sono definiti univocamente e se esattamente m archi sono definiti.

Le informazioni che descrivono il grafo sono organizzate in righe e tre sono le componenti che si possono individuare all'interno di un file DIMACS:

- COMMENTI: i commenti forniscono informazioni circa il file e il grafo rappresentato e sono ignorate dai programmi. Le righe di commento possono apparire in qualsiasi punto del file ed ognuna inizia con il carattere minuscolo c.

  c Questo è un esempio di commento

- PROBLEM LINE: in ogni file si può individuare una e una sola riga di questo tipo; deve apparire prima di qualsiasi descrittore di arco e può essere intesa come una riga di intestazione del problema.

Una problem line ha il seguente formato:

p FORMAT NODES EDGES

dove il carattere minuscolo *p* sta ad indicare che è una problem line; il campo *FORMAT* indica la tipologia di problema cui il grafo seguente fa riferimento e per il problema della colorazione dei vertici dovrebbe essere *"edge"*; i campi *NODES* ed *EDGES* contengono un valore intero che specifica rispettivamente il numero *n* dei nodi e il numero *m* degli archi del grafo.

- EDGE DESCRIPTORS: compare un descrittore d'arco per ogni arco del grafo e ha il seguente formato:

e V W

dove il carattere minuscolo *e* significa che è un edge descriptor e, se *(v, w)* è l'arco descritto, *V* e *W* rappresentano rispettivamente gli estremi *v* e *w* dell'arco. Ogni arco *(v, w)* appare esattamente solo una volta nel file di input e non viene ripetuto come *(w, v)* [2].

In figura 4.1 un esempio di file DIMACS che rappresenta un semplice grafo.

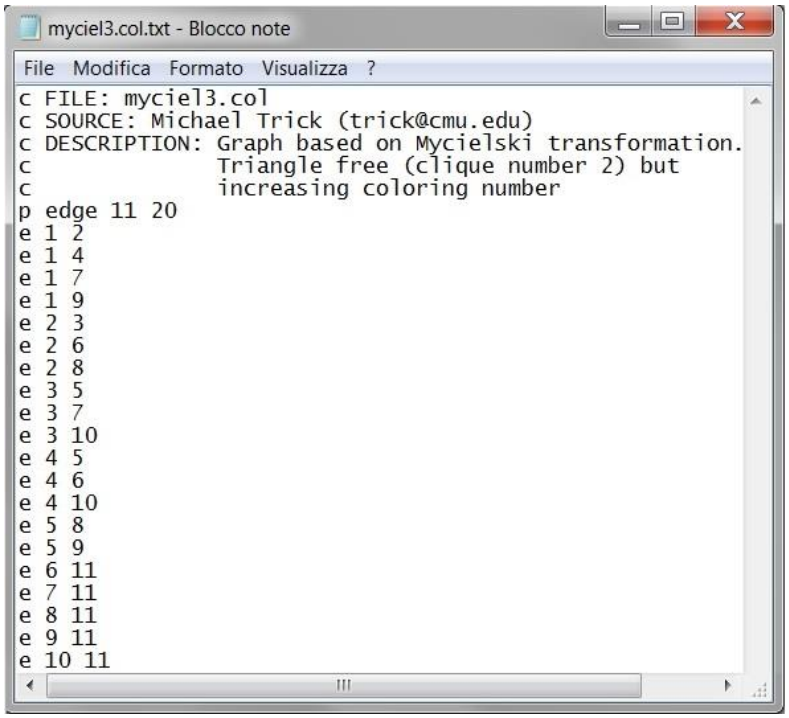


FIGURA 4.1 – ESEMPIO DI FILE DIMACS

4.3 FILE DI OUTPUT

Ogni algoritmo o euristica dovrebbe creare un proprio file di output. Lo standard DIMACS prevede, anche per la descrizione delle informazioni concernenti la soluzione del problema in un file di output, un'organizzazione a righe e le componenti che vi si possono trovare sono:

- SOLUTION LINE

  s TYPE SOLUTION

  Il carattere minuscolo *s* significa che è una solution line. Il campo *TYPE* denota il tipo di problema cui la soluzione, contenuta nel file, fa riferimento e per quanto riguarda il problema della colorazione dei vertici è previsto essere la stringa *"col"*. Il campo *SOLUTION*, infine, contiene un intero corrispondente al valore della soluzione, cioè al numero di colori utilizzati per colorare il grafo.

- BOUND LINE

  b BOUND

  Il carattere minuscolo *b* sta a significare che questo è un bound sulla soluzione e il campo *BOUND* contiene il valore che fornisce tale limite. Il bound è un limite inferiore sul numero di colori necessario per la colorazione del grafo.

- LABEL LINE

  l V N

  Il carattere minuscolo *l* significa che questa è una label line utilizzata per la colorazione dei grafi. Il campo *V* indica il numero del nodo, mentre il campo *N* fornisce la corrispettiva label e quindi ci sarà una label line per ogni nodo del grafo [2].

CAPITOLO 5

GRAPH COLORING PROBLEM

5.1 IL PROBLEMA E SUE GENERALIZZAZIONI

Si considerino i seguenti problemi:

1. Colorare la cartina dell'Inghilterra, in modo tale che due contee che condividono uno stesso confine non si vedano assegnato uno stesso colore, utilizzando il minor numero di colori possibile.

2. Organizzare l'orario degli esami di un'università. Ogni esame necessita di uno slot temporale e l'università vuole organizzare il maggior numero possibile di esami in parallelo, senza eccedere il numero di classi disponibili, in modo da minimizzare il numero di slot temporali utilizzati. Poiché gli studenti possono seguire le lezioni di più di un corso e devono essere messi nella condizione di sostenere l'esame di ognuno di essi, due esami non possono essere fissati nello stesso momento, se c'è almeno uno studente che ha seguito entrambe i relativi corsi.

3. Lo spettro delle radio frequenze deve essere assegnato alle diverse stazioni emittenti per la trasmissione, in modo tale che due stazioni adiacenti, che potrebbero interferire, usino frequenze differenti (ogni stazione potrebbe aver bisogno di una o più frequenze). In generale è richiesto che due stazioni

interferenti usino frequenze tra loro lontane, con una distanza dipendente dal fenomeno della propagazione. Poiché la banda radio è una risorsa limitata e costosa, l'allocazione delle frequenze deve essere la più efficiente possibile, cioè il numero totale di frequenze deve essere minimizzato.

4. Nell'industria metallurgica le bobine vengono trattate in appositi forni. Ogni bobina necessita di essere trattata per almeno un determinato periodo di tempo (differente per ogni bobina) e le bobine che vengono lavorate insieme devono essere compatibili, cioè devono avere altezze simili. Il problema è decidere quali bobine devono essere trattate insieme per minimizzare il tempo totale di lavorazione.

5. Una matrice di traffico $n \times n$ deve essere trasmessa attraverso un satellite analogico. Ogni entry della matrice rappresenta il traffico, cioè la durata della connessione, che deve essere inviato da un'antenna trasmittente ad una ricevente. La matrice, per essere trasmessa, deve essere scomposta in matrici con al più un elemento non nullo per riga e per colonna, corrispondente al traffico trasmesso per ogni coppia di antenne trasmittenti-riceventi, in modo tale che la loro somma corrisponda alla matrice originale. Il tempo di trasmissione di ogni matrice corrisponde al suo elemento più grande e il problema è quello di minimizzare il tempo totale di trasmissione.

6. Dei veicoli sono utilizzati per il trasporto di merci. Alcuni oggetti non possono viaggiare sullo stesso veicolo, perché sono pericolosi o richiedono particolari apparati. Il problema è quello di minimizzare il numero di veicoli, considerando che ogni oggetto ha un peso e la capacità di un veicolo è limitata.

7. Degli aerei sono in avvicinamento ad un aeroporto. Il sistema di controllo del traffico assegna loro un'altitudine a cui volare, mentre sono in attesa del loro turno di atterraggio. Se gli intervalli di arrivo di due aerei si dovessero sovrapporre, questi non potrebbero volare alla medesima quota; quindi, dato che le quote di volo sono limitate esse devono essere assegnate in modo efficiente.

A prima vista, i problemi sopra elencati non hanno nulla in comune. Essi spaziano dalla colorazione di mappe, a problemi nel campo delle telecomunicazioni e dell'industria metallurgica, passando poi attraverso la risoluzione di problematiche legate alla definizione di orari in un'università o legate al mondo dei trasporti e terminando l'elenco con problemi di assegnazione delle quote di volo per gli aerei in attesa di atterraggio. Tuttavia essi sono tutti problemi di ottimizzazione con una struttura simile: una risorsa è condivisa tra più utenti, alcuni dei quali possono accedervi contemporaneamente, mentre altri sono a coppie incompatibili e la risorsa deve essere duplicata; il problema richiede di raggruppare gli utenti che hanno l'accesso simultaneo alla risorsa, in modo tale che il numero di copie della stessa sia minimizzato.

Problemi con questa struttura vengono rappresentati come Vertex Coloring Problems. Formalmente, si consideri un grafo non orientato $G = (V, E)$, dove V è l'insieme dei vertici ed E l'insieme degli archi, aventi rispettivamente cardinalità n ed m. Il Vertex Coloring Problem (VCP) richiede che sia assegnato a ciascun vertice un colore dimodoché due vertici adiacenti non condividano lo stesso colore e il numero di colori complessivamente utilizzato sia minimizzato.

Il problema della colorazione dei vertici di un grafo è un ben noto problema NP-difficile e ha ricevuto grande attenzione nella letteratura, non solo per le sue applicazioni in molteplici campi dell'ingegneria, alcuni dei quali sono stati proposti come esempi, ma anche per i suoi aspetti teorici e per la sua difficoltà dal punto di vista computazionale.

Proprio per la sua rilevanza, sono stati proposti molti algoritmi per la risoluzione del problema, tuttavia questi sono in grado di risolvere in maniera esatta solo istanze di piccola taglia. D'altro canto, per le applicazioni del mondo reale, descritte molto spesso da grafi con molti più nodi e archi, sono stati proposti diversi altri algoritmi, euristici e meta-euristici, che sembrano essere molto promettenti e che sono in grado di trattare grafi con centinaia e migliaia di vertici.

Dato che n colori sono sempre sufficienti per la colorazione di qualsiasi grafo con n nodi, un modello matematico di ovvia derivazione può essere ottenuto definendo i due seguenti insiemi di variabili:

- variabili $x_{i\,k}$ ($i \in V$, $k = 1, \ldots, n$), con $x_{i\,k} = 1$ se e solo se al vertice i è stato assegnato il k-esimo colore e 0 altrimenti;
- variabili y_k ($k = 1, \ldots, n$), con $y_k = 1$ se il colore k è stato usato nella soluzione e 0 altrimenti.

Il modello descrittivo che ne deriva è il seguente:

$$\text{(5.1)} \qquad min \sum_{k=1}^{n} y_k$$

$$\text{(5.2)} \qquad \sum_{k=1}^{n} x_{ik} = 1, \qquad \forall i \in V$$

$$\text{(5.3)} \qquad x_{ik} + x_{jk} \leq y_k, \qquad \forall (i,j) \in E, \;\; k = 1, \ldots, n$$

$$\text{(5.4)} \qquad x_{ik} \in \{0,1\}, \qquad \forall i \in V, \;\; k = 1, \ldots, n$$

$$\text{(5.5)} \qquad y_k \in \{0,1\}, \qquad k = 1, \ldots, n$$

La funzione obiettivo (5.1) minimizza il numero di colori usati. I vincoli del tipo (5.2) richiedono che ogni vertice sia colorato, mentre i vincoli (5.3) impongono che al più uno, di ogni coppia di vertici adiacenti, sia colorato con un determinato colore. Infine i vincoli (5.4) e (5.5) impongono che le variabili siano binarie.

Ritornando alla lista di problemi proposti al principio del paragrafo, il problema 1 è un classico VCP, infatti si può definire un vertice per ogni contea dell'Inghilterra e un arco che connette due vertici se le corrispettive contee condividono un tratto di confine e quindi non possono essere colorate allo stesso modo. Lo stesso si può dire per il problema 7, in cui ad ogni aereo può essere associato un vertice e un arco viene inserito per ogni coppia di vertici che rappresentano due aerei con intervalli di arrivo in aeroporto che si sovrappongono.

Per quanto riguarda il problema 3, la sua modellazione richiede una maggior attenzione per l'introduzione di alcuni vincoli in più; infatti, se similmente ai due precedenti casi, ad ogni stazione emittente viene fatto corrispondere un

vertice, collegandoli tra loro a coppie se sono adiacenti, e ad ogni frequenza viene fatto corrispondere un colore, una soluzione ammissibile prevede la possibilità che ad ogni nodo possano essere assegnati più frequenze-colori, purché la loro distanza, in termini di banda, sia tale da non causare interferenze, sia all'interno della stessa stazione, sia con le stazioni vicine. Quindi nella descrizione del modello si dovranno tenere presenti vincoli non più che impongano la diversità di colori tra due vertici adiacenti, bensì la non intersezione tra insiemi di colori assegnati a vertici collegati tra loro da archi, considerando inoltre che una certa distanza deve essere rispetta tra le frequenze. Da notare che, in questo problema, il numero di colori utilizzato per la colorazione dei vertici potrebbe essere superiore al numero di nodi, appunto per l'ammissibilità dell'assegnazione di più colori ad uno stesso vertice, e può essere inferiore al valore del massimo colore contemplato nella soluzione a causa della distanza tra frequenze necessaria per non avere interferenze; è quindi quest'ultimo valore che la funzione obiettivo dovrà richiedere di minimizzare e non il numero di colori utilizzato, come avveniva nei precedenti casi.

In molte situazioni pratiche, può essere che il numero di utenti che ha accesso alla risorsa sia limitato, o può succedere che ogni utente possa consumare solo una parte dell'intera risorsa e la capacità totale della stessa sia limitata. Una tale situazione può essere modellata assegnando una variabile positiva w_i a ciascun vertice, che ne rappresenti il peso, e imponendo un vincolo di capacità sul peso totale dei vertici che possono ricevere uno stesso colore. Il problema che ne deriva è conosciuto come Bounded Vertex Coloring Problem (BVCP) e, se C è la capacità di ogni colore, il vincolo che ne deriva è della forma:

$$\sum_{i=1}^{n} w_i x_{ik} \leq C, \qquad \forall k = 1, \dots, n \tag{5.6}$$

Il modello (5.1)-(5.5), con l'aggiunta del vincolo (5.6) è un BVCP. Quando i pesi di tutti i vertici sono pari ad 1, il vincolo (5.6) assume il significato di massimo numero di vertici che possono al più essere colorati con lo stesso colore. Questo modello si adatta bene a descrivere, per esempio, il summenzionato problema 2: se un vertice i di peso 1 è associato a ciascun esame, e un colore k a ciascun slot temporale, il vertice i si vedrà assegnare il colore k se, e solo se, sarà fissato nello

slot *k;* due vertici saranno adiacenti, e collegati quindi da un arco, se i corrispettivi esami non possono tenersi nello stesso lasso di tempo, perché esiste almeno uno studente che è interessato a sostenere entrambe; se C è il numero totale di aule disponibili in ogni momento per ospitare gli esami, ogni slot temporale avrà capacità massima, e quindi il termine noto del relativo vincolo, pari a C. Il problema 6 relativo ai trasporti può essere modellato come BVCP associando ogni oggetto da trasportare ad un vertice e, ad ogni veicolo, un colore; C rappresenterà la capacità di trasporto di un veicolo in termini di spazio disponibile e di dimensioni totali della merce trasportabile.

In tutte le situazioni esemplificative e i corrispettivi modelli considerati finora, il costo di ciascun colore è stato impostato uguale ad uno. Tuttavia, in alcuni casi, ciascun vertice i di un grafo G può essere associato ad un peso positivo w_i, e l'obiettivo è quello di minimizzare la somma dei costi dei colori usati, dove il costo di ciascun colore è dato dal massimo peso dei vertici assegnati a quel particolare colore. In questo caso si parla quindi di Weighted Vertex Coloring Problem (WVCP) e il modello che ne scaturisce direttamente richiede, in aggiunta alle variabili x_{ik} che indicano se il vertice i è stato colorato con il colore k, l'utilizzo di una variabile continua z_k ($k = 1, \ldots, n$), che denoti il costo del colore k nella soluzione; il modello è quindi il seguente:

$$\min \sum_{k=1}^{n} z_k \tag{5.7}$$

$$z_k \geq w_i x_{ik}, \qquad \forall i \in V, \ k = 1, \ldots, n \tag{5.8}$$

$$\sum_{k=1}^{n} x_{ik} = 1, \qquad \forall i \in V \tag{5.9}$$

$$x_{ik} + x_{jk} \leq 1, \qquad \forall (i,j) \in E, \ k = 1, \ldots, n \tag{5.10}$$

$$x_{ik} \in \{0,1\}, \qquad \forall i \in V, \ k = 1, \ldots, n \tag{5.11}$$

dove la funzione obiettivo (5.7) minimizza la somma dei costi dei colori definiti dai vincoli (5.8). Questo modello può essere utile per descrivere il problema 4, dove

ogni bobina è associata ad un vertice, il tempo che deve rimanere nel forno al peso del vertice, e le bobine che devono subire il trattamento insieme si vedranno assegnare lo stesso colore, mentre le bobine che non possono entrare nel forno contemporaneamente sono connesse da un arco. Il tempo di lavorazione di un gruppo di bobine che entrano nel forno insieme corrisponde al tempo massimo necessario per il trattamento con quel particolare forno e quindi al costo del colore. Stesso dicasi per il problema 5, in cui un vertice è associato a ciascun elemento non nullo della matrice del traffico, e un arco connette ciascun vertice a tutti i vertici della stessa riga e della stessa colonna. In questo caso un colore corrisponde ad una delle matrici in cui viene scomposta la matrice di partenza [14, 16].

CAPITOLO 6

EURISTICHE PER PROBLEMI MIP GENERICI

6.1 STRATEGIE EURISTICHE BASATE SUL PRINCIPIO DI PROSSIMITÀ TRA SOLUZIONI

La programmazione mista-intera (Mixed Integer Programming – MIP) è un potente strumento per modellare e risolvere problemi d'ottimizzazione complessi. Poiché la risoluzione di problemi MIP rientra nella classe dei problemi NP-hard, l'introduzione di metodi di tipo euristico per affrontare questi problemi diventa molto importante. Un generico modello MIP si presenta nella forma:

$$\min c^T x$$

$$Ax = b$$

$$x \geq 0 \; intere$$

in cui $\min c^T x$ rappresenta la funzione obiettivo che il problema richiede di ottimizzare; $Ax = b$ sono un insieme di vincoli lineari che devono essere soddisfatti nella risoluzione del problema; le variabili x del problema sono definite come intere e non possono assumere valori negativi [4].

Definendo i due insiemi di variabili x_{ik} con $i \in V$ e $k = 1, \ldots, n$, e y_k con $k = 1, \ldots, n$, il modello MIP base per lo specifico problema della colorazione dei grafi, preso in esame in questo lavoro, risulta essere il modello matematico (5.1)-(5.5) già visto nel precedente capitolo e riproposto qui di seguito [14]:

$$min \sum_{k=1}^{n} y_k$$

$$\sum_{k=1}^{n} x_{ik} = 1, \qquad \forall i \in V$$

$$x_{ik} + x_{jk} \leq y_k, \qquad \forall (i,j) \in E, \;\; k = 1, \ldots, n$$

$$x_{ik} \in \{0,1\}, \qquad \forall i \in V, \;\; k = 1, \ldots, n$$

$$y_k \in \{0,1\}, \qquad k = 1, \ldots, n$$

Un approccio di tipo euristico di grande interesse è l'approccio noto in letteratura col nome di local search il quale permette di disporre di soluzioni di eccellente qualità e in tempi rapidi per molte tipologie di problemi anche particolarmente complessi.

L'approccio prevede la definizione di un intorno ben preciso da esplorare che sia prossimo alla soluzione ammissibile corrente, presa come punto di riferimento per il processo d'ottimizzazione. Anziché dover sondare l'intero spazio delle soluzioni, l'euristico restringe la ricerca all'intervallo considerato dove, sperabilmente, si individuerà la nuova soluzione migliorante. Sarà quest'ultima soluzione il nuovo punto di partenza per tentare di migliorare ulteriormente il valore della funzione obiettivo [1].

Integrando la strategia all'interno di un cosiddetto algoritmo di prossimità, per definire l'intervallo di ricerca su cui l'algoritmo deve essere eseguito vengono applicati al modello matematico d'origine nuovi vincoli. Questi vincoli si basano sulla nozione di distanza di Hamming tra due soluzioni e nella fattispecie tra una soluzione ammissibile corrente $\bar{x}$ e una nuova soluzione x, se esistente, e migliore della precedente:

$$\Delta(x, \bar{x}) = \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=1} (1 - x_{ik}) + \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=0} x_{ik}$$

Tali vincoli sono motivati dalla necessità di rendere più veloce il processo d'ottimizzazione, attraverso l'imposizione di bound che limitano lo spazio di ricerca. Ciò consente all'algoritmo di migliorare la soluzione corrente per step successivi, passando da una soluzione migliorante all'altra che siano caratterizzate da una certa vicinanza tra loro (in termini di distanza di Hamming) e quindi da tempi più limitati per individuarle [7].

Per definire l'intervallo di ricerca viene introdotto un nuovo parametro, il valore intero *k*, utile per caratterizzare di volta in volta il raggio dell'intorno da esplorare e che può assumere valori compresi tra 0 e un limite massimo dipendente dalla natura del problema considerato. Il settaggio di questo parametro è di cruciale importanza per la strategia local branching: mentre l'impostazione di k a valori molto bassi può restringere troppo lo spazio di ricerca e quindi portare all'insuccesso il processo di individuazione di una nuova soluzione migliorante, un'impostazione molto lasca con valori elevati assegnati al parametro k può invalidare i vantaggi della strategia euristica del local branching andando ad allungare indefinitamente i tempi di ricerca di una nuova soluzione. Dal momento in cui non vengono individuate nuove soluzioni nel ramo dell'albero di ricerca corrispondente a soluzioni con valori della distanza inferiori o uguali a *k*, l'algoritmo richiede di procedere esaminando l'altro ramo, cioè quello in cui la nuova soluzione è ricercata a distanze maggiori del parametro *k*. È importante quindi individuare il corretto valore per il parametro in modo da bilanciare i due opposti effetti appena descritti, cioè individuare un valore per k tale per cui si possano trovare dei nuovi aggiornamenti per la soluzione corrente e soprattutto in tempi rapidi [7].

I vincoli da introdurre nel modello base sopra esposto per definire il raggio dell'intorno da esplorare sono del tipo [1, 6, 7]:

$$\Delta(x, \bar{x}) = \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=1} (1 - x_{ik}) + \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=0} x_{ik} \leq k$$

e si ottiene così il seguente modello MIP generico:

$$\min c^T x$$

$$Ax = b$$

$$\Delta(x, \bar{x}) \leq k$$

$$x \geq 0 \; intere$$

Entrando nello specifico del problema trattato in questa tesi, l'applicazione di questa strategia euristica avviene in maniera leggermente diversa. Infatti, per definire l'intervallo di ricerca, il parametro k non è considerato come il raggio dell'intorno da esplorare, centrato sulla soluzione corrente, che va a suddividere in due parti disgiunte l'intero spazio delle soluzioni e quindi a creare una ramificazione all'interno dell'albero di ricerca. Esso è piuttosto considerato come il raggio dell'intervallo di distanze centrato sulla soluzione corrente che si dovrà esaminare; nel caso in cui il detto intervallo non contenesse nessuna soluzione migliorante la finestra di ricerca dovrà essere via via allargata fino al limite massimo del numero di nodi del grafo, cioè il numero massimo di cambiamenti che è possibile apportare ad un generica soluzione.

Il modello matematico che descrive il problema della colorazione dei vertici di un grafo e che deriva dall'applicazione di questa strategia euristica secondo quanto appena descritto è il seguente:

(6.1) $$min \sum_{k=1}^{n} y_k$$

(6.2) $$\sum_{k=1}^{n} x_{ik} = 1, \qquad \forall i \in V$$

(6.3) $$\Delta(x, \bar{x}) = \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=1} (1 - x_{ik}) + \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=0} x_{ik} \geq lb$$

(6.4) $$\Delta(x, \bar{x}) = \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=1} (1 - x_{ik}) + \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=0} x_{ik} \leq ub$$

(6.5) $$x_{ik} + x_{jk} \leq y_k, \qquad \forall (i,j) \in E, \;\; k = 1, \dots, n$$

(6.6) $$x_{ik} \in \{0, 1\}, \qquad \forall i \in V, \;\; k = 1, \dots, n$$

(6.7) $$y_k \in \{0,1\}, \qquad k = 1, \dots, n$$

con *lb* e *ub* rispettivamente lower bound e upper bound dell'intervallo di ricerca in esame. Il modello (6.1)-(6.7) permette di evidenziare come un generico modello MIP che utilizzi la strategia local branching possa essere utilizzato per la risoluzione del particolare problema della colorazione di un grafo.

Anziché aggiungere altri vincoli al modello che restringano il campo di ricerca, si potrebbe porre mano alla funzione obiettivo per cercare di semplificare l'esplorazione. Al posto di richiedere la minimizzazione del numero di variabili utilizzate dalla soluzione, la funzione obiettivo potrebbe essere sostituita con un'altra che sia essa a limitare lo spazio di ricerca, penalizzando una soluzione x in accordo con la sua distanza da quella corrente, con l'obiettivo di guidare euristicamente la ricerca verso territori promettenti dello spazio delle soluzioni e, sperabilmente, di individuare nuove e migliori soluzioni in prossimità di quella corrente (sempre in termini di distanza di Hamming).

La nuova funzione obiettivo che va a sostituire l'originale funzione $\min c^T x$ e che deve essere ottimizzata diventa quindi:

$$\min \left\{ \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=1} (1 - x_{ik}) + \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=0} x_{ik} \right\}$$

La riduzione del numero di variabili presenti in soluzione, che è lo scopo del problema da risolvere, avviene attraverso l'inserimento nel modello base di vincoli del tipo:

$$f(x) \leq f(\bar{x}) - \theta$$

dove $\theta > 0$ è la tolleranza imposta.

Questa è la logica alla base della strategia euristica nota in letteratura come proximity search [7].

Da quanto detto il modello descrittivo per un generico problema MIP che utilizzi l'euristica proximity search diventa quindi quello che è possibile osservare di seguito e da cui è possibile notare la sostituzione della funzione obiettivo del modello base con la nuova funzione obiettivo derivata dalla nozione di distanza di

Hamming e l'inserimento del nuovo vincolo che limita il numero di variabili:

$$\min\left\{\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=1}(1-x_{ik})+\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=0}x_{ik}\right\}$$

$$Ax = b$$

$$f(x) \leq f(\bar{x}) - \theta$$

$$x \geq 0 \; intere$$

Per il problema della colorazione dei grafi, la nuova funzione obiettivo presentata richiede di minimizzare il numero di cambiamenti apportati alla colorazione correntemente considerata come quella ottima. Invece, nei vincoli che fissano un valore di tolleranza per il miglioramento minimo richiesto al processo di ottimizzazione, il valore θ deve essere interpretato come il numero di colori in meno che la nuova colorazione da individuare, se esistente, dovrà utilizzare. Per quanto riguarda il parametro θ, si è deciso di utilizzare un valore pari ad 1, perché in questo modo ci si aspetta d'avere una serie di sotto-problemi non troppo difficili da risolvere, ognuno dei quali porta ad un piccolo miglioramento della soluzione [7].

Il modello matematico che deriva dall'applicazione di questa nuova strategia euristica al problema del graph coloring, menzionata in letteratura col nome di proximity search, è quello riportato di seguito:

(6.8) $$\min\left\{\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=1}(1-x_{ik})+\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=0}x_{ik}\right\}$$

(6.9) $$\sum_{k=1}^{n} x_{ik} = 1, \qquad \forall i \in V$$

(6.10) $$x_{ik} + x_{jk} \leq y_k, \qquad \forall (i,j) \in E, \;\; k = 1, \dots, n$$

(6.11) $$\sum_{k=1}^{n} y_k \leq z - 1$$

(6.12) $x_{ik} \in \{0,1\}, \qquad \forall i \in V, \;\; k = 1, \dots, n$

(6.13) $y_k \in \{0,1\}, \qquad k = 1, \dots, n$

con z che rappresenta il numero di colori presenti nella soluzione attuale.

Da sottolineare che la nuova idea che sta alla base della strategia proximity search può essere interpretata, in un certo senso, come la duale di quella su cui si basa la strategia local branching; vi si può infatti notare uno scambio di ruoli tra la funzione obiettivo di partenza e i vincoli che impongono un limite alla distanza entro cui cercare la nuova soluzione migliorante [7].

Un'ulteriore strategia euristica, ancora intentata in letteratura, può essere ricavata come strategia mista, che unisca le due precedenti logiche in un unico algoritmo. In questo caso vengono aggiunti al modello base i vincoli $f(x) \leq f(\bar{x}) - \theta$ propri della logica della strategia proximity search che impongono un miglioramento per il valore della soluzione almeno pari a θ. Anche la funzione obiettivo è sempre quella della logica proximity search che va ad utilizzare la nozione di distanza di Hamming per guidare euristicamente la ricerca e per penalizzare soluzioni lontane da quella correntemente individuata. Questa strategia ibrida contempla anche l'utilizzo dei vincoli $\Delta(x, \bar{x}) \geq lb$ e $\Delta(x, \bar{x}) \leq ub$ tipici della strategia local branching che limitano la ricerca ad un intervallo ben definito.

Il modello che se n'è ricavato è quindi un modello che utilizza la semplificazione d'esplorazione attuata dalla logica della strategia proximity search resa possibile da una nuova funzione obiettivo che penalizza soluzioni non prossime a quella corrente (sempre facendo riferimento alla nozione di distanza di Hamming). Allo stesso tempo la logica proximity search va a confinare la ricerca di una nuova soluzione migliorante, cioè che rispetti il nuovo vincolo di utilizzare un minor numero di variabili presenti in soluzione, all'interno di un limitato spazio d'esplorazione andando ad aggiungere vincoli per la definizione di un lower ed un upper bound, introdotti con la logica local branching.

Il modello matematico relativo a questa strategia ibrida che riunisce le due logiche precedenti viene riportato qui di seguito nella sua forma generale per la descrizione di un generico problema MIP:

$$\min\left\{\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=1}(1-x_{ik})+\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=0}x_{ik}\right\}$$

$$Ax = b$$

$$f(x) \leq f(\bar{x}) - \theta$$

$$\Delta(x,\bar{x}) \leq k$$

$$x \geq 0 \; intere$$

Se il modello generico appena visto e risultante dall'unione delle due logiche local branching e proximity search viene applicato per risolvere il più specifico problema della colorazione di un grafo si trasforma come di seguito presentato (con la stessa precisazione fatta sul parametro k per il modello local branching):

(6.14) $$\min\left\{\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=1}(1-x_{ik})+\sum_{i\in V}\sum_{k:\,\bar{x}_{ik}=0}x_{ik}\right\}$$

(6.15) $$\sum_{k=1}^{n} x_{ik} = 1, \qquad \forall i \in V$$

(6.16) $$x_{ik} + x_{jk} \leq y_k, \qquad \forall (i,j) \in E, \;\; k = 1, \dots, n$$

(6.17) $$\sum_{k=1}^{n} y_k \leq z - 1$$

(6.18) $$\Delta(x,\bar{x}) \geq lb$$

(6.19) $$\Delta(x,\bar{x}) \leq ub$$

(6.20) $$x_{ik} \in \{0,1\}, \qquad \forall i \in V, \;\; k = 1, \dots, n$$

(6.21) $$y_k \in \{0,1\}, \qquad k = 1, \dots, n$$

con z che indica il numero di variabili utilizzate dalla soluzione corrente e con lb e

ub rispettivamente i limiti inferiore e superiore dell'intervallo di ricerca.

Le strategie di tipo euristico che sono state spiegate in questo capitolo sono quelle alla base degli algoritmi euristici realizzati per il progetto di questa tesi. Le modalità con cui esse verranno integrate negli algoritmi saranno ampiamente e dettagliatamente trattate nel prossimo capitolo.

CAPITOLO 7

SOFTWARE IMPLEMENTATO

7.1 FINALITÀ DEL PROGETTO

Dato che lo scopo di questa tesi è quello di realizzare e successivamente analizzare algoritmi euristici e/o strategie che possano in qualche modo apportare miglioramenti a quelli già noti in termini di risultati conseguiti dalla loro esecuzione, e non di individuare algoritmi ad hoc per particolari situazioni reali, il problema della colorazione dei vertici viene considerato nella sua forma più semplice e immediata, tralasciando l'introduzione di termini che considerino una limitazione al numero massimo di vertici che possono ricevere uno stesso colore e tralasciando altresì termini per l'assegnazione di un peso diverso per i singoli nodi.

Il problema viene pertanto considerato indipendentemente dagli scenari in cui può essere utilizzato e, nella fattispecie, assegnando a tutti i vertici uno stesso peso pari a 1 e ad ogni colore una capacità illimitata, facendo riferimento quindi al modello (5.1)-(5.5).

Più in particolare, lo scopo di questo lavoro è quello di indagare sugli effetti del principio di distanza e di prossimità delle diverse soluzioni, utilizzati nella definizione della funzione obiettivo e anche come vincolo nel modello matematico del problema. In questo modo è possibile limitare lo spazio di ricerca a intorni di una soluzione ammissibile di partenza e dare la priorità a ricerche che modifichino il valore del minor numero possibile di variabili.

L'obiettivo è anche quello di migliorare il comportamento di un solver come CPLEX, utilizzato come black-box nella risoluzione del problema della colorazione dei vertici, e comprendere meglio le conseguenze della scelta di modificare i vincoli del modello per la riduzione dello spazio da esplorare, piuttosto che scegliere di apportare modifiche alla funzione obiettivo con lo scopo di semplificare l'esplorazione.

Infatti, tra un algoritmo e l'altro, nella descrizione del modello si può notare uno scambio di ruoli tra la funzione obiettivo di partenza, che richiede la minimizzazione del numero di colori nella ricerca di una nuova soluzione, e il vincolo di distanza, che impone di limitare la ricerca di una nuova soluzione in un intervallo dello spazio "vicino" ad una soluzione già nota.

Da quanto detto sono emersi due diversi algoritmi e, dalla loro combinazione, un terzo algoritmo misto in cui i suddetti funzione obiettivo e vincoli si alternano nei loro ruoli e si supportano nell'indirizzare il solutore verso l'individuazione delle nuove soluzioni.

Nel dettaglio i tre algoritmi cui si è fatto riferimento sono l'algoritmo local branching, l'algoritmo proxy e l'algoritmo hybrid in cui vengono fusi le strutture e i concetti sviluppati nei due precedenti per essere utilizzati congiuntamente.

Nel prosieguo del paragrafo verranno descritti separatamente e trattati in maniera più approfondita i tre diversi algoritmi e la logica da cui hanno origine, ponendo particolare attenzione nel sottolineare i punti più importanti; verrà inizialmente riservato un paragrafo per trattare di tutte quelle parti che sono in comune agli algoritmi.

7.2 Codice, Procedure e Logica Condivisi

Prima della descrizione approfondita degli algoritmi, verrà trattato tutto ciò che è la parte comune che essi condividono, cioè l'inizializzazione e la chiusura dell'ambiente di CPLEX, come viene trattato l'input e l'output del programma, le strutture dati e le variabili chiave e come viene generata la soluzione di partenza.

7.2.1 FILE DI INPUT

Nel paragrafo 4.1 si è detto che il formato che si è preferito utilizzare per l'input è quello proposto dal DIMACS dato che permette di utilizzare istanze e risultati su di esse in comunione con altri che si sono occupati di questo problema.

A questo scopo è stata predisposta una cartella di progetto denominata "*File di Risorse*" in cui è stato inserito il file di testo *graph.txt* per memorizzare la descrizione delle caratteristiche del grafo corrente di cui si vuole individuare la colorazione ottima.

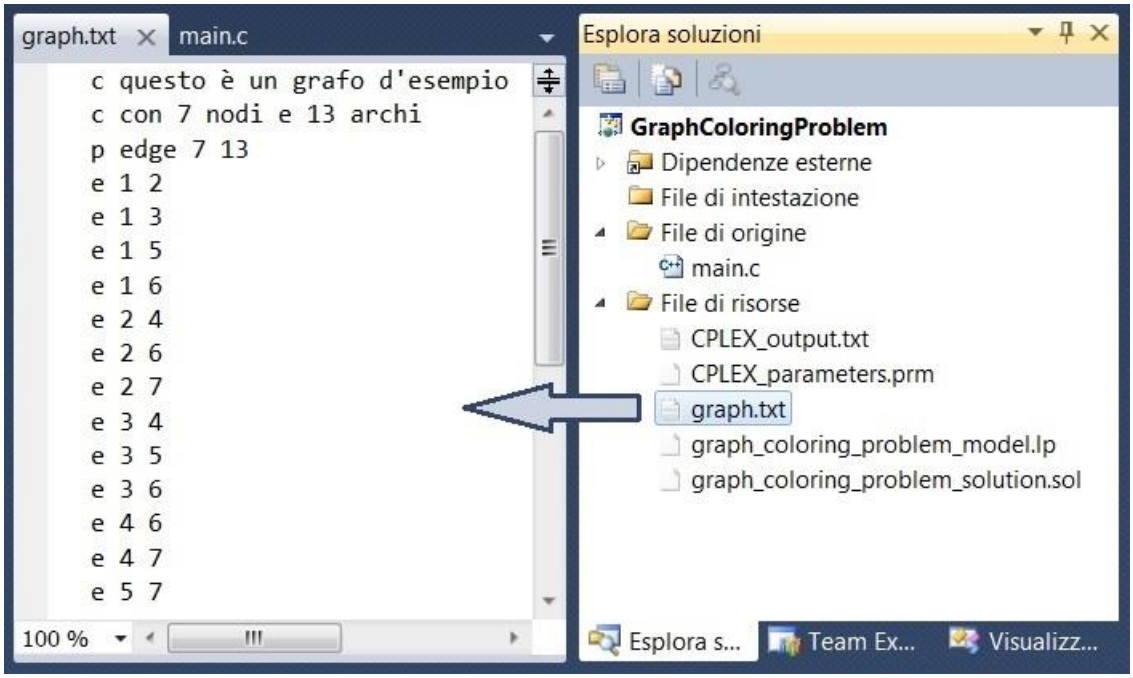


FIGURA 7.1 – VISUALIZZAZIONE DEL FILE GRAPH.TXT

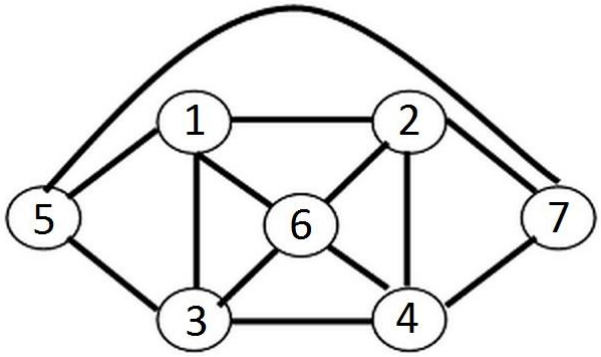
7.2.2 ACQUISIZIONE E MEMORIZZAZIONE DELL'INPUT

Per l'acquisizione dell'input è stata realizzata la funzione *storeMatrix( )* che, scansionando una ad una le varie righe che compongono il suddetto file, ricava il numero di nodi e di archi che compongono il grafo, memorizzandoli nelle due apposite variabili globali *num_nodes* e *num_edges*, e memorizza le altre informazioni di interesse che descrivono il grafo attraverso una cosiddetta matrice

di adiacenza. Una matrice di adiacenza A di un grafo semplice $G = (V, E)$ è la matrice simmetrica $|V| \times |V|$ con elementi

$$a_{ij} = \begin{cases} 1 \; se \; [i,j] \in E \\ 0 \; altrimenti \end{cases}$$

Tale matrice tornerà utile nella costruzione di una soluzione iniziale, permettendone un controllo di ammissibilità [5].



	1	2	3	4	5	6	7
1	0	1	1	0	1	1	0
2	1	0	0	1	0	1	1
3	1	0	0	1	1	1	0
4	0	1	1	0	0	1	1
5	1	0	1	0	0	0	1
6	1	1	1	1	0	0	0
7	0	1	0	1	1	0	0

FIGURA 7.2 – UN GRAFO D'ESEMPIO E LA SUA MATRICE DI ADIACENZA

7.2.3 L'ALGORITMO GREEDY E LA GENERAZIONE DELLA SOLUZIONE INIZIALE

Per la generazione di una colorazione ammissibile iniziale da cui far partire l'esecuzione dell'ottimizzazione da parte di CPLEX, si è scelto di realizzare un algoritmo che prediligesse la velocità piuttosto che la qualità della soluzione, caratteristiche proprie di un algoritmo greedy.

L'algoritmo greedy è stato realizzato attraverso la scrittura della funzione *greedyColoring( )*. Inizialmente viene inizializzato il vettore *color* contenente delle semplici strutture dati composte dalla coppia (*indice, colore*) che sta ad indicare l'indice del vertice e il colore che gli viene associato (all'inizio il colore di default per ogni nodo è 0 che sta ad indicare nessun colore). Vengono anche predisposte le variabili *uncolored* per tenere traccia del numero di nodi che non sono ancora stati colorati e *colorNumber* che rappresenta il colore corrente che si sta utilizzando per

colorare i nodi.

Ad ogni iterazione l'algoritmo greedy controlla se esistono ancora nodi cui non è stato associato nessun colore e in caso positivo inizializza il vettore *current_color_nodes_array* in cui vengono memorizzati gli indici dei nodi associati al colore corrente. A questo punto l'algoritmo individua, se esiste, il successivo nodo che è lecito colorare col colore *colorNumber*. L'individuazione del nodo da colorare col colore corrente è opera della funzione *next_vertex_coloring( )* che, scansionando tutti i vertici non colorati, controlla che il nodo candidato non sia adiacente a nessun altro nodo presente nell'array *current_color_nodes_array*. Il controllo di adiacenza viene eseguito utilizzando la funzione *isAdjacent( )* e le informazioni della matrice di adiacenza.

Se un tal nodo viene individuato, il suo indice è aggiunto all'array dei vertici colorati col colore corrente e nell'array *color* viene aggiornato il suo colore; in caso contrario la funzione *next_vertex_coloring( )* restituisce un segnale che indica l'impossibilità di colorare altri nodi con il colore corrente e dovrà quindi essere cambiato il colore da utilizzare per la colorazione [10, 12].

main.c*

(Ambito globale) greedyColoring()

```
/* funzione che genera una colorazione dei nodi di tipo greedy */
void greedyColoring ()
{
    int uncolored = num_nodes; // numero di vertici non ancora colorati
    int x; // indica se è stato colorato un altro nodo
    int i;
    int colorNumber = 1;


    while ( uncolored > 0 )
    {
        init_current_color_nodes_array ();

        for ( i = 0; i < num_nodes; i++ )
        {
            /* funzione che individua e colora il successivo nodo */
            x = next_vertex_coloring ( current_color_nodes[num_nodes], colorNumber );
            if ( x == -1 ) // non ci sono più nodi cui assegnare il colore corrente
                break;
            else // ( x == 0 ) ho colorato un altro nodo col colore corrente
                uncolored--;
        }
        colorNumber++;
    }
    color[num_nodes].color = colorNumber - 1;
}
```

100 %

FIGURA 7.3 – CODICE ALGORITMO GREEDY

L'algoritmo greedy è deterministico: eseguendolo più volte sullo stesso grafo la soluzione restituita è sempre la medesima. Per ottenere soluzioni diverse si è pensato di realizzare un semplice meccanismo di rimescolamento per perturbare l'ordine iniziale dei vertici, cosicché da imporre all'algoritmo greedy punti di partenza differenti da cui attuare le proprie scelte introducendo un elemento di casualità nella generazione delle soluzioni. Ciò è reso possibile grazie alla funzione *shuffle( )* che utilizza come seme per l'ordinamento random dei nodi il parametro *time* del sistema.

L'algoritmo greedy viene eseguito un certo numero di volte stabilito dall'utente attraverso l'impostazione della costante *ITERATIONS* prima dell'esecuzione del programma e, al termine di ogni iterazione, la nuova soluzione trovata viene messa a confronto con quella già individuata precedentemente e la migliore tra le due, cioè quella che utilizza il minor numero di colori, viene memorizzata nel vettore *best_solution*. Sperimentalmente si è potuto verificare che 1000 iterazioni greedy sono un numero sufficiente per ottenere una buona soluzione iniziale che difficilmente può essere migliorata attraverso un maggior numero di iterazioni.

7.2.4 FILE DI OUTPUT

Per quanto riguarda l'output si è scelto invece di discostarsi dallo standard DIMACS, infatti in rete si trovano solamente risultati in forma tabellare con i dati principali di interesse, cioè quali soluzioni sono state individuate, l'indicazione dell'ottimalità o meno della soluzione, gli algoritmi utilizzati e talune volte le tempistiche.

Pertanto si è considerata sufficiente la creazione di un file di testo, il file *CPLEX_output.txt*, all'interno della cartella "*File di Risorse*", su cui stampare, a seconda delle esigenze e in base a quale algoritmo si sceglie di eseguire, varie informazioni inerenti i risultati dell'esecuzione del programma, per esempio la soluzione greedy di partenza e le soluzioni intermedie, l'indicazione dei colori assegnati ai vari vertici, alcune caratteristiche del grafo come il numero di nodi e

di archi, il numero e l'indicazione dei colori utilizzati, gli intervalli di ricerca correnti, la distanza tra la soluzione corrente e quella precedente, la re-direzione dell'output di CPLEX, eccetera.

Tali informazioni, o perlomeno quelle riguardanti quei risultati che saranno considerati essenziali, verranno riassunte in forma tabellare per renderle disponibili e fruibili in modo più immediato da parte di chi volesse occuparsi del problema della colorazione dei grafi e utilizzare le stesse istanze per valutare i propri algoritmi.

7.2.5 Interfacciamento con CPLEX: Apertura e Chiusura dell'Ambiente

Come già detto nei precedenti capitoli, il linguaggio scelto per l'implementazione del codice del programma è stato il linguaggio C, dato che esiste un apposito insieme di librerie scritte in C, il CPLEX Callable Library, che permette di integrare le funzionalità di ottimizzazione di CPLEX con l'applicativo realizzato.

Quando un'applicazione deve utilizzare le routine del CPLEX Callable Library per la risoluzione di un problema, è necessario precedentemente realizzare l'interfacciamento con il solver, aprendo un nuovo ambiente CPLEX.

CPLEX, per essere eseguito correttamente, necessita di un particolare insieme di strutture dati interne al software. Queste strutture devono essere inizializzate prima di qualsiasi altra chiamata del CPLEX Callable Library, infatti la prima chiamata all'interno del proprio programma deve sempre

```
main()
{
    /* create the CPLEX environment pointer */
    CPXENVptr env = NULL;

    int status = 0;

    /* initialize the CPLEX environment */
    env = CPXopenCPLEX ( &status );

    ...

    /* close the CPLEX environment */
    status = CPXcloseCPLEX ( &env );
}
```

FIGURA 7.4 – APERTURA E CHIUSURA DEL CPLEX ENVIRONMENT

essere quella della funzione *CPXopenCPLEX( )*. Questa routine ritorna un puntatore all'ambiente di CPLEX, il quale deve essere passato a tutte le funzioni del CPLEX Callable Library con solo qualche eccezione per un ristretto numero di funzioni.

A conclusione dell'utilizzo di CPLEX, l'ambiente deve essere rilasciato attraverso la routine *CPXcloseCPLEX( )*. Questa routine specifica a CPLEX che tutte le chiamate necessarie al Callable Library sono state completate e che quindi tutta la memoria allocata dall'ottimizzatore può essere liberata e resa disponibile per il sistema e che l'uso di CPLEX è terminato per quanto riguarda questa esecuzione [8].

7.2.6 ISTANZIAMENTO DEL PROBLEMA

Un problema nell'ambiente CPLEX è istanziato, cioè creato e inizializzato, quando si effettua la chiamata della routine *CPXcreateprob( )*. Ad ogni oggetto problema è possibile fare riferimento attraverso l'utilizzo di un puntatore ritornato dalla suddetta funzione ed esso rappresenta l'istanza del problema che si vuole risolvere. La quasi totalità delle funzioni del Callable Library richiedono come parametro un puntatore ad un oggetto problema (eccetto le funzioni per il settaggio dei parametri d'ambiente e le funzioni per la gestione dei messaggi) e, quando il problema è stato risolto, esso deve essere eliminato con la chiamata della routine *CPXfreeprob( )*.

```
main()
{
    /* create the object problem pointer */
    CPXLPptr lp = NULL;

    int status = 0;

    /* create the object problem */
    lp = CPXcreateprob ( env, &status, "graph_coloring_problem" );

    ...

    /* remove the problem object from the CPLEX environment */
    status = CPXfreeprob ( env, &lp );
}
```

FIGURA 7.5 – ISTANZIAMENTO E RILASCIO DELL'OGGETTO PROBLEMA

7.2.7 POPOLARE IL PROBLEMA

Una volta istanziato con l'istruzione *CPXcreateprob( )*, l'oggetto problema rappresenta un problema vuoto che non contiene dati, ha zero vincoli, zero variabili e una matrice dei vincoli vuota. L'oggetto deve quindi essere popolato con i dati del problema che si sta trattando e questo può essere fatto seguendo diverse modalità:

- il problema può essere popolato predisponendo opportuni array contenenti i dati necessari e richiamando la funzione *CPXcopylp( )* per copiarli all'interno dell'oggetto problema creato;
- alternativamente, l'oggetto creato può essere popolato attraverso l'utilizzo di sequenze di chiamate alle routine *CPXnewcols( )*, *CPXnewrows( )*, *CPXaddcols( )*, *CPXaddrows( )* e *CPXchgcoeflist( )*, secondo l'ordine più conveniente alle proprie esigenze;
- infine, se i dati del problema sono memorizzati in un file MPS o LP, si può ricorrere all'istruzione *CPXreadcopyprob( )* per leggere il file e copiare i dati all'interno dell'oggetto problema [8].

Nel codice implementato per questa tesi si è utilizzata una combinazione dei primi due metodi. Più nello specifico, le funzioni che sono state impiegate, saranno viste e discusse nella sezione dedicata agli algoritmi.

7.2.8 FILE LP E SOL

Quando il problema è stato popolato con i dati, per verificare la correttezza del modello matematico realizzato, si richiama la funzione *CPXwriteprob( )* che genera il file *graph_coloring_problem_model.lp* che riporta la descrizione del problema attraverso la funzione obiettivo e tutti i vincoli imposti. Il file è stato inserito nella cartella di progetto "*File di Risorse*".

Seppur superfluo, infatti vi si può ritrovare la maggior parte delle informazioni di interesse contenute anche nel file creato dal software sviluppato, si è deciso di generare il file *graph_coloring_problem_solution.sol*, in quanto questo

tipo di file, generato direttamente dal solutore CPLEX attraverso la direttiva *CPXsolwrite( )*, formatta le informazioni contenute in *CPLEX_output.txt* in modo diverso adottando lo standard XML ed inoltre viene utilizzato come ulteriore forma di controllo dei risultati ottenuti, contenendo altresì informazioni relativamente ai valori di tutte le variabili e dei vincoli, anche quelle non riportate nel file txt. Anche questo file è stato inserito nella cartella di progetto "*File di Risorse*".

7.2.9 IMPOSTAZIONE DEI PARAMETRI DI CPLEX

Nella cartella "*File di Risorse*" è stato inserito anche il file *CPLEX_parameters.prm* con le impostazioni dei parametri di CPLEX a valori non di default.

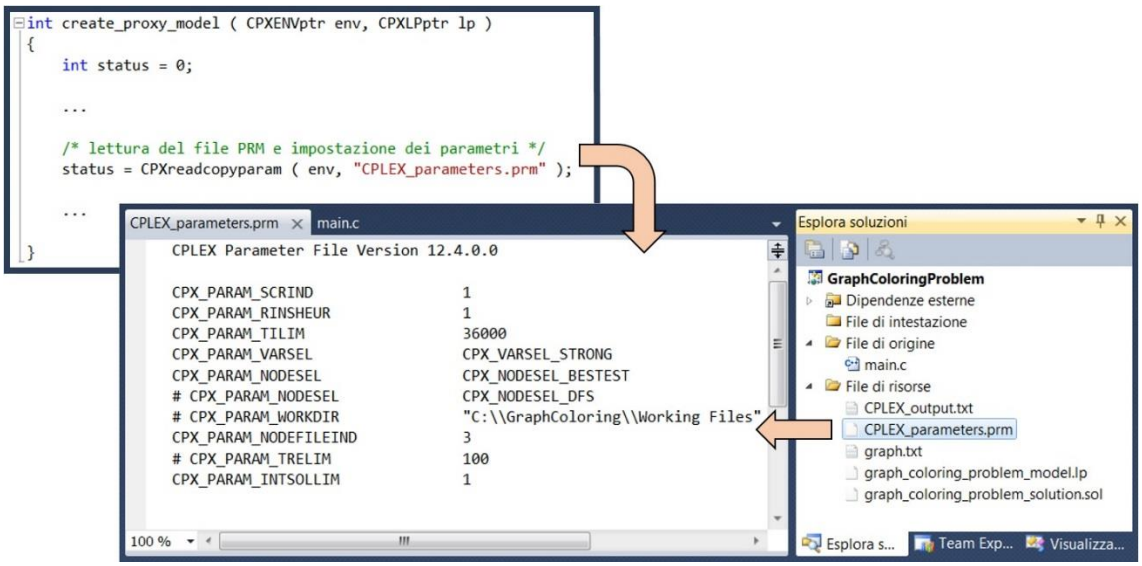


FIGURA 7.6 – LETTURA DEL FILE PRM

Il file è utile onde evitare riformulazioni delle impostazioni dei parametri ogniqualvolta sia necessario, e cioè all'interno del codice scritto per ognuno degli algoritmi, che potrebbero causare errori o dimenticanze. La creazione di questo file permette quindi, per ognuno degli algoritmi, di utilizzare gli stessi valori di parametro, impostati prima dell'esecuzione del programma, con valori opportuni a seconda delle esigenze dell'utente, semplicemente modificando manualmente un'unica volta ogni parametro utilizzato e successivamente, a livello di codice, leggendo il file attraverso la routine del CPLEX Callable Library

CPXreadcopyparam( ). Tale funzione richiede in ingresso solamente il puntatore all'ambiente di CPLEX e il nome del file.

In figura 7.6 il file PRM del programma implementato con i parametri utilizzati; i parametri preceduti dal simbolo "*#*" sono commentati e perciò non influiscono sull'esecuzione corrente di CPLEX, tuttavia sono i parametri impiegati più di frequente con l'obiettivo di valutare e individuare le migliori impostazioni che permettano di massimizzare l'efficienza dello strumento di ottimizzazione.

Nel caso non si utilizzasse il file PRM, dovrebbero essere modificati in maniera opportuna tutti i parametri con le impostazioni desiderate, e questo dovrebbe avvenire per ogni algoritmo con il set di funzioni di CPLEX *CPXsetxxxparam( )* dove al posto delle *"xxx"* è indicata la tipologia di parametro impostato e il valore richiesto come argomento dalla funzione: *"int"* per intero, *"dbl"* per double, *"long"* per long, *"str"* per stringa.

```
int create_proxy_model ( CPXENVptr env, CPXLPptr lp )
{
    int status = 0;

    ...

    status = CPXsetintparam ( env, CPX_PARAM_SCRIND, CPX_ON ); // turn on output to the screen
    status = CPXsetdblparam ( env, CPX_PARAM_TILIM, 36000 ); // time limit = 36000 secondi (=10h)
    status = CPXsetintparam ( env, CPX_PARAM_VARSEL, CPX_VARSEL_STRONG );
    status = CPXsetintparam ( env, CPX_PARAM_NODESEL, CPX_NODESEL_BESTEST );
    //status = CPXsetintparam ( env, CPX_PARAM_NODESEL, CPX_NODESEL_DFS ); // depth-first
    //status = CPXsetstrparam ( env, CPX_PARAM_WORKDIR, "C:\\GraphColoring\\Working Files" );
    status = CPXsetintparam ( env, CPX_PARAM_NODEFILEIND, 3 ); // memorizzazione su disco e compressione
    //status = CPXsetdblparam ( env, CPX_PARAM_TRELIM, 100 ); // limitazione albero a 100MB
    status = CPXsetlongparam ( env, CPX_PARAM_INTSOLLIM, 1 ); // restituisci la prima soluzione fattibile trovata
    status = CPXsetlongparam ( env, CPX_PARAM_RINSHEUR, 1);

    ...

}
```

FIGURA 7.7 – IMPOSTAZIONE DEI PARAMETRI DA CODICE

In figura 7.7 l'impostazione dei parametri da codice e, a seguire, una breve descrizione per ognuno di essi:

- CPX_PARAM_SCRIND permette all'utente di decidere se visualizzare a schermo oppure no tutti i risultati ottenuti durante l'esecuzione di CPLEX. Il parametro viene impostato ad 1 indicando la necessità di visualizzare a schermo i messaggi derivanti dall'esecuzione del software; di default il suo valore è 0 ed indica che nessun messaggio verrà visualizzato.

- CPX_PARAM_TILIM imposta un tempo massimo, in secondi, per una singola chiamata dell'ottimizzatore; se non diversamente specificato, non esistono limiti di tempo per una chiamata.

- CPX_PARAM_VARSEL stabilisce la regola per la selezione della variabile di branching. Il valore di default permette a CPLEX di valutare e selezionare la miglior regola che ritenga opportuna basandosi sul problema e sui progressi effettuati. Tra le altre regole, sono state provate le maximum e minimum infeasibility rule che scelgono la variabile con il valore, rispettivamente, più lontano e più vicino da un valore intero; mentre la prima porta più rapidamente ad una soluzione intera, ma di solito più lentamente al raggiungimento dell'ottimo, la seconda forza ad apportare rapidamente ampi cambiamenti nell'albero. Altra regola utilizzata è stata la strong branching che seleziona la nuova variabile di branching risolvendo un certo numero di sotto-problemi e selezionando la ramificazione più promettente.

- CPX_PARAM_NODESEL permette all'utente di selezionare la strategia da utilizzare per la selezione del prossimo nodo dell'albero da esplorare quando viene eseguito un backtracking. La strategia di default è una strategia di tipo best-bound che seleziona il nodo successivo da esplorare che presenta il miglior valore per la funzione obiettivo del problema. Le altre alternative sono la strategia depth-first che seleziona il nodo creato più recentemente e predilige una ricerca in profondità e la strategia best-estimate che effettua la selezione basandosi sulla miglior stima sul valore intero che la funzione obiettivo potrebbe assumere esplorando un determinato nodo.

- CPX_PARAM_WORKDIR specifica il nome di una directory esistente in cui CPLEX possa memorizzarvi i file temporanei di lavoro.

- CPX_PARAM_WORKMEM specifica un limite superiore sull'ammontare di memoria che CPLEX può sfruttare prima di utilizzare il disco per memorizzarvi i file temporanei di lavoro.

- CPX_PARAM_NODEFILEIND è usato quando la dimensione dell'albero di ricerca eccede la dimensione della memoria di lavoro disponibile. Se il parametro è impostato a 0, al raggiungimento del limite di memoria utilizzabile per l'albero, l'ottimizzazione termina, altrimenti un gruppo di nodi è rimosso dalla memoria. Di default CPLEX trasferisce i nodi in un node-file che mantiene in memoria in forma compressa; le alternative sono di trasferire il file su disco nella sua interezza o di trasferirlo su disco e comprimerlo.

- CPX_PARAM_TRELIM imposta un limite superiore sulla dimensione dell'albero di ricerca; quando questo limite viene raggiunto CPLEX termina la sua esecuzione.

- CPX_PARAM_NODELIM fissa il numero massimo di nodi che sono stati esplorati prima che l'algoritmo termini senza aver raggiunto l'ottimalità.

- CPX_PARAM_INTSOLLIM indica il numero di soluzioni che devono essere individuate prima che il solutore di fermi.

- CPX_PARAM_RINSHEUR determina la frequenza con cui viene applicato l'euristico RINS. L'euristico tenta di migliorare immediatamente la miglior soluzione finora trovata, però non può essere applicato finché CPLEX non abbia prima individuato almeno una soluzione incumbent. Con valore -1 RINS non viene mai applicato, con valore di default 0 l'euristico viene applicato ad intervalli scelti automaticamente da CPLEX, qualsiasi altro intero positivo fissa il numero di nodi dopo cui l'algoritmo viene riattivato. RINS è un potente euristico per individuare soluzioni fattibili di alta qualità, ma può essere costoso un suo utilizzo in termini di memoria. Per ulteriori dettagli sull'euristico si rimanda il lettore al paragrafo 7.3.2.4.1 [3].

7.2.10 OTTIMIZZAZIONE DEL PROBLEMA

In qualsiasi istante dopo la creazione del problema, con l'istruzione *CPXcreateprob( )*, è possibile passare all'ottimizzazione dello stesso attraverso la funzione *CPXmipopt( )*.

Al termine dell'esecuzione di questa routine, non esiste materialmente una soluzione LP, ma piuttosto per ottenere le informazioni della soluzione del problema, senza la creazione di un file di output, bisogna interrogare CPLEX in merito all'oggetto problema creato attraverso una serie di funzioni messe a disposizione dal Callable Library come *CPXgetstat( )*, *CPXgetmipobjval( )*, *CPXgetmipslack( )*, *CPXgetmipx( )*, utilizzate dalla funzione *cplex_print( )* scritta per il programma oggetto di questa tesi [8].

7.2.11 RECENTERING DELLA SOLUZIONE

Quando una colorazione migliorante, se esistente, viene individuata, per evitare che l'ottimizzazione da parte del solver ricominci dal principio, a partire dalla soluzione iniziale restituita dall'algoritmo greedy, si adotta un meccanismo di recentering [7]. Tale meccanismo si concretizza semplicemente nell'aggiornamento della miglior colorazione individuata fino al momento attuale, memorizzata nell'array *best_solution*. Questa procedura è affidata alla funzione *update_best_solution( )*.

La funzione, per operare l'aggiornamento, richiama innanzitutto l'istruzione del Callable Library *CPXgetmipx( )* che interroga CPLEX in merito all'oggetto problema creato e restituisce il valore delle diverse variabili di problema utilizzate.

Se CPLEX è riuscito ad apportare un miglioramento alla soluzione corrente, attraverso l'istruzione *CPXgetmipx( )*, si ottengono tutte le informazioni necessarie ad *update_best_solution( )* per aggiornare i colori assegnati ai diversi vertici, per procedere con il conteggio e l'individuazione dei colori utilizzati, per calcolare la distanza della nuova soluzione da quella precedente; viene anche reso continuo l'intervallo di colori utilizzati in soluzione qualora fosse necessario.

Per comprendere se i colori utilizzati da CPLEX nella colorazione sono continui o meno, durante l'aggiornamento, viene tenuto traccia dell'ultimo colore utilizzato e, tramite differenza, viene confrontato con il numero di colori totali: se il numero di colori totali non è uguale all'indice dell'ultimo colore utilizzato, ma minore, significa che CPLEX ha generato delle discontinuità nell'intervallo di colori.

Nel caso si verificassero delle discontinuità, la funzione deve individuarle e operare un cambio nell'utilizzo dei colori: i colori con gli indici maggiori dovranno sostituire quelli che non sono stati considerati in soluzione e quindi andare a coprire le discontinuità; nel fare questo, si dovrà anche correggere opportunamente il valore di colorazione relativamente a quei vertici colorati con i colori interessati dallo scambio.

7.2.12 COLORI: CONTINUITÀ VS DISCONTINUITÀ

Nell'individuare la nuova colorazione esistono due possibilità per quanto riguarda la scelta dei colori da utilizzare: si possono utilizzare i colori in modo continuo all'interno di un determinato intervallo di valori scartando, nel migliorare una soluzione, quelli identificati da un valore più elevato, oppure si possono utilizzare i colori dell'intervallo eliminando dalla soluzione non necessariamente quelli di valore maggiore e lasciando venire meno la condizione di continuità nel loro utilizzo.

Nel primo caso vengono fissate ad 1 le variabili relative ai primi $(z - 1)$ colori, con z valore della soluzione trovata in precedenza, imponendo che l'eventuale colore da non considerare nella colorazione sia quello di valore z, cioè appunto quello di valore massimo.

Nel secondo caso, invece, non vengono utilizzate regole di precedenza che impongano a CPLEX di preferire l'utilizzo di un colore piuttosto che di un altro, lasciando quindi un maggior grado di libertà al solutore.

Quindi, mentre nel caso dell'utilizzo della discontinuità non sono previsti ulteriori vincoli da aggiungere al modello, nel caso si volesse veder rispettata la

continuità si dovrebbero inserire vincoli che impongano a CPLEX la scelta continua nei colori da utilizzare.

Per la generazione dei vincoli di continuità nel modello è stata implementata la funzione denominata *continuity_colors_constraint( )*. Questa funzione impone la continuità nell'utilizzo dei colori presenti in soluzione introducendo vincoli del tipo:

$$y_k \geq y_{(k+1)}, \qquad k = 1, \dots, (z-1)$$

che riscritti nel formato accettato da CPLEX diventano

$$y_k - y_{(k+1)} \geq 0, \qquad k = 1, \dots, (z-1)$$

con *z* valore intero della soluzione di partenza.

Infatti, se per un certo *k*, la relativa y_k venisse impostata a 1, $y_{(k+1)}$ dovendo essere impostata ad un valore maggiore o uguale ad y_k ed essendo una variabile binaria, potrebbe assumere valore 0 o 1. Però, se per un generico *k*, la relativa y_k venisse impostata a 0, $y_{(k+1)}$, per i suddetti motivi, non può che essere impostata anch'essa a 0; un ragionamento a cascata si riproporrebbe per ogni altro *k* successivo imponendo la continuità nell'utilizzo dei colori.

Ognuno dei vincoli appena visti viene introdotto attraverso la routine *CPXaddrows( )*, richiamata una volta per ogni *k = 1, … , (z - 1)*, secondo quanto è possibile vedere in figura 7.8.

```
status = CPXaddrows ( env, lp, 0, 1, 2, &righthand_side_coeff, &sense,
                      &r_matbeg, r_matind, r_matval, NULL, &name );
```

FIGURA 7.8 – FUNZIONE PER LA CONTINUITÀ DEI COLORI

Ccnt è impostato a 0 in quanto non vengono introdotte nuove variabili; *rcnt* è impostato ad 1 dato che ad ogni chiamata viene aggiunta una sola riga alla matrice dei vincoli; *nzcnt* è impostato a 2 visto che le variabili y_k sono interessate a coppie consecutive e automaticamente è fissata a 2 anche la lunghezza degli array *matind* e *matval*.

I termini noti del vettore *rhs* sono fissati tutti a 0 e, tramite il vettore *sense*, vengono aggiunti vincoli di tipo maggiore-uguale.

Gli array *matind* e *matval* sono impostati in modo tale da valorizzare nello stesso vincolo i coefficienti di coppie consecutive di variabili y_k con valore 1, ma di segno opposto.

Ad ogni vincolo viene assegnato il nome identificativo *"constr_continuity_colors_k"*, con *k* valore intero del colore considerato [3].

Maggiori delucidazioni sul significato dei parametri passati alla funzione *CPXaddrows( )* saranno forniti nel paragrafo 7.3.1.2.

In entrambe i casi, sia che si volesse scegliere di utilizzare la continuità, sia che si volesse lasciare al solutore la scelta dei colori da utilizzare, anche a costo di ottenere delle discontinuità, la soluzione di partenza utilizzata da CPLEX per generare il modello e il file LP dovrà essere una soluzione in cui i colori utilizzati siano continui e, nel caso non lo fossero, dovrà essere predisposta una procedura per coprire le eventuali discontinuità nel loro utilizzo e per uniformare di conseguenza anche la valorizzazione delle variabili che rappresentano le associazioni nodo-colore.

Ambedue le strade sono state utilizzate ed analizzate per verificare quale potesse risultare la più vantaggiosa. Tuttavia, dalle prove effettuate, non sono emerse indicazioni specifiche che potessero far propendere per una o per l'altra e, data la completa libertà di scelta, si è deciso di optare per la discontinuità.

7.2.13 DISTANZA TRA SOLUZIONI

L'idea su cui si basano tutti e tre gli algoritmi sviluppati e che va ad influire sulla definizione della funzione obiettivo e di alcuni vincoli è quella di trovare nuove soluzioni che siano vicine, o meglio, simili, nei valori assunti dalle variabili che le compongono, a quelle precedenti. Questo concetto di vicinanza tra soluzioni permette di restringere lo spazio d'esplorazione ad un intorno limitato e quindi di rendere più efficiente e veloce il miglioramento di una soluzione, almeno fintantoché non diventa importante andare a dimostrare l'ottimalità di una colorazione, per la quale è necessaria la verifica della non esistenza di altre

soluzioni con un minor numero di colori nella totalità dello spazio di ricerca.

Il concetto di distanza utilizzato in questo lavoro è quello della distanza di Hamming:

$$\Delta(x, \bar{x}) = \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=1} (1 - x_{ik}) + \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=0} x_{ik}$$

con $\bar{x}$ soluzione di riferimento e x nuova soluzione da individuare, in cui ogni cambiamento nel valore di una variabile, corrispondente al cambiamento nella colorazione di un vertice, coincide con l'aumento della distanza della nuova soluzione di 1 da quella precedente [7].

7.3 I TRE ALGORITMI EURISTICI

Dopo essere state trattate tutte le parti in condivisione, nel seguente paragrafo verranno descritti e discussi i tre algoritmi euristici implementati e la logica su cui si basano.

Si noti che alcune funzioni realizzate nell'implementazione degli algoritmi sono simili, se non, in certi casi, addirittura le stesse. Queste funzioni saranno descritte approfonditamente solo la prima volta che verranno incontrate nella descrizione del codice, per farne esplicitamente riferimento nelle volte successive.

Anche per ciò che concerne le routine del CPLEX Callable Library si adotterà lo stesso criterio descrittivo, con una trattazione approfondita la prima volta che saranno utilizzate nel codice e con semplici rimandi negli altri casi.

7.3.1 LOCAL BRANCHING ALGORITHM

La logica alla base dell'algoritmo local branching è quella di minimizzare il numero dei colori utilizzati nella colorazione, imponendo che la nuova soluzione sia distanziata da quella precedente il meno possibile. Questo si concretizza in un modello descrittivo in cui, come funzione obiettivo da minimizzare, ritroviamo la sommatoria delle variabili che rappresentano i colori, e tra i vincoli si dovranno

imporre limiti sul valore della distanza entro cui effettuare la ricerca [7].

Il modello risultante è il modello (6.1)-(6.7) che è possibile osservare al paragrafo 6.1.

I vincoli (6.2) vengono trattati come vincoli di maggiore-uguale anziché come uguaglianze: per i vincoli di copertura è del tutto indifferente scriverli come uguaglianza oppure come disuguaglianza. Infatti, se il modello restituisse una determinata soluzione x di valore z, nella quale un generico vertice i venisse colorato con più di un colore, si potrebbe definire a posteriori una nuova soluzione x', nella quale il vertice i sia colorato con uno solo. Per fare questo è necessario colorare il vertice i con un qualunque colore tra quelli usati dalla soluzione x per colorarlo e scartare gli altri. Per definizione questa soluzione x' è ammissibile e non costa più di z. Quindi se la soluzione x era ottima, anche x' è ottima. Dal punto di vista pratico conviene lavorare con vincoli di maggiore-uguale in quanto questi generano meno problemi ai solver.

I vincoli (6.5) vengono riaggiustati, secondo le regole di CPLEX, ponendo tutte le variabili a sinistra e il solo termine noto a destra. I vincoli di adiacenza vengono pertanto riscritti come

$$x_{ik} + x_{jk} - y_k \leq 0, \qquad \forall (i,j) \in E, \;\; k = 1, \dots, n$$

In alternativa a questi vincoli se ne possono scrivere altri del tipo

$$\sum_{i \in C} x_{ik} \leq y_k, \qquad k = 1, \dots, n \tag{7.1}$$

dove C rappresenta una clique appartenente al grafo in esame, definita come un insieme di vertici tale che, ogni sua coppia, sia collegata da un lato; una clique realizza quindi un grafo completo. Dalla definizione di clique si può dedurre che almeno un insieme di colori pari alla cardinalità dell'insieme dei vertici della clique sia necessario per colorarla. Da notare che ogni lato può essere visto come una clique con cardinalità dell'insieme dei vertici uguale a 2. I vincoli (7.1) sono equivalenti ai vincoli (6.5), ma in numero minore. Tuttavia, si è ritenuto opportuno utilizzare questi ultimi, perché se si utilizzassero i vincoli (7.1), bisognerebbe prevedere un'ulteriore procedura per l'individuazione efficiente di tutte le clique massimali all'interno del grafo in esame e anche la risoluzione di un tal problema

risulta essere difficile.

Di seguito i vincoli del modello così come riportati dal file LP generato da CPLEX per un grafo preso come esempio.

```
\ENCODING=ISO-8859-1
\Problem name: graph_coloring_problem

Minimize
 obj: y_0 + y_1 + y_2 + y_3
Subject To
 node_0:               x_0_0 + x_0_1 + x_0_2 + x_0_3 >= 1
 ...
 node_6:               x_6_0 + x_6_1 + x_6_2 + x_6_3 >= 1
 constr_0_0:           x_0_0 + x_1_0 - y_0 <= 0
 ...
 constr_12_3:          x_4_3 + x_6_3 - y_3 <= 0
 constr_proximity_inf: x_0_0 + x_0_1 - x_0_2 + x_0_3 + x_1_0 - x_1_1 + x_1_2
                       + x_1_3 + x_2_0 - x_2_1 + x_2_2 + x_2_3 + x_3_0 + x_3_1
                       - x_3_2 + x_3_3 + x_4_0 + x_4_1 + x_4_2 - x_4_3 - x_5_0
                       + x_5_1 + x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2 + x_6_3
                       >= -7
 constr_proximity_sup: x_0_0 + x_0_1 - x_0_2 + x_0_3 + x_1_0 - x_1_1 + x_1_2
                       + x_1_3 + x_2_0 - x_2_1 + x_2_2 + x_2_3 + x_3_0 + x_3_1
                       - x_3_2 + x_3_3 + x_4_0 + x_4_1 + x_4_2 - x_4_3 - x_5_0
                       + x_5_1 + x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2 + x_6_3
                       <= 13
Bounds
 0 <= x_0_0 <= 1
 ...
 0 <= x_6_3 <= 1
 0 <= y_0 <= 1
 0 <= y_1 <= 1
 0 <= y_2 <= 1
 0 <= y_3 <= 1
Binaries
 x_0_0  x_0_1  x_0_2  x_0_3  x_1_0  x_1_1  x_1_2  x_1_3  x_2_0  x_2_1  x_2_2
 x_2_3  x_3_0  x_3_1  x_3_2  x_3_3  x_4_0  x_4_1  x_4_2  x_4_3  x_5_0  x_5_1
 x_5_2  x_5_3  x_6_0  x_6_1  x_6_2  x_6_3  y_0  y_1  y_2  y_3
End
```

Per generare il modello matematico del problema con la logica local branching si è implementata la funzione *create_local_branching_model( )* che, per generare la funzione obiettivo e i diversi vincoli, nell'ordine, richiama le seguenti altre funzioni:

- coloring_vertex_constraint_local_branching( )
- adjacency_constraint_local_branching( )
- proximity_constraint_with_interval( )
- binary_constraint( )

Dato che ogni iterazione dell'algoritmo verrà eseguita su un limitato intervallo dello spazio delle soluzioni, per dimostrare l'ottimalità di una determinata colorazione tutti gli intervalli dovranno essere esaminati. Ad ogni iterazione, se viene individuata una nuova soluzione, l'intervallo di ricerca viene riportato al valore iniziale, se invece non esistono miglioramenti della corrente colorazione nell'intervallo esaminato, la finestra di ricerca viene aggiornata. L'ottimalità è dimostrata quando tutti gli intervalli sono stati esplorati, e quindi anche la massima distanza possibile corrispondente al cambiamento dei colori di tutti i nodi è stata considerata, senza individuare nuove soluzioni. L'algoritmo termina quindi la sua esecuzione quando l'intervallo corrente oltrepassa il limite superiore del numero di nodi del grafo.

Di seguito verrà descritto come è stata implementata ciascuna delle funzioni elencate, indicato quali routine del CPLEX Callable Library richiama e spiegato il suo ruolo nella creazione del modello.

7.3.1.1 COLORING_VERTEX_CONSTRAINT_LOCAL_BRANCHING( )

Questa funzione viene utilizzata per aggiungere al modello il vincolo che impone la colorazione di ogni vertice; vengono anche create e predisposte le variabili $x_{i\,k}$ necessarie per generare il vincolo. Nel modello del problema saranno quindi presenti questi vincoli in numero pari al numero dei nodi del grafo che si sta considerando.

Come visibile dall'estratto del file LP nel paragrafo 7.3.1, i vincoli generati dalla presente funzione sono quelli del tipo:

node_0: *x_0_0 + x_0_1 + x_0_2 + x_0_3 >= 1*

Come detto precedentemente per popolare il problema si ricorre alla routine del Callable Library *CPXcopylp( )*, ma in una sua versione differente, cioè *CPXcopylpwnames( )*, che oltre a copiare i dati all'interno dell'oggetto problema di CPLEX, utilizza alcuni argomenti addizionali che permettono all'utente di

specificare il nome dei vincoli e delle variabili.

I parametri richiesti dalla routine sono quelli visibili di seguito in figura 7.9 e successivamente descritti:

```
status = CPXcopylpwnames (env,
                          lp,
                          numcols,
                          numrows,
                          objsen,
                          obj,
                          rhs,
                          sense,
                          matbeg,
                          matcnt,
                          matind,
                          matval,
                          lb,
                          ub,
                          rngval,
                          colname,
                          rowname);
```

FIGURA 7.9 – FUNZIONE CPXCOPYLPWNAMES( )

- *env* e *lp* sono i due puntatori di CPLEX che fanno riferimento rispettivamente all'ambiente ritornato dalla funzione *CPXopenCPLEX( )* e all'oggetto problema restituito da *CPXcreateprob( );*
- *numcols* è un intero che specifica il numero di colonne della matrice dei vincoli, o equivalentemente, il numero delle variabili del problema;
- *numrows* è un intero che indica il numero di righe della matrice dei vincoli, non considerando la funzione obiettivo e i limiti sui valori che possono assumere le variabili;
- *objsense* è un intero che può assumere i valori 1 o -1 e che fa riferimento alla tipologia di problema che si sta trattando, cioè se è un problema, rispettivamente, di minimizzazione o di massimizzazione. Equivalentemente, in sostituzione dell'intero, si può utilizzare il nome della variabile che indica la tipologia di problema come riportato dalla tabella 7.1;

objsense	= 1	(CPX_MIN) minimize
objsense	= -1	(CPX_MAX) maximize

TABELLA 7.1 – IMPOSTAZIONI PER OBJSENSE

- *objective* è un array di lunghezza almeno *numcols* contenente i coefficienti

della funzione obiettivo;

- *rhs* è un array di lunghezza almeno *numrows* contenente il termine noto relativo a ciascun vincolo della matrice dei vincoli;
- *sense* è un array di lunghezza almeno *numrows* contenente il verso di ogni vincolo della matrice. La tabella 7.2 mostra i possibili valori per il parametro.

`sense[i]`	`= 'L'`	<= constraint
`sense[i]`	`= 'E'`	= constraint
`sense[i]`	`= 'G'`	>= constraint
`sense[i]`	`= 'R'`	ranged constraint

TABELLA 7.2 – IMPOSTAZIONI PER SENSE

- *matbeg* (punto d'inizio della matrice), *matcnt* (contatori della matrice), *matind* (indici dei valori della matrice) e *matval* (valori dei coefficienti della matrice) sono quattro array che definiscono la matrice dei vincoli. In riferimento a questi argomenti, CPLEX necessita di conoscere solamente i coefficienti non nulli della matrice e questi vengono raggruppati per colonna nell'array *matval*. Gli elementi non nulli di ciascuna colonna devono essere memorizzati in locazioni consecutive in questo array, con *matbeg[j]* contenente l'indice d'inizio della *j-esima* colonna e *matcnt[j]* contenente il numero di entry della colonna *j*. Le componenti di *matbeg* devono essere in ordine crescente. Per ogni *k*, *matind[k]* specifica il numero di riga del corrispondente coefficiente *matval[k]*.
  Con un esempio, si supponga che CPLEX debba avere accesso alle entry di una generica colonna *j*. I valori cercati sono nell'array *matval* nelle locazioni:

  $$matval[\ matbeg[j]\ ], \ldots, matval[\ matbeg[j] + matcnt[j] - 1\ ].$$

  I corrispondenti indici si troveranno nell'array *matind* nelle locazioni:

  $$matind[\ matbeg[j]\ ], \ldots, matind[\ matbeg[j] + matcnt[j] - 1\ ].$$

  La lunghezza degli array *matbeg* e *matcnt* deve essere almeno pari a *numcols*. La lunghezza degli array *matind* e *matval* deve essere almeno pari a $matbeg[\ numcols - 1\ ] + matcnt[\ numcols - 1\ ]$.
- *lb* e *ub* sono due array di lunghezza almeno *numcols* contenenti rispettivamente i lower bound e gli upper bound di ciascuna variabile;

- *rngval* è un array di lunghezza almeno *numrows* contenente il range di valori di ciascun vincolo la cui validità sia limitata all'interno di un determinato intervallo e può anche essere impostato a NULL;
- *colname* e *rowname* sono due array di lunghezza almeno *numcols* e *numrows* contenenti puntatori a stringhe di caratteri che rappresentano i nomi che si intendono associare rispettivamente alle colonne o variabili e alle righe o vincoli della matrice dei vincoli [3].

Nella funzione *coloring_vertex_constraint_local_branching( )*, per la generazione del vincolo che impone che ogni vertice riceva un colore (in realtà, come già visto, almeno un colore), vengono inizialmente create le variabili x_{ik}, una per ogni nodo e per ogni colore utilizzato nella soluzione, con lower bound 0 e upper bound 1 e con nome *"x_i_k"*.

I coefficienti delle variabili appena create vengono impostati tutti a 1 nel vettore *matval* e a 0 nel vettore *objective*, dato che la funzione obiettivo, in questo caso, richiede la minimizzazione solamente del numero di colori, e cioè della sommatoria delle variabili y_k.

I termini noti del vettore *rhs* vengono impostati tutti pari ad 1 ed i vincoli, nel vettore *sense*, vengono specificati essere tutti vincoli di maggiore-uguale.

Il nome identificativo del vincolo che impone la colorazione del generico vertice *i* è stato deciso essere *"node_i"*.

Dopo aver predisposto tutti i parametri necessari, viene chiamata la funzione *CPXcopylpwnames( )*.

```
/* x_{ik} variables */
for ( i = 0; i < num_nodes; i++ )
{
    for ( k = 0; k < best_solution[num_nodes].color; k++ )
    {
        matbeg[contV] = contE;
        matcnt[contV] = 0;
        matind[contE] = i;
        matval[contE] = 1.0;
        contE++;
        matcnt[contV]++;
        lb[contV] = 0.0;
        ub[contV] = 1.0;
        obj_function_coeff[contV] = 0.0;

        colname[contV] = (char *) calloc ( 20, sizeof(char) );
        sprintf ( colname[contV], "x_%d_%d", i, k );
        contV++;
    }
}

/* constraints node_i */
for ( i = 0; i < nrows; i++ )
{
    righthand_side_coeff[i] = 1.0;
    sense[i] = 'G';
    rowname[i] = (char *) calloc ( 20, sizeof(char) );
    sprintf ( rowname[i], "node_%d", i );
}

status = CPXcopylpwnames ( env, lp, ncols, nrows, CPX_MIN, obj_function_coeff, righthand_side_coeff,
                           sense, matbeg, matcnt, matind, matval, lb, ub, NULL, colname, rowname );
```

FIGURA 7.10 – PARTI PRINCIPALI DELLA FUNZIONE COLORING_VERTEX_CONSTRAINT_LOCAL_BRANCHING( )

7.3.1.2 ADJACENCY_CONSTRAINT_LOCAL_BRANCHING( )

Questa funzione viene utilizzata per la creazione del vincolo che impone una differente colorazione per vertici del grafo che sono adiacenti; vengono anche create e predisposte le variabili y_k necessarie sia per il vincolo, sia per la descrizione della funzione obiettivo. Il numero dei vincoli di adiacenza che compaiono nella descrizione del problema è pari al numero di archi moltiplicato per il numero di colori utilizzato nella colorazione.

Con riferimento al file LP riportato in parte nel paragrafo 7.3.1, i vincoli introdotti dalla presente funzione sono quelli del tipo:

constr_0_0: *x_0_0 + x_1_0 - y_0 <= 0*

In questo caso per l'aggiunta dei vincoli si è utilizzata la routine *CPXaddrows( )* che aggiunge una riga per volta, una per ognuno dei vincoli.

I parametri per la chiamata della routine vengono illustrati in figura 7.11 e descritti nel successivo elenco:

```
status = CPXaddrows (env,
                     lp,
                     ccnt,
                     rcnt,
                     nzcnt,
                     rhs,
                     sense,
                     rmatbeg,
                     rmatind,
                     rmatval,
                     newcolname,
                     newrowname);
```

FIGURA 7.11 – FUNZIONE CPXADDROWS( )

- *env* e *lp* sono i due puntatori all'ambiente e all'oggetto problema creati da CPLEX;
- *ccnt* è un intero che specifica il numero di nuove colonne che verranno aggiunte nella matrice dei vincoli. Quando sono previste nuove colonne, e

cioè nuove variabili, ad esse vengono assegnati coefficienti nella funzione obiettivo uguali a 0, lower bound 0 e limite superiore infinito;

- *rcnt* è *nzcnt* sono due interi che indicano rispettivamente il numero di nuove righe che stanno per essere aggiunte al problema e il numero di coefficienti non nulli in esse compresi. *nzcnt* specifica anche la lunghezza degli array *rmatind* e *rmatval;*
- *rhs* e *sense* sono due array di lunghezza *rcnt* contenenti i termini noti e la tipologia di ogni vincolo aggiunto al problema;
- *rmatbeg*, *rmatind* e *rmatval* sono tre array usati per definire le nuove righe di vincoli aggiunte, similmente a quanto avviene con gli array *matbeg*, *matcnt*, *matind* e *matval* nella funzione *CPXcopylpwnames( )*, ma i coefficienti non nulli sono raggruppati, in locazioni consecutive, per riga anziché per colonna nell'array *rmatval* e gli elementi non nulli di ogni riga sono memorizzati dalla posizione *rmatbeg[i]* a *rmatbeg[i + 1] – 1* (o da *rmatbeg[i]* a *nzcnt – 1* se *i = rcnt – 1*);
- *colname*, di lunghezza *ccnt*, e *rowname* sono due array contenenti stringhe di caratteri che rappresentano i nomi che si vogliono assegnare alle nuove variabili e ai vincoli appena aggiunti [3].

Nella funzione *adjacency_constraint_local_branching( )*, prima di chiamare la routine *CPXaddrows( )*, vengono predisposti gli array e le variabili che devono essere passati come argomenti.

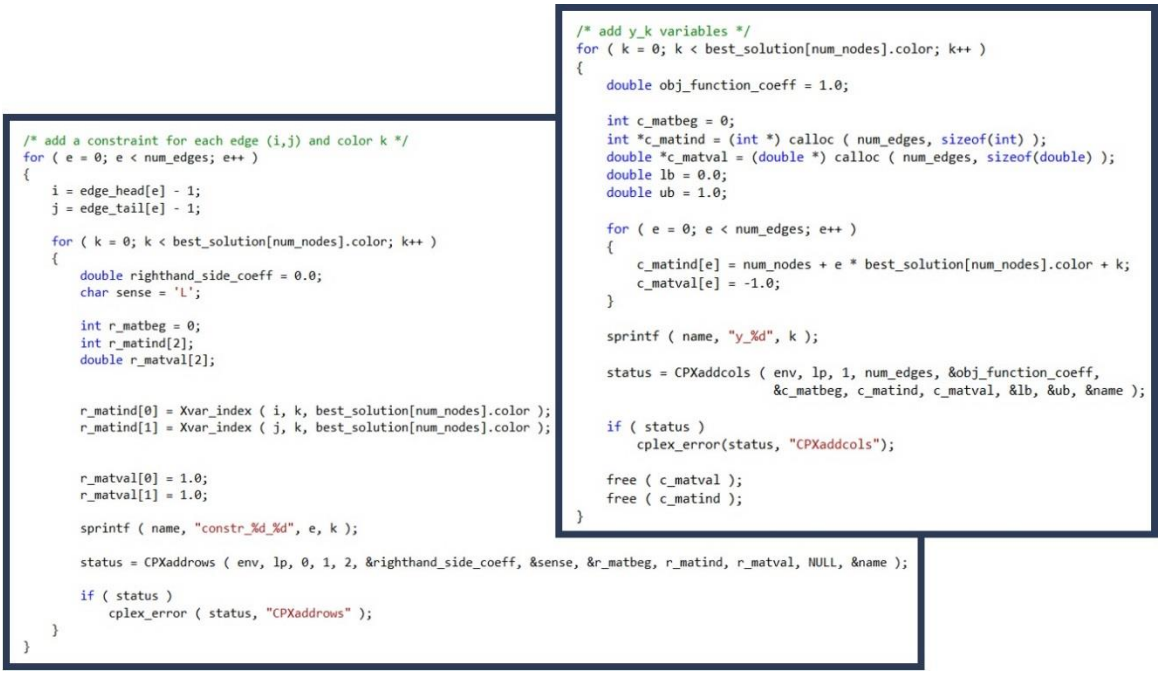
Ccnt e *rcnt* sono impostati rispettivamente a 0 e 1 dato che non è previsto l'inserimento di nuove variabili e ad ogni iterazione viene aggiunta alla matrice dei vincoli una riga alla volta.

Ogni vincolo interessa due variabili $x_{i\,k}$ e $x_{j\,k}$, con i e j nodi agli estremi dell'arco e corrente e k il colore che si sta prendendo in esame. Quindi i vettori *rmatbeg*, *rmatind* e *rmatval* saranno tutti composti da soli due elementi, come si evince dall'impostazione del parametro *nzcnt*, e, nel caso di *rmatval*, i coefficienti verranno impostati ad 1.

I termini noti del vettore *rhs* vengono impostati tutti a 0 e il vettore *sense* sarà predisposto per descrivere vincoli di minore-uguale.

Ogni vincolo sarà identificato dal nome inserito nel vettore *rowname* e sarà del tipo *"constr_e_k"*, con e e k interi riferiti rispettivamente al numero dell'arco e

del colore che si stanno prendendo in considerazione.



```
/* add a constraint for each edge (i,j) and color k */
for ( e = 0; e < num_edges; e++ )
{
    i = edge_head[e] - 1;
    j = edge_tail[e] - 1;

    for ( k = 0; k < best_solution[num_nodes].color; k++ )
    {
        double righthand_side_coeff = 0.0;
        char sense = 'L';

        int r_matbeg = 0;
        int r_matind[2];
        double r_matval[2];


        r_matind[0] = Xvar_index ( i, k, best_solution[num_nodes].color );
        r_matind[1] = Xvar_index ( j, k, best_solution[num_nodes].color );


        r_matval[0] = 1.0;
        r_matval[1] = 1.0;

        sprintf ( name, "constr_%d_%d", e, k );

        status = CPXaddrows ( env, lp, 0, 1, 2, &righthand_side_coeff, &sense, &r_matbeg, r_matind, r_matval, NULL, &name );

        if ( status )
            cplex_error ( status, "CPXaddrows" );
    }
}
```

```
/* add y_k variables */
for ( k = 0; k < best_solution[num_nodes].color; k++ )
{
    double obj_function_coeff = 1.0;

    int c_matbeg = 0;
    int *c_matind = (int *) calloc ( num_edges, sizeof(int) );
    double *c_matval = (double *) calloc ( num_edges, sizeof(double) );
    double lb = 0.0;
    double ub = 1.0;

    for ( e = 0; e < num_edges; e++ )
    {
        c_matind[e] = num_nodes + e * best_solution[num_nodes].color + k;
        c_matval[e] = -1.0;
    }

    sprintf ( name, "y_%d", k );

    status = CPXaddcols ( env, lp, 1, num_edges, &obj_function_coeff,
                          &c_matbeg, c_matind, c_matval, &lb, &ub, &name );

    if ( status )
        cplex_error(status, "CPXaddcols");

    free ( c_matval );
    free ( c_matind );
}
```

FIGURA 7.12 – PARTI PRINCIPALI DELLA FUNZIONE ADJACENCY_CONSTRAINT_LOCAL_BRANCHING( )

Per completare ciascun vincolo di adiacenza, devono essere inserite anche le variabili y_k, rappresentanti i colori utilizzati in soluzione. Dato che l'inserimento di nuove variabili corrisponde all'aggiunta di nuove colonne nella matrice dei vincoli, per la generazione delle variabili y_k si utilizza l'istruzione *CPXaddcols( )*.

Gli argomenti necessari per l'utilizzo della routine sono riportati in figura 7.13 e trattati a seguire nell'elenco:

```
status = CPXaddcols (env,
                     lp,
                     ccnt,
                     nzcnt,
                     obj,
                     cmatbeg,
                     cmatind,
                     cmatval,
                     lb,
                     ub,
                     newcolname);
```

FIGURA 7.13 – FUNZIONE CPXADDCOLS( )

- *env* e *lp* corrispondono all'ambiente e all'oggetto problema creati con

CPXcreateprob( );

- *ccnt* e *nzcnt* sono due interi che specificano il numero di nuove colonne che devono essere aggiunte e il numero di coefficienti non nulli che le compongono;
- *obj* è un array di lunghezza *ccnt* contenente i coefficienti che le nuove variabili hanno all'interno della funzione obiettivo;
- *cmatbeg*, *cmatind* e *cmatval* sono tre vettori per la definizione degli elementi non nulli nelle colonne create e sono il corrispettivo dei vettori *rmatbeg*, *rmatind* e *rmatval* nella funzione *CPXaddrows( );*
- *lb* e *ub*, array di lunghezza *ccnt* contenenti i lower bound e gli upper bound delle nuove variabili, sono necessari per definire i limiti dell'intervallo, all'interno del quale, queste ultime possano assumere valori validi;
- *colname*, di lunghezza *ccnt*, specifica un nome per le nuove colonne [3].

Come si può notare dalla descrizione, *CPXaddcols( )* è molto simile alla funzione *CPXaddrows( )*, tuttavia mentre la prima non può aggiungere coefficienti in righe che già non esistano, *CPXaddrows( )* può aggiungere coefficienti in nuove colonne create con la stessa funzione.

Prima della chiamata della routine di CPLEX, vengono predisposti i vari parametri da passare come argomento: i coefficienti della funzione obiettivo vengono impostati a 1, dato che la funzione obiettivo richiede la minimizzazione della variabili y_k associate ai colori, mentre i coefficienti relativi ai vincoli di adiacenza sono fissati a -1; i limiti di validità, inferiore e superiore, sono impostati a 0 e 1; il nome assegnato ad ogni colonna di variabile sarà *"y_k"*, con *k* il numero identificativo del colore. Il numero di colonne aggiunte ad ogni chiamata della routine è 1 e le righe della matrice dei vincoli interessate dall'inserimento di ciascuna nuova variabile è pari al numero degli archi del grafo.

Le impostazioni, così come sono state appena descritte, sono illustrate dalla precedente figura 7.13.

7.3.1.3 PROXIMITY_CONSTRAINT_WITH_INTERVAL( )

Questa funzione, molto importante nella logica dell'algoritmo, limita la distanza delle nuove colorazioni generate dall'ottimizzatore CPLEX, con riferimento alle variabili $x_{i\,k}$ che rappresentano la colorazione dei singoli nodi del grafo.

Inizialmente concepita per inserire nel modello del problema il vincolo di prossimità, cioè un vincolo che limitasse la ricerca ad un determinato intorno della soluzione corrente, è stata modificata per permettere comunque di limitare la ricerca rispetto alla totalità dello spazio delle soluzioni, ma impostando un intervallo, anziché un intorno. Questo cambiamento riduce lo sforzo computazionale quando non si individuano altre soluzioni vicine a quella corrente e si deve allargare l'intorno iniziale; l'allargamento comporta un maggior spazio in cui poter trovare nuove soluzioni, ma allo stesso tempo un determinato numero di operazioni di ricerca non necessarie in quanto già eseguite prima dell'aggiornamento dell'intorno. Con l'aggiornamento di un intervallo, anziché di un intorno, si eliminano tutte le operazioni di ricerca superflue, cioè già eseguite, in quanto si andrebbe ad aumentare l'intorno di ricerca, ma dato che una determinata parte è già stata esplorata, la si tralascerebbe esplorando solo la nuova parte appena aggiunta.

In relazione al paragrafo 7.3.1 in cui viene riportata come esempio una parte del file LP di un grafo, i vincoli introdotti sono quelli del tipo:

constr_proximity_inf: *x_0_0 + x_0_1 – x_0_2 + x_0_3 + x_1_0 – x_1_1 + x_1_2 + x_1_3 + x_2_0 – x_2_1 + x_2_2 + x_2_3 + x_3_0 + x_3_1 – x_3_2 + x_3_3 + x_4_0 + x_4_1 + x_4_2 – x_4_3 – x_5_0 + x_5_1 + x_5_2 + x_5_3 – x_6_0 + x_6_1 + x_6_2 + x_6_3 >= –7*

constr_proximity_sup: *x_0_0 + x_0_1 – x_0_2 + x_0_3 + x_1_0 – x_1_1 + x_1_2 + x_1_3 + x_2_0 – x_2_1 + x_2_2 + x_2_3 + x_3_0 + x_3_1 – x_3_2 + x_3_3 + x_4_0 + x_4_1 + x_4_2 – x_4_3 – x_5_0 + x_5_1 + x_5_2 + x_5_3 – x_6_0 + x_6_1 + x_6_2 + x_6_3 <= 13*

I vincoli per limitare la distanza della nuova soluzione da quella attuale sono

due: uno per impostare il limite inferiore e uno per l'impostazione di quello superiore. Questo risultato è ottenuto con una duplice chiamata della routine *CPXaddrows( )* con parametri *ccnt* e *rcnt* rispettivamente 0 e 1, visto che non vengono aggiunte nuove colonne e che viene introdotto solo un vincolo per chiamata.

Le differenze tra le due chiamate, nell'impostazione degli argomenti passati, si possono notare, oltre che nell'ovvio cambiamento del nome dei vincoli per poterli distinguere (*"constr_proximity_inf"* per il limite inferiore, *"constr_proximity_sup"* per quello superiore), nell'altrettanto logica diversità di settaggio dell'argomento *sense* e dell'argomento *rhs*, per le direzioni dei vincoli e per la definizione del valore del termine noto.

```c
/* funzione che aggiunge il vincolo di prossimità */
void proximity_constraint_with_interval ( CPXENVptr env, CPXLPptr lp )
{
    int i, k;
    int status = 0;
    int num_x = num_nodes * best_solution[num_nodes].color;
    int nonzero_coeff = num_x;

    /* indicano l'intervallo di ricerca entro cui cercare la nuova soluzione */
    double inf_d = inf;
    double sup_d = sup;
    double righthand_side_coeff_inf = inf_d * 2 - num_nodes;
    double righthand_side_coeff_sup = sup_d * 2 - num_nodes;

    /* indicano il verso dei due vincoli che delimitano l'intervallo di ricerca */
    char sense_inf = 'G'; // limite inferiore
    char sense_sup = 'L'; // limite superiore

    int r_matbeg = 0;
    int *r_matind = (int *) calloc ( num_x, sizeof(int) );
    double *r_matval = (double *) calloc ( num_x, sizeof(double) );

    char *name = (char *) calloc ( 30, sizeof(char) );

    for ( i = 0; i < num_nodes; i++ )
    {
        for ( k = 0; k < best_solution[num_nodes].color; k++ )
        {
            r_matind[i * best_solution[num_nodes].color + k] = i * best_solution[num_nodes].color + k;

            if ( k == best_solution[i].color - 1 )
                r_matval[i * best_solution[num_nodes].color + k] = -1.0;
            else
                r_matval[i * best_solution[num_nodes].color + k] = 1.0;
        }
    }
```

```c
/* limite inferiore */
sprintf ( name, "constr_proximity_inf" );

status = CPXaddrows ( env, lp, 0, 1, nonzero_coeff, &righthand_side_coeff_inf,
                      &sense_inf, &r_matbeg, r_matind, r_matval, NULL, &name );

/* limite superiore */
sprintf ( name, "constr_proximity_sup" );

status = CPXaddrows ( env, lp, 0, 1, nonzero_coeff, &righthand_side_coeff_sup,
                      &sense_sup, &r_matbeg, r_matind, r_matval, NULL, &name );
```

FIGURA 7.14 – PARTI PRINCIPALI DELLA FUNZIONE PROXIMITY_CONSTRAINT_WITH_INTERVAL( )

Il punto chiave di questa funzione si riscontra nella scelta dell'impostazione dei coefficienti delle variabili $x_{i\,k}$ nell'array *rmatval:* dato che il cambiamento di valore di una di esse corrisponde all'aumento di 1 della distanza dalla antecedente colorazione, i coefficienti delle variabili presenti in soluzione sono impostati a -1, mentre per tutti gli altri è previsto il valore 1; se un nodo cambia di colore la relativa variabile (con coefficiente -1) verrebbe azzerata, mentre, dato che è obbligatoria la colorazione di ogni nodo, un'altra variabile con coefficiente 1 passerebbe dal valore 0 a valore 1. Si nota così un incremento di 2 nel computo della distanza: ecco quindi spiegato il motivo per il quale nel calcolo dei due limiti

si raddoppiano i valori dei due estremi dell'intervallo (ogni cambiamento corrisponde ad un doppio incremento) e si sottrae il numero dei nodi del grafo (il valore iniziale della distanza è pari al numero dei nodi col segno negativo).

7.3.1.4 BINARY_CONSTRAINT( )

Questa funzione viene utilizzata per impostare tutte le variabili come variabili binarie.

```
status = CPXchgctype (env,
                      lp,
                      cnt,
                      indices,
                      ctype);
```

FIGURA 7.15 – FUNZIONE CPXCHGCTYPE( )

Le variabili vengono fissate a binarie attraverso l'istruzione *CPXchgctype( )* che prevede gli argomenti di seguito elencati e illustrati in figura 7.15:

- *env* e *lp* fanno riferimento all'ambiente e all'oggetto problema impostati inizialmente;
- *cnt* indica il numero di variabili interessate dalla chiamata della funzione e specifica la lunghezza dei successivi due array;
- i due vettori *indices* e *xctype* che contengono rispettivamente, per ogni variabile, l'indice di colonna e uno dei caratteri della tabella 7.3 che ne indicano la tipologia [3].

CPX_CONTINUOUS	C	make column indices[j] continuous
CPX_BINARY	B	make column indices[j] binary
CPX_INTEGER	I	make column indices[j] general integer
CPX_SEMICONT	S	make column indices[j] semi-continuous
CPX_SEMIINT	N	make column indices[j] semi-integer

TABELLA 7.3 – IMPOSTAZIONI PER XCTYPE

Le variabili devono essere impostate tutte a binarie, quindi il vettore *xctype* conterrà tutti valori "*B*" e il parametro *cnt* assumerà un valore pari al numero dei

nodi moltiplicato per il numero dei colori utilizzati, cioè il numero delle variabili x_{ik}, e sommato al numero dei colori in soluzione, cioè il numero delle variabili y_k.

```
/* funzione che imposta tutte le variabili come binarie */
void binary_constraint ( CPXENVptr env, CPXLPptr lp )
{
    int i;
    int status = 0;
    int num_x = num_nodes * best_solution[num_nodes].color;
    int num_y = best_solution[num_nodes].color;

    int *indices = (int *) calloc ( num_x + num_y, sizeof(int) );
    char *ctype = (char *) calloc ( num_x + num_y, sizeof(char) );

    for ( i = 0; i < num_x + num_y; i++ )
    {
        indices[i] = i;
        ctype[i] = 'B';
    }

    status = CPXchgctype ( env, lp, num_x + num_y, indices, ctype );

}
```

FIGURA 7.16 – FUNZIONE BINARY_CONSTRAINT( )

7.3.2 PROXY ALGORITHM

L'algoritmo proxy si basa sull'ottimizzazione della distanza cui trovare la nuova colorazione, la quale deve essere minimizzata, imponendo l'individuazione di una soluzione con un numero minore di colori di quella attuale [7].

L'algoritmo proxy si può considerare come il ribaltamento della logica dell'algoritmo local branching: nell'algoritmo proxy, infatti, la funzione obiettivo (6.1) e i vincoli (6.3) e (6.4), che definiscono l'intervallo di ricerca, vengono scambiati di ruolo; i due vincoli confluiscono nella nuova funzione obiettivo e ciò che prima era la funzione obiettivo va a formare uno dei vincoli.

Il modello prevede quindi di sostituire la funzione obiettivo (6.1) che minimizzava il numero di colori, con la funzione obiettivo

$$\min \left\{ \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=1} (1 - x_{ik}) + \sum_{i \in V} \sum_{k:\, \bar{x}_{ik}=0} x_{ik} \right\}$$

che esprime la distanza di Hamming della nuova soluzione x da quella precedente

$\bar{x}$, anch'essa da essere minimizzata, e di sostituire i vincoli (6.3) e (6.4) con il nuovo vincolo

$$\sum_{k=1}^{n} y_k \leq z - 1$$

con z numero dei colori nella precedente soluzione, che rappresenta la somma dei colori utilizzati che deve essere diminuita di almeno 1 perché sia apportato un miglioramento nella soluzione.

Il modello risultante, modificato da questa inversione di ruoli, è perciò il modello (6.8)-(6.13) descritto nel paragrafo 6.1. Per i vincoli (6.9) e (6.10) valgono le stesse considerazioni fatte al paragrafo 7.3.1 per l'algoritmo local branching relativamente ai vincoli (6.2) e (6.5).

Di seguito il modello matematico del problema come viene generato da CPLEX nell'apposito file LP.

```
\ENCODING=ISO-8859-1
\Problem name: graph_coloring_problem

Minimize
 obj: x_0_0 + x_0_1 - x_0_2 + x_0_3 + x_1_0 - x_1_1 + x_1_2 + x_1_3 + x_2_0
      - x_2_1 + x_2_2 + x_2_3 + x_3_0 + x_3_1 - x_3_2 + x_3_3 + x_4_0 + x_4_1
      + x_4_2 - x_4_3 - x_5_0 + x_5_1 + x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2
      + x_6_3 + 1000 z + 7 constant
Subject To
 node_0:               x_0_0 + x_0_1 + x_0_2 + x_0_3 >= 1
 ...
 node_6:               x_6_0 + x_6_1 + x_6_2 + x_6_3 >= 1
 constr_0_0:           x_0_0 + x_1_0 - y_0 <= 0
 ...
 constr_12_3:          x_4_3 + x_6_3 - y_3 <= 0
 constr_number_colors: y_0 + y_1 + y_2 + y_3 - z <= 3
Bounds
 0 <= x_0_0 <= 1
 ...
 0 <= x_6_3 <= 1
 0 <= y_0 <= 1
 0 <= y_1 <= 1
 0 <= y_2 <= 1
 0 <= y_3 <= 1
 0 <= z <= 1
      constant = 1
Binaries
 x_0_0  x_0_1  x_0_2  x_0_3  x_1_0  x_1_1  x_1_2  x_1_3  x_2_0  x_2_1  x_2_2
 x_2_3  x_3_0  x_3_1  x_3_2  x_3_3  x_4_0  x_4_1  x_4_2  x_4_3  x_5_0  x_5_1
 x_5_2  x_5_3  x_6_0  x_6_1  x_6_2  x_6_3  y_0  y_1  y_2  y_3
End
```

Per la descrizione del modello matematico del problema che prevede l'utilizzo della logica proxy è stata realizzata la funzione *create_proxy_model( )* che, per generare funzione obiettivo e vincoli, richiama, nell'ordine, le seguenti altre funzioni:

- coloring_vertex_constraint_proxy( )
- adjacency_constraint_proxy( )
- number_colors_constraint( )
- incumbent( )
- binary_constraint( )
- add_constant_n( )

Dato che ad ogni iterazione l'algoritmo esplora l'intero spazio di ricerca, o viene individuata una nuova colorazione o, non individuando miglioramenti possibili nella colorazione, ne dimostra l'ottimalità di quella corrente. Quindi l'algoritmo termina quando, completata un'iterazione non è stato in grado di diminuire ulteriormente il numero di colori necessari per colorare i vertici del grafo.

Di seguito verranno trattate le funzioni suelencate, descrivendo come sono state implementate, indicando quali routine di CPLEX vengono sfruttate e indicandone il ruolo all'interno dell'algoritmo.

7.3.2.1 COLORING_VERTEX_CONSTRAINT_PROXY( )

La funzione *coloring_vertex_constraint_proxy( )* viene utilizzata per imporre nella risoluzione del problema l'obbligo di assegnare ad ogni nodo del grafo almeno un colore e per creare, nel contempo, le variabili $x_{i\ k}$ necessarie per descrivere il vincolo e la funzione obiettivo di cui, sempre nella presente funzione, vengono definiti i coefficienti.

Molto simile, anche nel nome, a quella utilizzata nell'algoritmo local branching, ne è accomunata per la quasi totalità del codice e per il ruolo che ricopre

all'interno dell'algoritmo.

L'unica differenza riscontrabile è imputabile ad una diversa funzione obiettivo che deve essere realizzata: in local branching erano le variabili y_k che andavano a formarla, mentre in proxy sono le variabili $x_{i\,k}$, che rappresentano le associazioni nodo-colore nella soluzione, ad essere utilizzate per la descrizione della funzione obiettivo, dato che sono necessarie per esprimere la distanza di Hamming che l'algoritmo richiede di minimizzare.

Rapportando quanto detto al codice scritto per implementare la funzione, è il vettore *objective*, passato come argomento alla routine *CPXcopylpwnames( )* e contenente i coefficienti delle variabili della funzione obiettivo, l'unico parametro a subire un cambiamento: precedentemente i coefficienti erano impostati tutti a 0 dato che le variabili del vincolo che sta per essere aggiunto al modello non interessavano la funzione obiettivo; in proxy, invece, i coefficienti vengono impostati a -1 o ad 1, rispettivamente se la corrispettiva variabile è in soluzione oppure è assente, secondo quanto indicato dalla definizione di distanza di Hamming utilizzata dall'algoritmo per la ricerca della nuova colorazione.

```
obj_function_coeff[contV] = 0.0;
```

```
if ( k == best_solution[i].color - 1 )
    obj_function_coeff[contV] = -1.0;
else
    obj_function_coeff[contV] = 1.0;
```

FIGURA 7.17 – COLORING_VERTEX_CONSTRAINT( ) DA LOCAL BRANCHING A PROXY

7.3.2.2 ADJACENCY_CONSTRAINT_PROXY( )

La funzione *adjacency_constraint_proxy( )* obbliga l'ottimizzatore, attraverso la generazione del vincolo di adiacenza, ad utilizzare colori differenti per i vertici del grafo che sono collegati da un arco e, allo stesso tempo, vengono create le variabili y_k, necessarie per la definizione del vincolo.

Come per la precedente funzione, sia il nome, sia il codice scritto per la sua implementazione all'interno dell'algoritmo proxy, rimandano ad un

comportamento quasi del tutto uguale a quello della corrispettiva funzione *adjacency_constraint_local_branching( )* per l'algoritmo local branching.

Anche in questo caso, l'unica differenza che si può notare è conseguenza della diversa funzione obiettivo che si va a creare, che deve minimizzare una sommatoria di variabili x_{ik}, per individuare una colorazione il più possibile simile a quella di partenza, e che non interessa più le variabili rappresentanti i colori y_k come avveniva per l'algoritmo local branching.

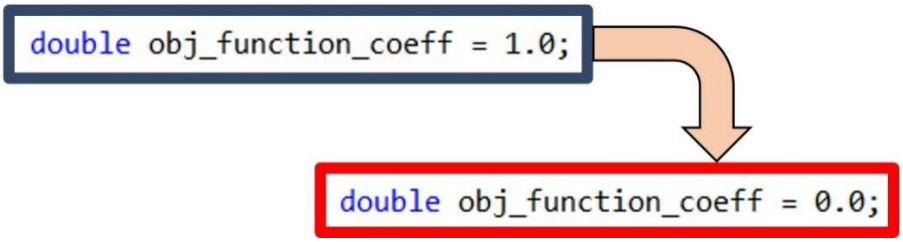


FIGURA 7.18 – ADJACENCY_CONSTRAINT( ) DA LOCAL BRANCHING A PROXY

Ciò si traduce, a livello di codice implementato, nell'impostazione a 0, attraverso il vettore *obj* passato come parametro alla routine *CPXaddcols( )* di CPLEX, dei coefficienti delle variabili y_k all'interno della funzione obiettivo, precedentemente fissati a 1 in local branching.

7.3.2.3 NUMBER_COLORS_CONSTRAINT( )

Questa funzione impone che la nuova colorazione del grafo individuata da CPLEX abbia almeno un colore in meno della colorazione correntemente considerata come la migliore.

Come si può osservare dall'estratto del file LP nel paragrafo 7.3.2, i vincoli per imporre un tale miglioramento sono quelli del tipo:

constr_number_colors: $y\_0 + y\_1 + y\_2 + y\_3 - z <= 3$

Il vincolo viene inserito nel modello attraverso un'unica chiamata della routine *CPXaddrows( )*, come si può vedere dalla figura 7.19:

```
/* funzione che impone la colorazione del grafo con almeno un colore in meno
   della colorazione trovata in precedenza */
void number_colors_constraint ( CPXENVptr env, CPXLPptr lp )
{
    int k;
    int status = 0;

    int num_x = num_nodes * best_solution[num_nodes].color;

    int nonzero_coeff = best_solution[num_nodes].color;
    double righthand_side_coeff = best_solution[num_nodes].color - 1.0;
    char sense = 'L';

    int r_matbeg = 0;
    int *r_matind = (int *) calloc ( best_solution[num_nodes].color, sizeof(int) );
    double *r_matval = (double *) calloc ( best_solution[num_nodes].color, sizeof(double) );

    char *name = (char *) calloc ( 30, sizeof(char) );

    for ( k = 0; k < best_solution[num_nodes].color; k++ )
    {
        r_matind[k] = num_x + k;
        r_matval[k] = 1.0;
    }

    sprintf ( name, "constr_number_colors" );

    status = CPXaddrows ( env, lp, 0, 1, nonzero_coeff, &righthand_side_coeff,
                          &sense, &r_matbeg, r_matind, r_matval, NULL, &name );

}
```

FIGURA 7.19 – FUNZIONE NUMBER_COLORS_CONSTRAINT( )

Il parametro *ccnt* viene impostato a 0, non essendo da inserire nuove colonne, mentre il parametro *rcnt* viene settato ad 1, essendo sufficiente uno solo di tali vincoli. Il numero di coefficienti non nulli è pari al numero di colori utilizzati nella colorazione attuale. Il termine noto è impostato per essere uguale al numero di colori attualmente in uso meno uno, appunto per individuare, se esistente, un miglioramento. Il vettore *sense* imposta un vincolo di tipo minore-uguale, mentre i coefficienti nell'array *rmatval* sono impostati tutti ad uno. Al vincolo è associato il nome *"constr_number_colors"*.

7.3.2.4 INCUMBENT( )

La funzione qui esaminata non aggiunge alcun vincolo al modello, ma la colonna della variabile *z*, variabile presente sia nella funzione obiettivo, sia nel vincolo *"constr_number_colors"* che impone un miglioramento nella colorazione attuale.

La funzione obiettivo, infatti, non contiene solamente la distanza di Hamming definita come detto precedentemente, ma alla sommatoria viene

aggiunto il termine *Mz*, con $z \geq 0$ variabile continua e con *M* valore intero positivo molto grande, se comparato con i possibili valori che la funzione obiettivo può assumere.

La stessa variabile *z* è sottratta dalla sommatoria delle variabili y_k nel vincolo aggiunto dalla funzione *number_colors_constraint( )*.

In figura 7.20 si possono notare, cerchiati in rosso, i termini aggiunti dalla presente istruzione nella funzione obiettivo e nel vincolo suddetti:

```
\ENCODING=ISO-8859-1
\Problem name: graph_coloring_problem

Minimize
 obj: x_0_0 + x_0_1 - x_0_2 + x_0_3 + x_1_0 - x_1_1 + x_1_2 + x_1_3 + x_2_0
      - x_2_1 + x_2_2 + x_2_3 + x_3_0 + x_3_1 - x_3_2 + x_3_3 + x_4_0 + x_4_1
      + x_4_2 - x_4_3 - x_5_0 + x_5_1 + x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2
      + x_6_3 + 1000 z + 7 constant
Subject To
 ...
 constr_number_colors: y_0 + y_1 + y_2 + y_3 - z <= 3
 ...
```

FIGURA 7.20 – VARIABILE Z

Il termine *Mz* inserito nella funzione obiettivo permette di poter usufruire dell'euristico RINS (Relaxation Induced Neighborhood Search) durante il processo di ottimizzazione di CPLEX.

Il ruolo della variabile *z* è quello di consentire di non avere tutto il miglioramento richiesto, ma solo una parte di esso, passando attraverso soluzioni parziali che non apportano miglioramenti alla soluzione di partenza. Questo meccanismo, però, porta ad avere un coefficiente di valore elevato nella funzione obiettivo, rappresentato da *M*, per cui il solver, dovendo minimizzare la funzione obiettivo visibile in figura 7.20, sarà portato a cercare soluzioni, se esistono, nelle quali $z = 0$ e a completare come può la soluzione parziale. Il reale vantaggio che motiva l'introduzione della variabile è che la soluzione di riferimento è una soluzione ammissibile se *z* viene posta ad 1 e ciò permette a CPLEX di avere sempre a disposizione una soluzione ammissibile, eventualmente di costo elevato e, nella fattispecie, esisterebbe sempre almeno la colorazione individuata in precedenza su cui ora ci si è centrati, che permette ad alcuni euristici, tra cui RINS, di funzionare. Incumbent, infatti, che dà il nome alla funzione, fa proprio riferimento alla miglior soluzione intera ammissibile conosciuta da CPLEX durante l'ottimizzazione [7].

La colonna della variabile *z* viene aggiunta con un'unica chiamata alla routine *CPXaddcols( )* come si può vedere dalla figura 7.21:

```
/* funzione che aggiunge la colonna della variabile "z" che comparirà
   nel vincolo del numero di colori e nella funzione obiettivo */
void incumbent ( CPXENVptr env, CPXLPptr lp )
{
    int status = 0;

    int num_z = 1;
    double bigM = 1000.0;

    int c_matbeg = 0;
    int c_matind[1];
    double c_matval[1];

    double lb = 0.0;
    double ub = 1.0;

    char *name = "z";

    int indices[1];
    char ctype[1];

    c_matind[0] = num_nodes + num_edges * best_solution[num_nodes].color;
    c_matval[0] = -1.0;

    status = CPXaddcols ( env, lp, 1, num_z, &bigM, &c_matbeg, c_matind, c_matval, &lb, &ub, &name );

    ctype[0] = 'C';
    indices[0] = num_nodes * best_solution[num_nodes].color + best_solution[num_nodes].color;

    status = CPXchgctype ( env, lp, num_z, indices, ctype );
}
```

FIGURA 7.21 – FUNZIONE INCUMBENT( )

Il numero di colonne aggiunte è impostato ad 1, così come il numero di coefficienti non nulli inseriti nella matrice dei vincoli. Il valore del coefficiente *M* della variabile *z* all'interno della funzione obiettivo è stato impostato a 1000, valore che si è ritenuto elevato rispetto i possibili valori che la funzione obiettivo può assumere. I due array *cmatind* e *cmatval* sono stati predisposti, rispettivamente, per memorizzare l'indice di riga che individua il vincolo *"constr_number_colors"* cui deve essere sottratta la variabile *z* e per fissare il coefficiente che la stessa variabile assume all'interno del vincolo. Lower e upper bound sono stati fissati pari a 0 e 1.

Successivamente, attraverso l'istruzione *CPXchgctype( )*, la variabile *z* viene impostata a continua, come si può vedere sempre dalla figura 7.21.

7.3.2.4.1 L'EURISTICO RINS

Nell'esplorazione di un albero di ricerca, generalmente due soluzioni sono a disposizione: la soluzione incumbent che è ammissibile rispetto al vincolo di interezza, ma non è ottima finché l'ottimizzazione non ha termine e la soluzione del rilassamento continuo del nodo corrente che molto spesso non è intera, ma che

è caratterizzata da un valore migliore di quello della soluzione incumbent. Quindi, la soluzione incumbent e quella del rilassamento continuo raggiungono uno dei due seguenti obiettivi e falliscono nella realizzazione dell'altro: interezza e ottimalità del valore della funzione obiettivo.

Mentre alcune variabili chiaramente assumono valori differenti nella soluzione incumbent e in quella del rilassamento continuo, è importante notare che molte hanno lo stesso valore. RINS è basato sull'intuizione che l'istanziamento di queste variabili forma una soluzione parziale che probabilmente guiderà il solutore verso il raggiungimento di una soluzione completa caratterizzata da entrambe gli obiettivi visti in precedenza. Perciò, l'euristico focalizza la propria attenzione su quelle variabili che differiscono nel rilassamento continuo e nella soluzione incumbent, che intuitivamente sono quelle che meritano maggiori attenzioni.

La forza di RINS sta nell'esplorazione di un intorno di entrambe le soluzioni, incumbent e del rilassamento continuo. L'ottimizzatore non conosce quali delle due soluzioni sia la più vicina alla soluzione ottima intera, e, in RINS, esse ricoprono ruoli perfettamente simmetrici, così se una è di scarsa qualità, l'altra automaticamente permette di definire un intorno più promettente, e viceversa. Perciò, se da un lato, RINS può apportare forti miglioramenti a soluzioni incumbent di limitata qualità perché guidato dal rilassamento continuo, d'altro canto, RINS, probabilmente può incrementare la robustezza di un rilassamento troppo lasco perché guidato dalla soluzione incumbent.

RINS formula l'esplorazione dell'intorno come un altro MIP, cui ci si riferisce con il nome di sub-MIP. Il sotto-problema, potenzialmente, può essere ampio e difficile da risolvere, pertanto la sua esplorazione deve essere molto spesso troncata. Questo avviene limitando il numero di nodi esplorato nell'albero di ricerca che ha origine dal nuovo problema [4].

In CPLEX sono due i parametri che interessano l'algoritmo: RINSHEUR per l'impostazione dell'intervallo di nodi da esplorare prima di una sua nuova attivazione e SUBMIPNODELIM che limita il numero di nodi esplorati quando CPLEX risolve un sub-MIP.

7.3.2.5 BINARY_CONSTRAINT( )

La funzione è utilizzata solamente per definire la tipologia binaria delle variabili utilizzate nel modello.

Anche in questo caso, la funzione è la medesima che si è utilizzata per l'algoritmo local branching e quindi per una sua descrizione più esauriente si rimanda al paragrafo 7.3.1.4.

7.3.2.6 ADD_CONSTANT_N( )

Questa funzione è stata implementata con lo scopo di introdurre una nuova variabile nel modello descrittivo.

La nuova variabile, denominata *constant* nel modello, è una variabile fittizia che tiene conto del numero di nodi del grafo all'interno della funzione obiettivo.

Infatti, nella definizione della distanza di Hamming nella funzione obiettivo, c'è un termine costante che, senza questa variabile fittizia, non verrebbe ottimizzato e che rappresenta il numero di variabili che hanno assunto valore 1 nella soluzione precedente; il numero di queste variabili è proprio pari al numero dei nodi.

In figura 7.22 è stata evidenziata in rosso la variabile generata dalla funzione *add_constant_n( )*.

```
\ENCODING=ISO-8859-1
\Problem name: graph_coloring_problem

Minimize
 obj: x_0_0 + x_0_1 - x_0_2 + x_0_3 + x_1_0 - x_1_1 + x_1_2 + x_1_3 + x_2_0
      - x_2_1 + x_2_2 + x_2_3 + x_3_0 + x_3_1 - x_3_2 + x_3_3 + x_4_0 + x_4_1
      + x_4_2 - x_4_3 - x_5_0 + x_5_1 + x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2
      + x_6_3 + 1000 z + 7 constant
...
```

FIGURA 7.22 – VARIABILE CONSTANT

Se ad esempio viene considerato un grafo con 10 nodi, dieci tra le variabili $x_{i\,k}$ della funzione obiettivo sono impostate ad 1 con coefficiente -1 ottenendo un valore iniziale uguale a -10; se vengono cambiati i colori di due nodi il divario nel valore della funzione obiettivo è di 4 e il valore di partenza risulta essere modificato

in -6, infatti $(-6 + 10)/2 = 2$, come due sono i nodi interessati da una modifica. Se però viene aggiunta la nuova variabile *constant*, il valore iniziale viene azzerato e con il cambiamento nella colorazione di due nodi si modifica il valore della funzione obiettivo a 4, ottenendo immediatamente $4/2 = 2$ variabili totali modificate.

Per modellare questo termine è stata introdotta la variabile fittizia *constant* nella funzione obiettivo e non inserita in alcun altro vincolo, con coefficiente pari al numero dei vertici del grafo e limiti inferiore e superiore impostati entrambe a 1, per fissare il suo valore anch'esso ad 1; questo spiega sia il nome scelto, cioè *constant*, sia il motivo per cui è stata definita più volte fittizia: perché nella realtà ha valore costante.

La colonna della variabile *constant* viene inserita nel modello richiamando un'unica volta la funzione *CPXaddcols( )* di CPLEX, con parametri impostati come evidenziato dalla figura 7.23.

```
/* funzione che aggiunge una costante alla funzione obiettivo
   che tiene conto del numero di nodi del grafo */
void add_constant_n ( CPXENVptr env, CPXLPptr lp )
{
    int status = 0;

    double n = (double)num_nodes;
    double lb = 1.0;
    double ub = 1.0;

    char *name = "constant";

    status = CPXaddcols ( env, lp, 1, 0, &n, NULL, NULL, NULL,
                          &lb, &ub, &name );

    if ( status )
        cplex_error ( status, "CPXaddcols" );
}
```

FIGURA 7.23 – FUNZIONE ADD_CONSTANT_N( )

Ccnt è impostato ad 1 in quanto viene aggiunta solamente una nuova colonna, mentre *nzcnt* è impostato a 0 dato che la variabile non è inserita in nessun'altra riga della matrice dei vincoli; questo motiva anche l'impostazione a NULL dei tre array *cmatbeg*, *cmatind*, *cmatval*. Infine, come già indirettamente detto, *obj* è impostato per essere pari al numero di nodi, *lb* e *ub* sono fissati entrambe ad 1 e *colname* è stato valorizzato con la stringa *constant*.

7.3.3 HYBRID ALGORITHM

L'algoritmo hybrid nasce dall'unione delle logiche dei due precedenti algoritmi, basandosi sulla minimizzazione della distanza della nuova colorazione da quella corrente con l'obbligo di diminuire i colori utilizzati in precedenza di almeno un'unità (funzione obiettivo e vincolo tipici dell'algoritmo proxy) e imponendo, inoltre, come vincoli, i limiti inferiore e superiore dell'intervallo entro cui deve avvenire la ricerca (vincoli propri dell'algoritmo local branching).

Il modello matematico dell'algoritmo ibrido è quindi il modello (6.14)-(6.21) descritto in precedenza nel paragrafo 6.1. Per i vincoli (6.15) e (6.16) valgono come sempre le stesse considerazioni del paragrafo 7.3.1 fatte in relazione all'algoritmo local branching per i vincoli (6.2) e (6.5).

Di seguito viene riportato un estratto del file LP così come viene generato dal solver CPLEX.

```
\ENCODING=ISO-8859-1
\Problem name: graph_coloring_problem

Minimize
 obj: x_0_0 - x_0_1 + x_0_2 + x_0_3 + x_1_0 + x_1_1 + x_1_2 - x_1_3 - x_2_0
      + x_2_1 + x_2_2 + x_2_3 + x_3_0 - x_3_1 + x_3_2 + x_3_3 + x_4_0 + x_4_1
      + x_4_2 - x_4_3 + x_5_0 + x_5_1 - x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2
      + x_6_3 + 1000 z + 7 constant
Subject To
 node_0:               x_0_0 + x_0_1 + x_0_2 + x_0_3 >= 1
 ...
 node_6:               x_6_0 + x_6_1 + x_6_2 + x_6_3 >= 1
 constr_0_0:           x_0_0 + x_1_0 - y_0 <= 0
 ...
 constr_12_3:          x_4_3 + x_6_3 - y_3 <= 0
 constr_number_colors: y_0 + y_1 + y_2 + y_3 - z <= 3
 constr_proximity_inf: x_0_0 - x_0_1 + x_0_2 + x_0_3 + x_1_0 + x_1_1 + x_1_2
                       - x_1_3 - x_2_0 + x_2_1 + x_2_2 + x_2_3 + x_3_0 - x_3_1
                       + x_3_2 + x_3_3 + x_4_0 + x_4_1 + x_4_2 - x_4_3 + x_5_0
                       + x_5_1 - x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2 + x_6_3
                       >= -7
 constr_proximity_sup: x_0_0 - x_0_1 + x_0_2 + x_0_3 + x_1_0 + x_1_1 + x_1_2
                       - x_1_3 - x_2_0 + x_2_1 + x_2_2 + x_2_3 + x_3_0 - x_3_1
                       + x_3_2 + x_3_3 + x_4_0 + x_4_1 + x_4_2 - x_4_3 + x_5_0
                       + x_5_1 - x_5_2 + x_5_3 - x_6_0 + x_6_1 + x_6_2 + x_6_3
                       <= 13
Bounds
 0 <= x_0_0 <= 1
 ...
 0 <= x_6_3 <= 1
```

```
 0 <= y_0 <= 1
 0 <= y_1 <= 1
 0 <= y_2 <= 1
 0 <= y_3 <= 1
 0 <= z <= 1
      constant = 1
Binaries
 x_0_0  x_0_1  x_0_2  x_0_3  x_1_0  x_1_1  x_1_2  x_1_3  x_2_0  x_2_1  x_2_2
 x_2_3  x_3_0  x_3_1  x_3_2  x_3_3  x_4_0  x_4_1  x_4_2  x_4_3  x_5_0  x_5_1
 x_5_2  x_5_3  x_6_0  x_6_1  x_6_2  x_6_3  y_0  y_1  y_2  y_3
End
```

In sintesi, si potrebbe definire l'algoritmo hybrid come l'algoritmo proxy in cui la nuova colorazione non viene ricercata esplorando l'intero spazio delle soluzioni, ma viene limitata, per ogni iterazione, ad uno specifico intervallo.

Per la generazione del modello descrittivo per l'algoritmo hybrid è stata implementata la funzione *create_hybrid_model( )* che, per definire la funzione obiettivo e la matrice dei vincoli, utilizza, nell'ordine, le seguenti altre funzioni:

- coloring_vertex_constraint_proxy( )
- adjacency_constraint_proxy( )
- number_colors_constraint( )
- incumbent( )
- proximity_constraint_with_interval( )
- binary_constraint( )
- add_constant_n( )

Dato che per ogni iterazione, l'algoritmo esamina solamente una porzione dell'intero spazio di ricerca, per dimostrare l'ottimalità di una determinata colorazione individuata, dovranno essere esplorati tutti gli intervalli possibili. Quindi come per l'algoritmo local branching, l'individuazione in un intervallo di una soluzione migliorante provoca la re-inizializzazione dei limiti di ricerca, mentre l'impossibilità di un tale evento determina l'aggiornamento dell'intervallo da esplorare. L'algoritmo termina quando viene esplorato anche l'ultimo intervallo disponibile, cioè quello con limite superiore pari al numero di vertici del grafo.

Nei paragrafi a seguire, per ognuna delle funzioni richiamate da *create_hybrid_model( )*, verrà riportata solamente una breve descrizione del ruolo

ricoperto e indicati i vincoli introdotti, in quanto, essendo le medesime funzioni utilizzate per local branching e proxy, esse sono state già ampiamente trattate nel dettaglio nei capitoli relativi agli altri due algoritmi.

7.3.3.1 COLORING_VERTEX_CONSTRAINT_PROXY( )

La funzione *coloring_vertex_constraint_proxy( )* si occupa della creazione delle variabili $x_{i\,k}$ che rappresentano l'associazione nodo-colore ed è utilizzata per imporre l'assegnazione di un colore ad ogni nodo e per la realizzazione della funzione obiettivo secondo la definizione di distanza di Hamming.

Per una trattazione più completa si rimanda al paragrafo 7.3.2.1.

7.3.3.2 ADJACENCY_CONSTRAINT_PROXY( )

Con questa funzione vengono introdotti i vincoli di adiacenza che vincolano CPLEX ad individuare soluzioni in cui due vertici che condividono uno stesso arco nel grafo in esame non si vedano assegnato lo stesso colore. Vengono anche create le variabili y_k che rappresentano i colori presenti in soluzione.

Per maggiori dettagli sul ruolo dei vincoli e sulle impostazioni utilizzate per il loro inserimento nel modello si vedano i paragrafi 7.3.2.2.

7.3.3.3 NUMBER_COLOR_CONSTRAINT( )

Questa funzione impone il vincolo di miglioramento della colorazione corrente, nel senso di diminuzione del numero di colori utilizzato.

Per una descrizione più particolareggiata si veda il paragrafo 7.3.2.3.

7.3.3.4 INCUMBENT( )

La funzione *incumbent( )* genera la colonna della nuova variabile *z*,

inserendola con l'opportuno coefficiente nella funzione obiettivo e nel vincolo *constr_number_colors* che obbliga CPLEX ad individuare una soluzione con almeno un colore in meno di quella corrente.

La variabile *z* è quella che permette l'attivazione dell'algoritmo RINS all'interno dell'esecuzione di CPLEX.

Per comprendere meglio il ruolo della funzione e della variabile *z* nell'attivazione dell'euristico RINS si rimanda al paragrafo 7.3.2.4.

7.3.3.5 PROXIMITY_CONSTRAINT_WITH_INTERVAL( )

La presente funzione è il fulcro della logica dell'algoritmo local branching, infatti è attraverso i due vincoli di prossimità da essa introdotti nel modello del problema che si limita la distanza entro cui deve essere individuata la nuova soluzione e quindi si riduce l'intervallo di ricerca da esplorare.

Per ulteriori dettagli e una descrizione più completa, sia di come sono strutturati i due vincoli per i limiti inferiore e superiore, sia per quanto concerne l'implementazione della funzione si rimanda al paragrafo 7.3.1.3.

7.3.3.6 BINARY_CONSTRAINT( )

Questa funzione è stata realizzata per impostare la tipologia delle variabili impiegate nella definizione del problema e nella descrizione del modello matematico. Nella fattispecie *binary_constraint( )* fissa tutte le variabili come binarie.

Per una descrizione completa della funzione si veda il paragrafo 7.3.1.4.

7.3.3.7 ADD_CONSTANT_N( )

La funzione *add_constant_n( )* aggiunge la variabile fittizia *constant* al modello descrittivo del problema. La variabile viene definita fittizia in quanto assume un valore costante e serve per tenere conto del numero di nodi di cui si

compone il grafo nella minimizzazione della funzione obiettivo. In sua assenza, infatti, nella funzione obiettivo comparirebbe un termine costante non ottimizzato pari al numero di vertici.

Per maggiori particolari si rimanda al paragrafo 7.3.2.6.

CAPITOLO 8

TEST E RISULTATI

8.1 RISULTATI SPERIMENTALI

In questo capitolo verranno presentati i risultati derivanti dai test effettuati sui tre diversi algoritmi che sono stati implementati durante il lavoro presentato in questa tesi.

Come già detto al capitolo 4, i grafi che sono stati presi in esame per verificare la correttezza e l'efficienza dei tre algoritmi euristici rappresentano tutti istanze proposte dal DIMACS, descritti ovviamente nel relativo formato standard, e sono stati reperiti sul sito "*http://mat.gsia.cmu.edu/COLOR/instances.html"* [2].

Le istanze considerate sono state 56 e in particolare:

- le istanze del tipo *"fpsol2.i.x"*, *"inithx.i.x"*, *"mulsol.i.x"* e *"zeroin.i.x"* descrivono problemi di colorazione generati da problemi di register allocation relativi a parti di codice reali;
- le istanze del tipo *"le450_x"* sono grafi di Leighton e rappresentano problemi di schedulazione di grande entità;
- le istanze del tipo *"school1"* sono grafi derivanti da problemi di class scheduling;
- le istanze *"anna"*, *"david"*, *"homer"*, *"huck"* e *"jean"* sono book graph: data un'opera letteraria viene creato un grafo dove ogni nodo rappresenta un

personaggio e due nodi sono connessi da un lato se i corrispondenti personaggi si incontrano nel libro; *"anna"* sta per Anna Karenina di Tolstoy, *"david"* per David Copperfield di Dickens, *"homer"* per l'Iliade di Omero, *"huck"* per Huckleberry Finn di Twain e *"jean"* per I miserabili di Hugo);

- l'istanza *"games120"* rappresenta le partite di football giocate nella stagione 1990 dai college americani: i nodi rappresentano le diverse squadre e due squadre sono collegate da un arco se durante la stagione queste si sono dovute scontrare durante una partita;
- le istanze del tipo *"milesx"* sono costituite da nodi che rappresentano punti nello spazio e due nodi sono collegati tra loro se i due punti sono abbastanza vicini tra loro. Questi non sono grafi random, ma rappresentano ad esempio le città degli Stati Uniti e le distanze sono quelle reali ricavate dalle mappe stradali;
- le istanze del tipo *"queenx_x"* descrivono grafi con n^2 nodi che rappresentano quadrati su di una scacchiera. Due nodi sono connessi da un arco se i corrispettivi quadrati sono nella stessa riga, colonna o diagonale. Colorare questi grafi significa dare una risposta al famoso problema delle regine: *"data una scacchiera $n \times n$, è possibile collocare n regine in modo tale che nessuna regina possa essere mangiata da un'altra?"* La risposta è sì se e solo se il grafo ha una colorazione ammissibile con numero di colori utilizzato pari a n;
- le istanze del tipo *"mycielx"* sono grafi di Mycielski: grafi privi di triangoli, ma con numero di colori crescente con la taglia del problema.

I nomi delle istanze cui fanno riferimento i risultati ottenuti sono riportati nella prima colonna della tabella 8.1 e, per ognuna di esse, è stato indicato nelle successive due colonne il numero di nodi e di archi di cui è composto il relativo grafo che la descrivono.

La colonna z^* riporta per ognuno dei grafi la soluzione ottima individuata, cioè il numero di colori utilizzato dalla colorazione ottima del grafo, così come indicato nel sito di riferimento summenzionato e il caso in cui il problema non è ancora stato risolto all'ottimo viene indicato con un simbolo "-".

Successivamente la quinta colonna, quella con la soluzione greedy, indica la soluzione iniziale da cui ha inizio l'intero processo di ottimizzazione di ognuno dei tre algoritmi euristici che è stata ottenuta dall'esecuzione del relativo algoritmo descritto al paragrafo 7.2.3 reiterato per 1000 volte. Più nel dettaglio è stata ricavata verificando mediamente quale soluzione individuasse l'algoritmo greedy per lo specifico grafo cui fa riferimento.

Infine, le ultime tre colonne riportano i risultati ottenuti dall'esecuzione del corrispettivo algoritmo euristico cui fa riferimento.

Se nelle tre colonne relative alle soluzioni restituite dai tre algoritmi euristici sono presenti dei simboli "-" significa che l'algoritmo greedy non solo ha calcolato la soluzione iniziale, ma ha anche già individuato la soluzione ottima senza richiedere l'intervento degli altri tre algoritmi.

Un simbolo "*" accanto alla soluzione trovata indica che l'algoritmo ha terminato la sua esecuzione ed è stato in grado di individuare la soluzione ottima dimostrandone altresì l'ottimalità.

Se, in caso contrario, l'algoritmo non è stato in grado di dimostrare l'ottimalità della soluzione individuata, significa che non ha neanche terminato la sua esecuzione in modo corretto; le motivazioni sono imputabili a due casistiche:

- *errore out of memory* (indicato con *"out"* in tabella) restituito da CPLEX in quanto le risorse messe a disposizione per l'ottimizzazione del problema da parte delle macchine utilizzate non sono state sufficienti;
- l'algoritmo, non avendo apportato ulteriori miglioramenti alla soluzione individuata, dopo un prefissato intervallo di tempo (i risultati in tabella fanno riferimento al tempo di una settimana), è stato terminato lato utente e ciò viene denotato con l'indicazione *"time"*.

Per quanto riguarda i due algoritmi local branching e hybrid, per i quali è prevista l'impostazione di un intorno di ricerca ben preciso, si è deciso di utilizzare un'ampiezza dell'intervallo pari a 10.

Tutti i test sono stati effettuati attraverso il server Blade DELL M1000 messo a disposizione dal centro di calcolo del Dipartimento di Ingegneria

dell'Informazione dell'Università degli Studi di Padova. Il cluster è costituito da 14 lame di calcolo, modello M600, ognuna delle quali con 2 processori quad core Intel Xeon E5450 (12M Cache, 3.00 GHz, 1333 MHz FSB), 16GB di RAM, 2 HD da 72GB in configurazione RAID-1 (mirror); le lame sono interconnesse da uno switch gigabit ethernet.

Di seguito la tabella 8.1 con i risultati dei test effettuati relativamente ai tre algoritmi euristici implementati.

Istanza	#nodi	#archi	z*	Greedy	Local Branching	Proxy	Hybrid
fpsol1.i.1	496	11564	65	65	-	-	-
fpsol2.i.1	451	8691	30	30	-	-	-
fpsol3.i.1	425	8688	30	30	-	-	-
inithx.i.1	864	18707	54	54	-	-	-
inithx.i.2	654	13979	31	31	-	-	-
inithx.i.3	621	13969	31	31	-	-	-
le450_15b	450	8169	15	20	17(time)	15(out)	17(time)
le450_15c	450	16680	15	30	26(time)	23(out)	25(out)
le450_15d	450	16750	15	30	26(time)	24(time)	25(out)
le450_25a	450	8260	25	27	25*	25*	25*
le450_25b	450	8263	25	27	25*	25*	25*
le450_25c	450	17343	25	36	31(time)	29(out)	31(time)
le450_25d	450	17425	25	34	31(time)	29(time)	32(time)
le450_5a	450	5714	5	13	10(time)	5*	10(time)
le450_5b	450	5734	5	12	10(time)	5*	10(time)
le450_5c	450	9803	5	16	5*	5*	5*
le450_5d	450	9757	5	15	5*	5*	5*
mulsol.i.1	197	3925	49	49	-	-	-
mulsol.i.2	188	3885	31	31	-	-	-
mulsol.i.3	184	3916	31	31	-	-	-
mulsol.i.4	185	3946	31	31	-	-	-
mulsol.i.5	186	3973	31	31	-	-	-
school1	385	19095	14	42	30(time)	14*	30(time)
school1_nsh	352	14612	14	38	28(time)	14*	23(out)
zeroin.i.1	211	4100	49	49	-	-	-
zeroin.i.2	211	3541	30	30	-	-	-
zeroin.i.3	206	3540	30	30	-	-	-
anna	138	986	11	11	-	-	-
david	87	812	11	11	-	-	-
homer	561	3258	13	13	-	-	-
huck	74	301	11	11	-	-	-
jean	80	254	10	10	-	-	-
games120	120	1276	9	9	-	-	-
miles1000	128	6432	42	44	42*	42*	42*

miles1500	128	10396	73	73	-	-	-
miles250	128	774	8	9	8*	8*	8*
miles500	128	2340	20	22	20*	20*	20*
miles750	128	4226	31	32	31*	31*	31*
queen10_10	100	2940	-	15	12(time)	12(out)	12(out)
queen11_11	121	3960	11	17	13(out)	13(out)	13(out)
queen12_12	144	5192	-	18	14(time)	14(time)	14(time)
queen13_13	169	6656	13	19	15(out)	15(out)	15(out)
queen14_14	196	8372	-	20	17(out)	17(out)	17(out)
queen15_15	225	10360	-	22	18(time)	18(time)	18(time)
queen16_16	256	12640	-	22	19(time)	19(time)	19(out)
queen5_5	25	320	5	7	5*	5*	5*
queen6_6	36	580	7	9	7*	7*	7*
queen7_7	49	952	7	10	7*	7*	7*
queen8_12	96	2736	12	15	12*	12*	12*
queen8_8	64	1456	9	13	9*	9(out)	9(time)
queen9_9	81	2112	10	14	10(time)	10(time)	10(out)
myciel3	11	20	4	4	-	-	-
myciel4	23	71	5	5	-	-	-
myciel5	47	236	6	6	-	-	-
myciel6	95	755	7	7	-	-	-
myciel7	191	2360	8	8	-	-	-

TABELLA 8.1 – RISULTATI DEI TEST EFFETTUATI

Nella tabella non sono evidenziate le tempistiche di esecuzione dei tre algoritmi euristici che permetterebbero un ulteriore confronto in merito all'efficienza nei tre diversi casi. Questo in quanto i test sono stati eseguiti su macchine non dedicate, in concorrenza tra loro e con test relativi ad altri lavori, rendendo imprecise le informazioni relative ai tempi impiegati.

CAPITOLO 9

CONCLUSIONI

9.1 CONCLUSIONI E PROSPETTIVE FUTURE

Con la presente tesi si è voluto affrontare il problema della colorazione dei grafi, un noto problema NP-hard e quindi difficile da risolvere in modo esatto se non per istanze di modesta taglia. Tuttavia l'utilizzo di strategie, logiche e algoritmi di tipo euristico permettono, se non proprio di individuare la soluzione ottima, almeno di esibire soluzioni di buona qualità che si avvicinino a quella ottima e soprattutto in tempi più rapidi e per istanze anche di maggior entità [13].

Lo strumento che si è deciso di utilizzare per affrontare questo problema è stato il paradigma della programmazione mista-intera (Mixed-Integer Programming – MIP) che è un potente strumento per modellare e risolvere problemi d'ottimizzazione complessi [4].

Per quanto riguarda invece la strategia euristica che si è andati ad integrare negli algoritmi realizzati si è deciso di utilizzare la logica local search, la quale permette di disporre di soluzioni di eccellente qualità e soprattutto in tempi brevi per molte tipologie di problemi particolarmente complessi. La velocità nell'individuare nuovi miglioramenti è resa possibile dalla limitazione dello spazio

di ricerca ad intorni prossimi della soluzione corrente, quindi attraverso l'introduzione del concetto di vicinanza tra soluzioni [1]. Per definire meglio questo concetto di vicinanza, in questo lavoro, si è utilizzata la nozione di distanza di Hamming tra due colorazioni che predilige nuove soluzioni che apportano il minor numero possibile di cambiamenti alla soluzione attuale, intesi come cambiamento di colore ad un vertice del grafo in esame [7].

Utilizzando l'approccio della ricerca locale si sono realizzati tre diversi algoritmi euristici: l'algoritmo local branching, l'algoritmo proxy e infine l'algoritmo hybrid.

Il modello dell'algoritmo local branching richiede come funzione obiettivo che sia minimizzato il numero di colori utilizzato in soluzione, andando ad imporre come vincolo di individuare un miglioramento che sia il più vicino possibile alla colorazione corrente. Qui la ricerca locale è mirata a velocizzare il miglioramento all'inizio del processo d'ottimizzazione quando, verosimilmente, la maggior disponibilità di soluzioni, dovuta ad un maggior numero di colori che è possibile utilizzare, rende più probabile individuare una soluzione migliorante nell'intorno esplorato. Tuttavia, mano a mano che il processo di ottimizzazione si avvicina alla soluzione ottima, diminuiscono le probabilità di trovare una soluzione che sia prossima a quella corrente ed individuare una soluzione molto distante può essere oneroso in termini di tempo, considerando inoltre che lo spazio di ricerca è più esteso. Anche la dimostrazione dell'ottimalità di una soluzione non è cosa facile da parte di una strategia che utilizzi il principio di prossimità tra soluzioni, richiedendo la verifica sull'intero spazio di ricerca [1, 6, 7].

La logica alla base dell'algoritmo proxy prevede, invece, la minimizzazione della distanza entro cui individuare la nuova soluzione nel tentativo di semplificare l'esplorazione dello spazio di ricerca, imponendo come vincolo la riduzione del numero di colori rispetto la colorazione precedente. La tolleranza per il miglioramento richiesto è stata impostata ad 1, perché ci si aspetta in tal modo d'avere una serie di sotto-problemi semplici da risolvere, ognuno dei quali porta a piccoli miglioramenti della soluzione. La logica dell'algoritmo proxy può essere considerata come il ribaltamento della logica local branching, infatti vi si può

notare lo scambio di ruoli tra la funzione obiettivo e i due vincoli che definiscono l'intervallo d'esplorazione corrente [7].

Infine l'algoritmo hybrid è stato realizzato come unione delle logiche dei due algoritmi local branching e proxy con l'obiettivo di riunire in un unico algoritmo gli aspetti positivi dei due precedenti, cioè la velocità d'esplorazione derivante dalla limitazione dello spazio di ricerca e la semplificazione d'esplorazione derivante dalla modifica della funzione obiettivo.

Come visto nel capitolo precedente sono state prese in considerazione 56 istanze.

In 26 casi le istanze considerate sono state risolte solamente con l'utilizzo dell'algoritmo greedy deputato ad individuare la soluzione iniziale e quindi non si è reso necessario chiamare in causa le strategie euristiche degli algoritmi implementati. Probabilmente ciò è dovuto ad una certa semplicità riscontrabile nella struttura del grafo.

In 12 casi tutti e tre gli algoritmi scritti sono stati in grado di migliorare la soluzione iniziale proposta dall'algoritmo greedy fino ad individuare la soluzione ottima e a dimostrarne l'ottimalità.

A prima vista si può notare subito come l'algoritmo proxy surclassi sia l'algoritmo local branching, sia l'algoritmo hybrid. Nella totalità dei casi, infatti, o eguaglia i risultati ottenuti dagli altri due algoritmi o riesce ad individuare colorazioni con un minor numero di colori utilizzati. Questo dimostra la validità della strategia utilizzata per semplificare l'esplorazione dello spazio di ricerca, ponendo mano alla funzione obiettivo di un generico problema di colorazione di un grafo. Infatti, nell'algoritmo proxy, anziché richiedere di minimizzare il numero di colori utilizzati in soluzione, si sostituisce la funzione originale con un'altra che penalizza le soluzioni individuate in termini di distanza di Hamming da quella precedente. L'obiettivo di ridurre il numero di variabili in soluzione viene quindi inserito tra i vincoli del modello descrittivo.

Per quanto concerne, invece, l'algoritmo local branching e l'algoritmo hybrid, si possono constatare risultati ottenuti pressoché uguali. Entrambi sono caratterizzati dai vincoli che definiscono l'intervallo d'esplorazione corrente al quale è limitata la ricerca di una nuova soluzione migliorante e quindi un miglior

comportamento potrebbe essere ricercabile attraverso diverse impostazioni del parametro *k* che va a definire l'ampiezza dell'intorno esplorato.

Considerando che i test degli algoritmi euristici implementati sono stati eseguiti su macchine sulle quali erano in esecuzione in modo concorrenziale anche altri test relativi ad altri lavori, le risorse a disposizione per questo progetto erano limitate e ciò spesso ha portato a situazioni in cui l'ottimizzatore restituiva errori del tipo out of memory o in cui si è stati costretti ad impostare un time limit. Per migliorare i risultati ottenuti riportati in tabella, sicuramente potrebbe essere di una qualche utilità avere a disposizione macchine dedicate, su cui eseguire unicamente i test di questi algoritmi euristici. Si potrebbe così disporre di una maggior priorità per ciascun test, che assicurerebbe un conseguente incremento della velocità del processo di ottimizzazione che, in ultima istanza, diminuirebbe i casi in cui il time limit viene raggiunto. Macchine dedicate assicurerebbero anche una maggior disponibilità di memoria dedicata per il singolo processo che in questo caso eviterebbero, fatta eccezione per istanze di grande dimensione e particolarmente complesse, i casi in cui l'ottimizzatore termina con un errore per mancanza di memoria sufficiente.

Importante per riuscire a migliorare i risultati ottenuti è anche individuare la più corretta impostazione di tutti quei parametri che sono legati al funzionamento dei tre algoritmi, sia a livello di descrizione del modello matematico da sottoporre all'ottimizzatore per la sua risoluzione, sia a livello di parametri per l'indicazione del comportamento da seguire da parte del software di ottimizzazione. Tra i parametri da considerare si annoverano ad esempio: la frequenza con cui applicare l'algoritmo euristico RINS interno a CPLEX, il numero di soluzioni individuate prima che CPLEX restituisca una nuova soluzione migliorante, la tolleranza per il miglioramento richiesto, il time limit, l'ampiezza per l'intervallo corrente di ricerca.

Un'idea per lavori futuri nell'ambito della colorazione dei grafi potrebbe essere quella di accettare comunque come valide, magari all'inizio del processo di ottimizzazione, soluzioni con uno stesso numero di colori dal momento in cui non si riescano ad individuarne di migliori nell'intorno prossimo alla soluzione

corrente. Si potrebbero cioè accettare, ma solo in taluni casi in cui ciò sia di qualche utilità, anche soluzioni non peggioranti. In questo modo si potrebbe andare ad aggiornare la soluzione attuale e quindi spostare l'intorno di ricerca nella speranza di trovare in questo nuovo intervallo nuove soluzioni. Il meccanismo introdurrebbe una maggior diversificazione dello spazio di ricerca e sarebbe mirato a velocizzare l'individuazione di nuove soluzioni all'inizio del processo di ottimizzazione, quando la distanza di nuove colorazioni da quella attuale di riferimento comincia a diventare di una certa entità. Il meccanismo infatti, con lo spostamento dell'intorno da esaminare, permette di limitare entro certi limiti lo spazio da esplorare.

BIBLIOGRAFIA E SITOGRAFIA

[1] Berthold T., Grotschel M., 2006. *Primal Heuristics for Mixed Integer Programs*. PhD Thesis, Università di Berlino.

[2] Coloring problems DIMACS graph format
http://prolland.free.fr/works/research/dsat/dimacs.html

[3] CPLEX Manual website
http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r4/index.jsp

[4] Danna E., Rothberg E., Le Pape C., 2005. *Exploring relaxation induced neighborhoods to improve MIP solutions*. Mathemathical Programming, volume 102, pubblicazione 1, pagine 71-90.

[5] Fischetti M., 1999. *Lezioni di Ricerca Operativa*, Seconda Edizione, Libreria Progetto Padova. Capitolo 6, pagina 118.

[6] Fischetti M., Lodi A., 2003. *Local Branching*: Mathematical Programming 98, pagine 23-47.

[7] Fischetti M., Monaci M., 2013. *Proximity search for 0-1 mixed-integer convex programming*: DEI, Università di Padova.

[8] IBM Corporation, 2011. *IBM ILOG CPLEX Optimization Studio – Getting Started with CPLEX (Version 12, Release 4).*

[9] IBM ILOG website
http://www-01.ibm.com/software/websphere/ilog

[10] Johnson D. S., Aragon C. R., McGeoch L. A., Schevon C., 1991. *Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning*: Oper. Res. 39, pagine 378-406.

[11] Kubale M., 2004. *Graph Colorings.* Marek Kubale editor, Contemporary Mathematics.

[12] Leighton F. T., 1979. *A graph coloring algorithm for large scheduling problems*. J. Res. National Bureau Standards 84, pagine 489-503.

[13] Malaguti E., Monaci M., Toth P., 2008. *A Metaheuristic Approach for the Vertex Coloring Problem*. INFORMS Journal on Computing, Volume 20, Numero 2, pagine 302-316.

[14] Malaguti E., 2006. *The Vertex Coloring Problem and its generalizations*. PhD thesis, Università di Bologna.

[15] Maritz P. e Mouton S., 2012. *Francis Guthrie: a colourful life*. Dirk Huylebrouk editor. The Mathematical Tourist.
[16] Marx D., Dicembre 2003. Graph colouring problems and their applications in scheduling. Periodica Polytechnica, volume 48, pagine 11-16.
[17] Microsoft Visual Studio 2010 website
http://msdn.microsoft.com/it-it/library/vstudio/52f3sw5c(v=vs.100).aspx
[18] Pasotti A., Ottobre 2007. Il teorema dei quattro colori e la teoria dei grafi. Personal publication on Matematicamente.it Magazine, Numero 4.
[19] Voloshin V. I., 2009. *Graph Coloring: History, results and open problems.* Alabama Journal of Mathematics.
[20] Wilson R., 2004. *Four Colors Suffice – How the map problem was solved*. Princeton University Press. Capitolo 2.