ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA e ARCHITETTURA

DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA TRIENNALE IN INGEGNERIA GESTIONALE

TESI DI LAUREA

in

RICERCA OPERATIVA

Modelli e Algoritmi per Problemi di Coloring con il Linguaggio AMPL

CANDIDATO

Lorenzo Giliberti

RELATORE:

Prof. Enrico Malaguti

CORRELATORE

Prof.ssa Valentina Cacchiani

Anno Accademico 2015/2016

Sessione I

Indice

1. INTRODUZIONE

Come il titolo suggerisce, due sono gli aspetti oggetto di questo elaborato: i modelli e gli algoritmi (con le rispettive criticità), e il linguaggio di modellazione AMPL. Il filo conduttore che integra le due parti, nonché mezzo ultimo per un'applicazione pratica dell'attività di modellazione, è l'ottimizzatore, la "macchina" che effettua la risoluzione vera e propria dei suddetti modelli.

Fra gli anni '50 e gli anni '70, la programmazione matematica ha compiuto progressi importanti, cui tuttavia, non è corrisposta un'applicazione altrettanto estesa a problemi reali. Un forte ostacolo è stata la difficoltà pratica di redigere i modelli, raccogliere e organizzare i dati, ma soprattutto la mancanza di mezzi risolutivi adatti alla dimensione dei problemi affrontati.

Negli ultimi decenni sono stati sviluppati diversi risolutori di modelli per soddisfare una richiesta sempre più stringente di migliorare la qualità del processo decisionale delle imprese. Questo perché, con l'introduzione dell'ICT (Information and Communication Technology), la quantità di informazioni utili in gioco e la dimensione dei problemi affrontati sono cresciuti esponenzialmente; proporzionalmente alle nuove problematiche da affrontare, si sono aperti dunque nuovi campi di applicazione per la Ricerca Operativa, la scienza che si occupa di fornire un supporto all'attività decisionale di un'azienda, modellizzando matematicamente la realtà e producendo soluzioni scientificamente ottime tramite l'ausilio di calcolatori. Gli ottimizzatori, creati per soddisfare queste necessità, sono strumenti incredibilmente potenti; tuttavia, le loro capacità risolutive non sempre vengono sfruttate al meglio, dal momento che spesso si ignora il loro funzionamento interno.

Questo elaborato si pone l'obiettivo di approfondire la relazione tra modelli sviluppati e software ottimizzatori. O meglio, si vorrebbe dimostrare che per alcune classi di problemi, la cui la risoluzione per grandi istanze risulta lenta o addirittura impossibile (tempi eccessivamente estesi, memoria disponibile insufficiente), bastano semplici accorgimenti per migliorare sensibilmente la qualità delle informazioni in uscita e la velocità di creazione di un output affidabile.

In particolare, si prenderà in considerazione un noto problema della teoria dei grafi, il *Vertex Coloring Problem* (VCP). Esso appartiene alla classe dei cosiddetti *NP-hard problem*, cioè problemi dei quali non si riesce a definire tempi di risoluzione polinomiali. Si tratta, inoltre, di un problema di programmazione lineare intera (ILP), a variabili binarie. Il VCP cerca di attribuire un colore ad ogni

vertice di un grafo dato, in modo da non avere mai due vertici adiacenti dello stesso colore. L'obiettivo del problema è minimizzare la quantità di colori utilizzati.

In funzione delle proprietà del Vertex Coloring Problem, si considera di utilizzare il software *CPLEX*, attualmente di proprietà dell'IBM ma già in commercio dal 1988: prodotto di grande successo, noto per la sua robustezza ed efficacia risolutiva. Questo ottimizzatore è adatto alla risoluzione di problemi di programmazione lineare e quadratica: il campo di applicazione risulta quindi molto ampio.
CPLEX, come si vedrà più approfonditamente nei successivi paragrafi, utilizza la tecnica del *Branch&Bound*, metodo di ottimizzazione combinatoria che sfrutta la finitezza dell'insieme delle soluzioni ammissibili. Nella ricerca dell'ottimo, CPLEX analizza l'intero poliedro rappresentante lo spazio delle possibili soluzioni, ne verifica l'ammissibilità, infine vi applica la funzione obiettivo (FO): l'ottimo sarà la soluzione che meglio soddisfa la richiesta di minimizzarne o massimizzarne il valore.

I due principali limiti che CPLEX pone, soprattutto per modelli applicati su grandi istanze, sono la valutazione della FO e la cardinalità dello spazio delle soluzioni. Infatti, nel processo di valutazione delle possibili soluzioni, CPLEX salva un Lower Bound (LB) e un Upper Bound (UB): il primo è il valore minimo che la soluzione deve assumere per essere ammissibile, il secondo è il miglior valore fino a quel momento trovato. Utilizzando il *Branch&Bound,* CPLEX applica una suddivisione in sotto-problemi: tuttavia la ricerca di una molteplicità di soluzioni produce un conseguente utilizzo di grandi quantità di memoria, che potrebbero portare l'elaboratore a bloccarsi per saturazione della memoria stessa.

Una volta noto il funzionamento del risolutore, è possibile facilitarne l'attività di computazione limitando il campo di ricerca dell'ottimo. Per fare ciò, sfrutteremo algoritmi euristici al fine di limitare il LB e l'UB della particolare istanza analizzata: ciò significa ridurre considerevolmente il numero di vincoli e variabili create da CPLEX per analizzare il problema e il corrispettivo tempo di risoluzione.

Per le istanze scelte, vedremo come spesso i tempi si riducono drasticamente, invece in alcuni casi la soluzione migliore si trova solamente con l'applicazione integrata di algoritmi e modello del VCP, in altri ancora osserveremo come si riduce il gap tra UB e LB, senza perdere garanzia di ottimalità della soluzione trovata.

Il mezzo usato per implementare il modello di Vertex Coloring Problem nelle strutture dati richieste dal risolutore è *AMPL* (A Mathematical Programming Language).

AMPL è un linguaggio di modellazione di alto livello, possiede infatti una sintassi intuitiva ed algebrica, ma al tempo stesso mantiene un elevato grado di formalità per un'efficace traduzione del codice sviluppato.

AMPL contiene al suo interno un traduttore, che permette di usufruire di una grande varietà di ottimizzatori: questo consente di risolvere non solo problemi di programmazione lineare (LP), ma anche quadratica (QP) e non-lineare (NLP), con variabili intere, continue o miste.

2. VERTEX COLORING PROBLEM

2.1 Premessa

La Teoria dei Grafi nasce da una congettura proposta per la prima volta nel 1852 in una lettera di Augustus De Morgan a W.R. Hamilton. Si consideri di voler colorare una mappa degli Stati in modo che due paesi confinanti abbiano lo stesso colore. Indicando gli Stati come vertici su un piano, connessi solo se hanno confini comuni, si ottiene un grafo planare.
Il famoso *Four Colour Problem* richiede che ogni grafo planare sia colorato con al più quattro colori. Da questo problema, apparentemente semplice, nasce la più ampia teoria dei grafi, da cui deriva il Vertex Coloring Problem. La prima prova, per dimostrare che effettivamente bastino quattro colori per un grafo planare, fu data da Kempe nel 1879; undici anni più tardi, Heawood trovò un errore nella dimostrazione, quest'ultimo riuscì solo a provare che bastano cinque colori per colorare qualsiasi mappa. Dopo una lunga serie di prove confutate protrattesi fino al 1977, quasi un secolo dopo la nascita del problema Appel, Haken e Koch (University of Illinois) pubblicano la dimostrazione definitiva. Fu la prima volta che, per provare un noto problema matematico, si è fatto ricorso a un estensivo utilizzo di un computer, con ben 1200 ore di calcolo.
Al giorno d'oggi, per i problemi legati alla Teoria dei Grafi, risulta fondamentale l'uso del calcolatore, sia per fornire dimostrazioni teoriche, sia per utilizzi pratici, come la risoluzione di istanze reali.

Il Vertex Coloring Problem (VCP) è la versione generalizzata dei problemi appartenenti alla classe dei *Graph Coloring Problem*. Nel mondo reale, trova applicazione in molti campi ingegneristici: nelle attività di schedulazione, ad esempio, per organizzare il calendario degli esami di un'università, o nell'attività di una fonderia; altrettanto utile risulta nell'assegnamento di frequenze per la trasmissione delle radio emittenti, affinché non ci siano interferenze tra stazioni adiacenti, e nella gestione del traffico aereo in prossimità di un aeroporto, dove agli aerei, in attesa dell'autorizzazione all'atterraggio, aspettano a un'altitudine assegnata.

In funzione del campo d'applicazione, il VCP cambia o modifica vincoli per meglio adattarsi alle specifiche del caso, in questo modo sono nate diverse varianti:

- BVCP (Bounded Vertex Coloring Problem), pone un vincolo sulla capacità di un colore, ovvero il numero massimo di vertici a cui può essere attribuito in un grafo. Risulta utile nel momento in cui le risorse, rappresentate dai colori, sono limitate.
- BCP (Bandwidth Coloring Problem), tipicamente utilizzato nelle telecomunicazioni, impone, anziché la non-adiacenza tra due vertici a cui è assegnato lo stesso colore, un vincolo di distanza minima fra i vertici in questione, sotto la quale non è possibile assegnare lo stesso colore;
- MCP (Multicoloring Problem) ad ogni vertice è attribuito un peso w_i, che rappresenta il un numero minimo di colori che gli si devono attribuire;
- WVCP (Weighted Vertex Coloring Problem): ogni colore ha un costo differente, la FO non è più minimizzare il numero di colori, ma il costo relativo al loro utilizzo.

2.2 Definizione

Dato un grafo non orientato *G=(V,E)*, dove *V* è l'insieme dei Vertici, *E* l'insieme dei Lati, rispettivamente di cardinalità n e m, e *H* l'insieme dei Colori. Il **Vertex Coloring Problem** è così definito:

$$(1.1) \quad z = min \sum_{h=1}^{n} y_h$$

$$(1.2) \quad \sum_{h=1}^{n} x_{ih} = 1 \qquad i \in V$$

$$(1.3) \quad x_{ih} + x_{jh} \leq 1 \qquad (i,j) \in E, h \in H$$

$$(1.4) \quad x_{ih} \in \{0,1\} \qquad i \in V, h \in H$$

$$(1.5) \quad y_h \in \{0,1\} \qquad h \in H$$

La FO (1.1) descrive l'obiettivo ultimo del modello: minimizzare il numero di colori utilizzati colorare il grafo.

Il *Vertex Coloring Problem* assegna un colore ad ogni vertice del grafo *G* non ammettendo lati monocromatici, ovvero due vertici adiacenti appartenenti alla stessa classe cromatica. L'obiettivo è quello di minimizzare il numero di colori utilizzati, il cui valore è definito come *chromatic number of G*, indicato con $\chi(G)$.

A tal fine, sono necessarie due variabili, entrambe binarie (1.4) (1.5):

$$y_h \begin{cases} 1 \; se \; il \; colore \; h \; viene \; utilizzato \\ 0 \; altrimenti \end{cases}$$

$$x_{ih} \begin{cases} 1 \; se \; il \; colore \; h \; è \; assegnato \; al \; vertice \; i \\ 0 \; altrimenti \end{cases}$$

Il vincolo (1.2) richiede che tutti i vertici di cui si compone il grafo abbiano uno e un solo colore assegnato. Invece, il vincolo (1.3) impone che non esistano due vertici adiacenti (cioè collegati da un arco) con lo stesso colore.

Il Vertex Coloring Problem è un problema di programmazione lineare intera (ILP): *lineare*, perché la funzione obiettivo è un'espressione dove la variabile incognita y_h, appare alla prima potenza; *intera* perché le variabili, essendo binarie, non possono mai assumere altri valori appartenenti a R che non siano 0 o 1.

2.3 Complessità

Per complessità si intende il numero di risorse impiegate per la risoluzione di un particolare tipo di problema. La complessità a cui ci riferiamo è quella computazionale, per cui le risorse principali impegnate sono il tempo, richiesto per eseguire le operazioni necessarie, e lo spazio, per memorizzare e manipolare i dati.

Il più grande ostacolo da affrontare nella risoluzione di problemi come il Vertex Coloring Problem è quello di limitare il tempo di risoluzione. Per considerare un problema "efficiente" è necessario che il running-time sia *polinomiale*, ovvero esprimibile come $O(n^k)$, dove *n* rappresenta la dimensione degli input al problema.

Invece, il Vertex Coloring Problem appartiene alla classe dei cosiddetti *NP-Hard Problem*, ovvero l'insieme di tutti quei problemi dei quali, ad oggi, non si riesce a definire tempi di risoluzione polinomiali, dove è necessario verificare ogni possibile soluzione per trovare l'ottimo.

Attualmente, algoritmi esatti proposti per il Vertex Coloring Problems sono capaci di risolvere correttamente solo piccole istanze, con meno di 100 vertici; tuttavia, applicazioni reali generalmente hanno dimensioni molto maggiori: trattano grafi con centinaia o migliaia di vertici. Senza l'ausilio di tecniche euristiche o meta-euristiche, questa classe di problemi non troverebbe soluzione, tantomeno si è certi di trovare sicuramente l'ottimo. Questo limite ha portato grande interesse in letteratura per questa categoria di problemi, sia per aspetti teorici come per quelli computazionali.

2.4 Algoritmi euristici

Il più grande ostacolo da affrontare nella risoluzione di problemi come il Vertex Coloring Problem è quello di limitare il tempo di risoluzione. Per considerare un problema "efficiente" è necessario che il running-time sia *polinomiale*, ovvero esprimibile come $O(n^k)$, dove *n* rappresenta la dimensione degli input al problema.

Per ridurre il tempo di elaborazione, ci si avvale dell'ausilio di algoritmi euristici e meta-euristici: essi non forniscono con certezza la soluzione ottima al particolare problema, ma producono a una soluzione per lo meno sub-ottima, ovvero che può essere vicina (o appartenere) allo spazio delle soluzioni ottime. In funzione delle necessità dell'utente, ci si può limitare alle soluzioni fornite da tali algoritmi, o utilizzarli per limitare il campo di soluzioni analizzate dal modello di Vertex Coloring Problem.

I vari algoritmi utilizzabili sono valutati tramite un'analisi comparativa sperimentale: gli elementi di giudizio di un algoritmo sono essenzialmente due: la qualità della soluzione ottenuta (quanto mediamente si avvicina alla soluzione ottima) e il tempo di calcolo richiesto. I risultati vengono confrontati utilizzando istanze comuni, provenienti dalla letteratura o da applicazioni reali, in ogni caso a disposizione della comunità scientifica per permettere a chiunque di effettuare questo tipo di analisi comparative.

Ad esempio, per i problemi di *Graph Coloring*, il DIMACS (Center for DIscrete MAthematics and Theoretical Computer Science) è un ente che si propone di fornire una serie di istanze in formato standardizzato, su cui la comunità scientifica può comparare le differenti soluzioni al problema.

2.4.1 Algoritmo di Upper Bound

Il SEQ è un algoritmo euristico appartenente alla famiglia degli algoritmi *greedy* (letteralmente voraci, golosi). Essi determinano la soluzione attraverso una serie di decisioni "localmente ottime" senza mai rivalutare le decisioni prese. Sono algoritmi di facile implementazione con buona efficienza computazionale; forse tra i più semplici proposti per la risoluzione di problemi di *Graph Coloring.* Esistono diverse varianti, proposte per rendere più sofisticato questo algoritmo, ma sostanzialmente la colorazione avviene sempre "al primo tentativo". Inoltre, anche variando la permutazione dei vertici, può accadere che SEQ restituisca una soluzione diversa, il numero di colori utilizzati potrebbe aumentare come diminuire.

Supponendo di avere i vertici numerati $v_1, v_2, \dots, v_n$, SEQ analizza sequenzialmente ogni vertice singolarmente, ricercando la soluzione ottima con il seguente ragionamento: attribuisce v_1 alla prima classe di colore, dopodiché assegna i seguenti vertici v_i $(i = 2, \dots, n)$ alla classe di colore avente indice più basso, a patto che non contenga altri vertici adiacenti a v_i.

Nel caso specifico del Vertex Coloring Problem, questo algoritmo impone un limite superiore alla funzione obiettivo (1.1), ovvero fornisce il numero di colori che, al massimo, il modello potrà utilizzare per risolvere il problema. Nel caso in cui corrisponda al valore trovato dal modello, allora è ottima anche la soluzione data da SEQ, altrimenti permette di limitare lo spazio delle soluzioni che il solver deve analizzare per trovare la soluzione ottima al problema. Per questo motivo, viene anche detto algoritmo di "Upper Bound", dal momento che impone un limite superiore all'ottimo.

La complessità dell'algoritmo SEQ è $O(n^2)$. Considerando il caso peggiore, l'algoritmo deve costruire una struttura dati che per ogni vertice assegni un colore diverso. Quando realmente si prova ad assegnare un vertice v_i a una classe di colore, l'algoritmo deve verificare $O(n)$ colori (si spera cioè di non essere nel caso peggiore) per tutti gli n vertici; otteniamo quindi la complessità sopraindicata.

La logica di un generale algoritmo greedy presenta tre proprietà fondamentali.

- incrementalità: la soluzione viene costruita per gradi, l'elemento selezionato è scelto secondo un criterio attribuito all'algoritmo;
- no-backtracking: la decisione presa non viene mai messa in discussione durante l'intera esecuzione dell'algoritmo;
- criterio di selezione greedy: la selezione avviene secondo un criterio "goloso" (greedy, per l'appunto), cioè tra gli elementi non valutati, viene scelto quello che, localmente, ottimizza il criterio di selezione dell'algoritmo.

2.4.2 Algoritmo di Lower Bound

L'algoritmo di Lower Bound appartiene sempre alla famiglia degli algoritmi euristici di tipo greedy; esso va a definire il valore minimo che deve assumere la funzione obiettivo. L'obiettivo di questo algoritmo consiste nel determinare la più grande *clique* dell'istanza data, quindi del grafo. In altre parole, un insieme di vertici connessi forma una clique nel momento in cui tutti gli elementi che la compongono sono adiacenti fra loro. Questo significa che per ognuno di essi si dovrà utilizzare un colore differente, ottenendo così il valore minimo della funzione obiettivo.
L'algoritmo implementato ricerca innanzitutto il vertice con il maggior numero di altri vertici adiacenti. Dal momento che la ricerca della *maximum clique,* quella in assoluto migliore, potrebbe risultare più complessa della ricerca della soluzione al modello stesso (quando invece l'obiettivo dell'algoritmo è quello di ottimizzarne i tempi di risoluzione), si suppone il che il vertice con il più alto grado sia il primo elemento della clique K. Dopodiché si itera la ricerca dei successivi elementi del sottoinsieme di vertici connessi al suddetto vertice, dando la precedenza ai vertici con grado più alto, fino a che non viene più soddisfatta la condizione di totale adiacenza per entrare nella clique K. Fissare una clique non significa solo dare un LB al risolutore, ma anche fissare il valore delle variabili in gioco: in questo modo viene ridotto il numero di soluzioni ottime equivalenti che verrebbero analizzate dall'ottimizzatore, dal momento che ci interessa esclusivamente conoscere il numero di classi di colore utilizzate, e non un particolare assegnamento dei vertici ad esse.

Se le soluzioni dei due algoritmi, di Upper Bound (UB) e Lower Bound (LB) coincidono, allora tale soluzione è ottima: non sarà necessario inizializzare il modello.

3. AMPL

3.1 Caratteristiche generali del linguaggio

AMPL è un linguaggio di modellazione algebrica, ad alto livello, per la programmazione matematica. Ideato e implementato dai creatori Robert Fourer, David Gay e Brian W. Kernighan, nasce nel 1985 nei *Bell Laboratories*. L'obiettivo per cui venne progettato, è stato quello di definire un linguaggio che aiutasse i utenti nella formulazione di modelli di programmazione matematica.

La modellazione matematica non consiste solo nell'applicazione di semplici algoritmi, dei quali minimizzare o massimizzare la funzione obiettivo. Se i modelli potessero affrontare problemi di ottimizzazione come lo fa un essere umano, l'attività di formulazione potrebbe risultare abbastanza semplice e veloce.
Nella realtà invece, le modalità espressive del modellatore e di un algoritmo sono differenti: questo comporta una notevole spesa di tempo nella fase di traduzione, inoltre l'efficienza dell'algoritmo può risultare peggiore delle aspettative, poiché vincolato alla sintassi del linguaggio.

AMPL si propone come un nuovo mezzo per facilitare l'attività di modellazione e renderla meno soggetta ad errori. Sinteticamente, possiamo riassumere la forza di questo linguaggio nei seguenti punti:

- sfrutta una sintassi molto simile alla **notazione algebrica-simbolica**, comunemente utilizzata per descrivere modelli matematici, e allo stesso tempo preserva una **struttura sufficientemente formale** per essere processata su qualsiasi programma di ottimizzazione, anche chiamati *solver*;

- mantiene la struttura logica del modello (descritto con variabili, parametri, insiemi, vincoli e funzioni obiettivo) autonoma rispetto al valore delle specifiche istanze;

- possiede un “traduttore” implementato al suo interno che, presi come input i modelli e i dati associati, produce un output corretto per i più importanti ottimizzatori a disposizione sul mercato, dal momento che ognuno di essi propone un proprio ambiente integrato di sviluppo dei modelli e strutture dati;

- garantisce al modello completa indipendenza dai diversi risolutori compatibili; allo stesso tempo permette all’utente di passare velocemente da un solver all’altro, confrontandone le prestazioni in funzione del modello utilizzato.

L’insieme di queste caratteristiche rendono AMPL uno strumento molto versatile, poiché permette all’utente di approcciarsi, con un unico linguaggio, a diversi ottimizzatori quando, invece, ognuno di questi avrebbe richiesto una particolare programmazione. Quindi, al tempo stesso, facilita enormemente la formulazione di modelli di ottimizzazione e genera strutture dati con i corretti requisiti computazionali.

AMPL, inoltre, non si limita ad essere un *linguaggio di modellazione*, ma integra anche un *linguaggio di comando* per il debug (localizzazione errori) di modelli e analisi degli output, e un *linguaggio di scripting* (richiamo e accorpamento di altri programmi) per la manipolazione dei dati e l’attuazione di strategie di ottimizzazione.

I principali ottimizzatori disponibili su AMPL si possono suddividere in due macrocategorie, in funzione delle proprietà del modello studiato:

- CPLEX, Gurobi, Xpress sono solver adatti per la risoluzione di modelli lineari o quadratici, con variabili intere, continue o miste;
- KNITRO, MINOS, SNOPT permettono di trovare soluzioni locali a problemi non lineari con variabili continue.

Con una sintetica rappresentazione grafica (Fig.1), possiamo vedere come AMPL si pone come filtro tra modello, dati associati e solver. In maniera tanto trasparente come nella traduzione dei dati, AMPL fornisce gli output dell’ottimizzatore allo stesso modo in cui sono stati sviluppati dati e modello:

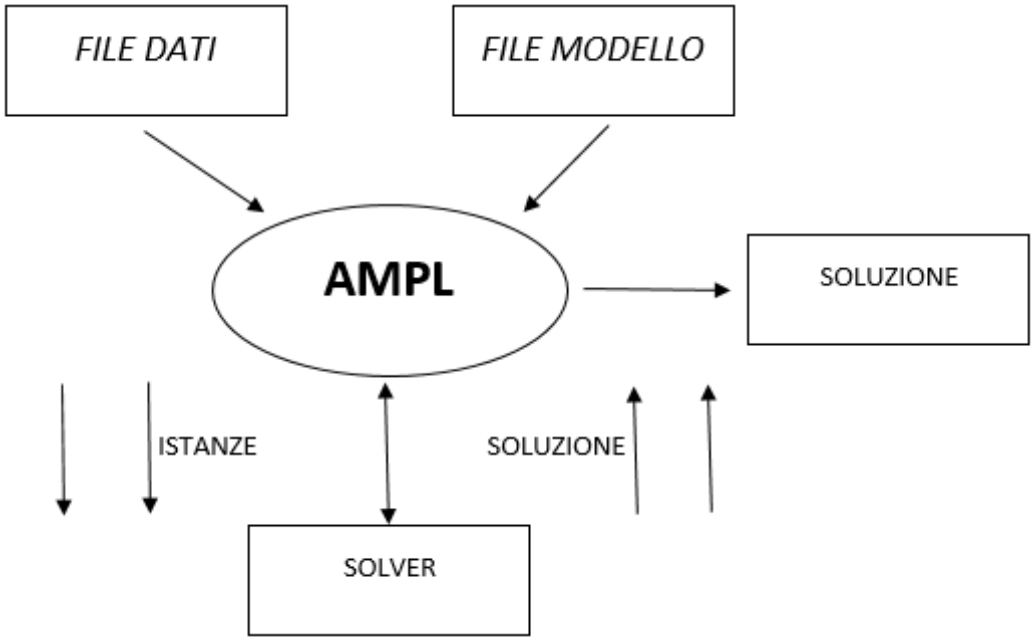


Fig. 1

3.2 Introduzione all'interfaccia grafica

Nel sito *www.ampl.com*, oltre alla versione completa del software AMPL, a pagamento, è disponibile una versione Demo (limitata a 500 variabili, 500 vincoli) chiamata AMPLIDE, insieme a solver rappresentativi eseguibili su Windows, Linux e Mac OS X.
AMPLIDE risulta utile per sperimentare e allenarsi con il linguaggio.

AMPL, nella versione completa come in quella Demo, è dotato di una GUI, ovvero di un'interfaccia grafica, che facilita l'utilizzo del software. Tale interfaccia rende disponibili diversi solver contemporaneamente: ciò permette di passare da un risolutore all'altro, e confrontarne le performance in funzione della classe di modello analizzata. Una volta elaborate le soluzioni ottime, vengono automaticamente restituite all'utente, così da vedere ed analizzarne i risultati.

L'interfaccia si suddivide in tre aree:

- a sinistra, permette un accesso veloce ai file;
- al centro vi è la *pagina di comando*, ovvero un collegamento diretto col prompt dei comandi;

- a destra, infine, è la parte che sintetizza il punto di forza di AMPL. Qui, è possibile sviluppare il modello e i rispettivi dati in file di testo (anche se, in verità, presentati in diversi formati a seconda dell'utilizzo).

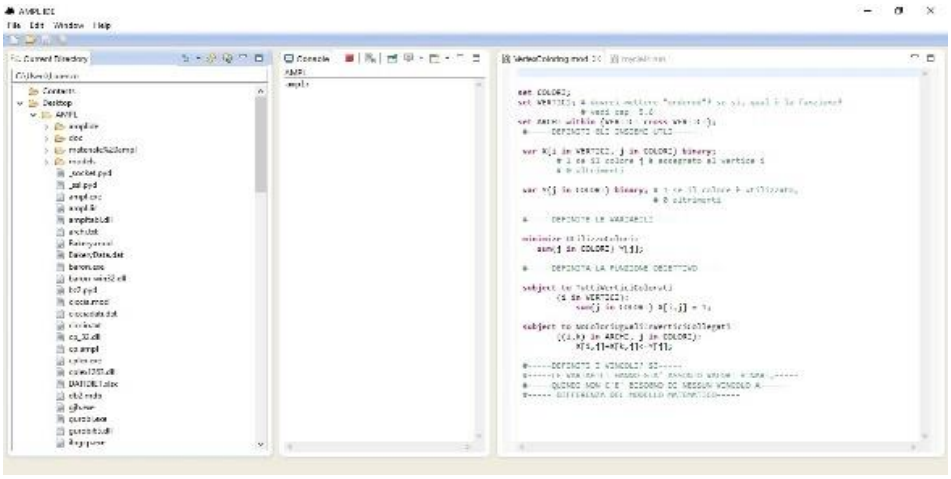


Fig. 2

3.3 Scrivere un modello

Eseguendo AMPL si apre l'interfaccia grafica, dalla cui barra del menù, per iniziare, è possibile selezionare un nuovo file, di cui dovremo specificare nome e formato. Si possono utilizzare tre tipi di estensioni:

- .mod: file dove si sviluppa la struttura logica del modello;
- .dat: dove vengono salvati, opportunamente inseriti, i dati specifici dei modelli, lasciando a loro così un certo grado di indipendenza dai dati;
- .run: particolare formato che consente l'utilizzo di AMPL in modalità "batch": al suo interno, una volta caricati modello e dati previamente sviluppati e controllati, si possono applicare diversi comandi, far iterare operazioni. Questo solleva l'utente dal doverle riscrivere più volte nella sezione centrale, ovvero nella *pagina di comando*.

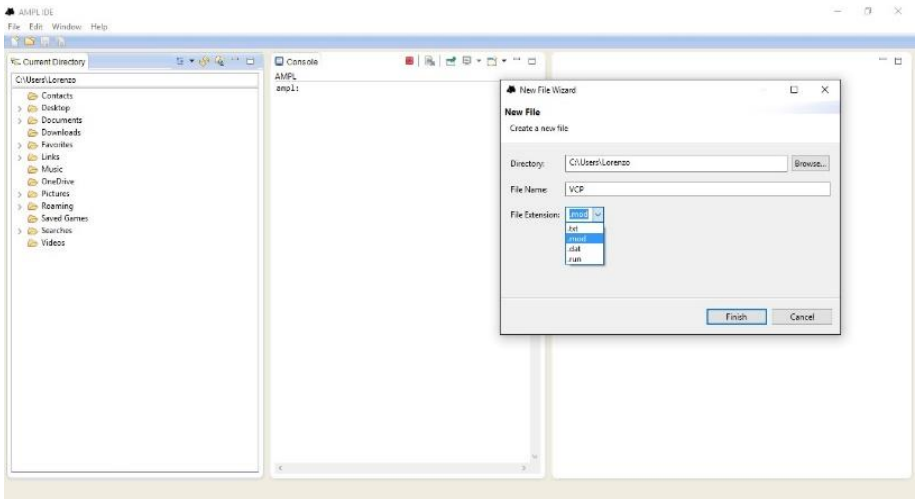


Fig. 3

Introdurremo ora il modello rispettando la sintassi del linguaggio AMPL. Il linguaggio utilizza 5 entità fondamentali, a cui associa una determinata parola chiave:

ENTITA'	**PAROLA CHIAVE**
Insiemi	Set
Variabili	Var
Funz. Obiettivo	minimize/maximize
Vincoli	subject to
Parametri	param

In generale, un'istruzione in AMPL può essere di due tipi: dichiarativa o esecutiva. Si dice dichiarativa quando serve a descrivere i modelli, esecutiva se utilizzata per eseguire operazioni su di essi (ad esempio, imporre rilassamenti di vincoli, iterare la risoluzione partendo da dati diversi etc.), come faremo per lo sviluppo degli algoritmi euristici.

Lo schema sintattico generale delle istruzioni dichiarative è:

entità *nome* [quantificatori] [specificazioni] [: espressione-matematica | := definizione];

es. **var** *X* **integer;**
es. **set** *INSIEME* **ordered;**

AMPL, per segnalare il corretto inserimento delle parole chiave, attribuisce ad esse un colore differente da quello del resto della dichiarazione.

4. IMPLEMENTAZIONE IN LINGUAGGIO AMPL

4.1 VCP

In questo paragrafo, andremo ad analizzare tutti i passi necessari per implementare il modello tramite il linguaggio AMPL. Per prima cosa, creiamo un file in formato *.mod* dove salvare la struttura logica del problema. Verranno introdotte le entità precedentemente descritte, dando per ognuna di esse una precisa definizione e il metodo per una corretta dichiarazione.

4.1.1 Insiemi e Parametri

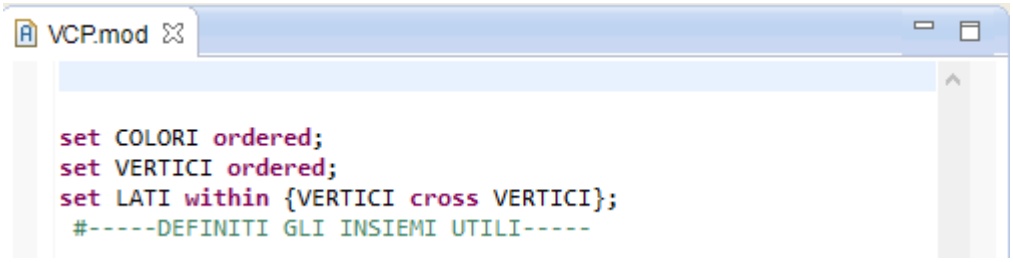


Fig. 4

Il primo aspetto da definire in AMPL sono gli *insiemi*, generalmente identificabili come una collezione (al momento non concretamente specificata) di oggetti pertinenti al modello. Al momento questi vettori che possono contenere valori numerici o stringhe non vengono popolati poiché, come affermato nell'introduzione al linguaggio, istanze e modello sono mantenuti in file differenti, al fine di garantire l'indipendenza logica al problema descritto in maniera astratta.

L'oggetto delle attenzioni del problema, ovvero il grafo, è composto da due insiemi, Vertici e Lati, anche chiamati rispettivamente V ed E nella definizione del modello matematico; creiamo inoltre un insieme di Colori (H) possibili da attribuire ai vari vertici: nel caso peggiore, il numero di colori utilizzati sarà pari a quello dei vertici, ovvero *n*.
In AMPL, come si può notare dalla Fig.4, le istruzioni per dichiarare un insieme è la seguente:

set <nome_insieme> <proprietà_insieme>;

Ogni istruzione completa deve terminare con il simbolo **";".**
I commenti che accompagnano le dichiarazioni cominciano con il simbolo # e si estendono fino alla fine della linea, ovvero AMPL non considera come comando tutto ciò che segue il cancelletto. I commenti stessi sono spesso utilizzati per facilitare la lettura delle righe di codice.

Gli elementi di un insieme possono essere numeri o stringhe: su questi valori, possono essere indicizzati, una volta dichiarati, parametri e variabili. Indicizzare su valori numerici ha il vantaggio di poter effettuare operazioni matematiche, qualora sia necessario, sugli indici stessi, facilitando l'attività di programmazione matematica.

Tornando al VCP, possiamo notare che i Lati altro non sono che collegamenti fra coppie di vertici, motivo per cui l'insieme di valori sarà dato dalle possibili combinazioni che possono assumere n^2. In questo caso, per creare un insieme bidimensionale, come "incrocio" di valori di un altro insieme precedentemente dichiarato, è possibile utilizzare la parola chiave *cross.* Il termine within, invece, sta a indicare che E sarà sottoinsieme di ciò che segue.

set E **within** {V **cross** V};

Anche se in questo modello non compaiono, è necessario menzionare l'esistenza dei *parametri*. Essi sono valori numerici, non variabili, pertinenti al modello; possono essere valori limitati inferiormente o superiormente, indipendenti o indicizzati su insiemi, interi razionali o binari. La loro dichiarazione assomiglia a quella degli insiemi, se non che cambia la parola chiave della dichiarazione:

param <nome_parametro> <proprietà_parametro>;

Nella scelta dei nomi da attribuire ai vari elementi dichiarati, bisogna ricordare che AMPL è *case sensitive,* cioè le lettere maiuscole e minuscole sono considerate differenti.

4.1.2 Variabili

A seguire la definizione delle *variabili* del modello, che, come descritto, sono due e possono assumere solo valori binari.

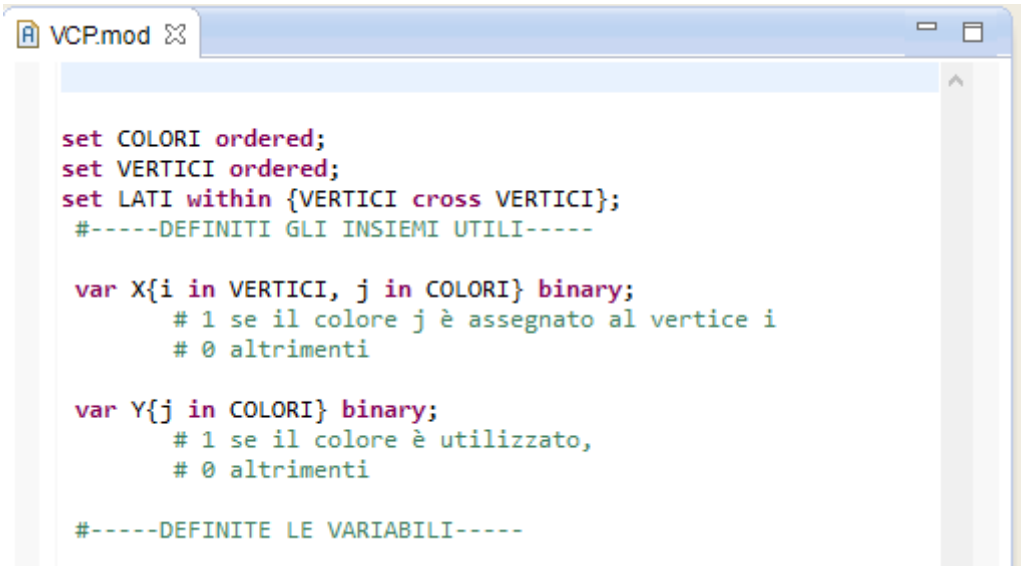


Fig. 5

Le variabili rappresentano le *grandezze incognite* di cui ricerchiamo il valore risolvendo il problema di ottimizzazione. La struttura della dichiarazione delle variabili è la seguente:

var <nome_variabile> <proprietà_variabile>;

Come affermato nel precedente paragrafo, spesso le variabili sono indicizzate su uno o più insiemi dichiarati in precedenza. Per permettere questa operazione, sono stati ideati i cosiddetti *quantificatori*: racchiudendo tra parentesi graffe l'insieme di valori su cui indicizzare la variabile, andiamo a mettere in relazione i due elementi considerati.

In AMPL, non è necessario porre nei vincoli sui valori assumibili dalle variabili, ma, essendo una loro proprietà, queste specifiche possono essere direttamente incluse al momento della sua dichiarazione: ad esempio, le variabili del Vertex Coloring Problem assumono valori binari {0,1}: possiamo imporlo utilizzando la parola chiave *binary*.

4.1.3 Funzione Obiettivo

Vedremo ora la *funzione obiettivo* (FO):

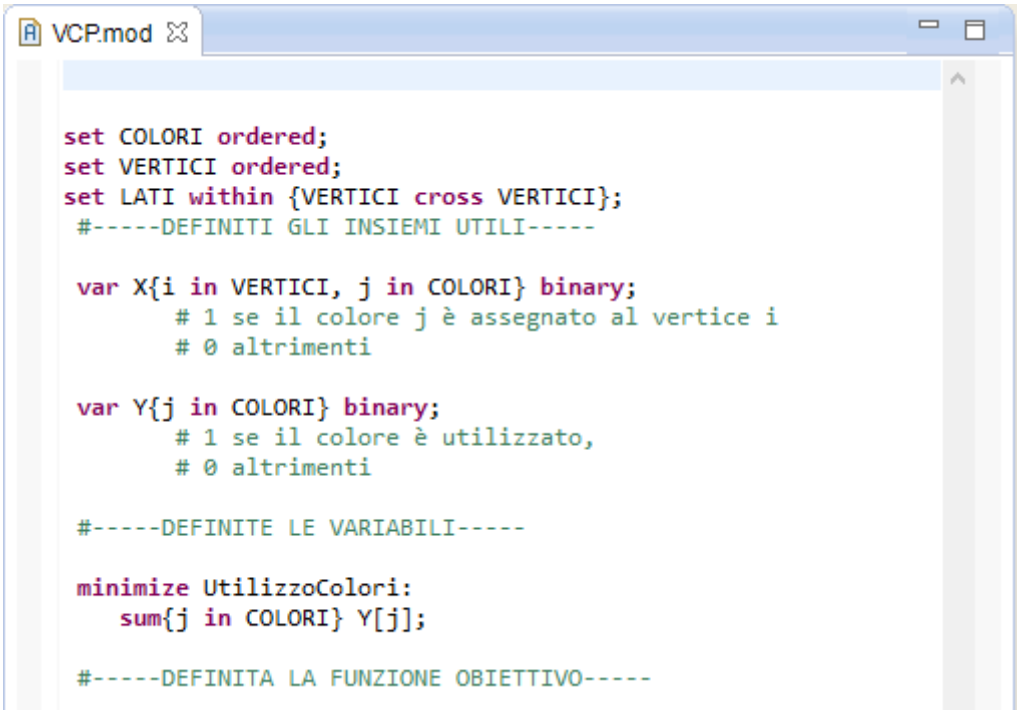


Fig. 6

Si tratta di un'*espressione matematica* in funzione delle variabili, parametri e indici validi al momento della dichiarazione. La funzione obiettivo specifica la complessità del problema di cui si vuole trovare la soluzione ottimale.

Come per le altre entità, AMPL mantiene sostanzialmente la stessa struttura di dichiarazione anche per la funzione obiettivo:

minimize/maximize <nome_FO> : <espressione_FO>;

Sommatorie $\sum_{i=1}^{n} x_i$ o prodotti $\prod_{i=1}^{n} x_i$ iterati su un insieme, sia nella funzione obiettivo che in un vincolo, sono tradotte, rispettivamente, con le parole chiave *sum* e *prod*.

Nella programmazione lineare, la funzione obiettivo ha sostanzialmente la stessa struttura di un vincolo, eccetto che non possiede un operatore relazionale e inizia con le parole *minimize* o maximize. Inoltre, AMPL permette di dichiarare più di una funzione obiettivo, sebbene vengano elaborate una alla volta; esse possono essere indicizzate o comparire singolarmente.

4.1.4 Vincoli

Per ultimi, definiamo i *vincoli*:

VCP.mod

```
set COLORI ordered;
set VERTICI ordered;
set LATI within {VERTICI cross VERTICI};
 #-----DEFINITI GLI INSIEMI UTILI-----

 var X{i in VERTICI, j in COLORI} binary;
        # 1 se il colore j è assegnato al vertice i
        # 0 altrimenti

 var Y{j in COLORI} binary;
        # 1 se il colore è utilizzato,
        # 0 altrimenti

 #-----DEFINITE LE VARIABILI-----

 minimize UtilizzoColori:
    sum{j in COLORI} Y[j];

 #-----DEFINITA LA FUNZIONE OBIETTIVO-----

 subject to TuttiVerticiColorati
        {i in VERTICI}:
            sum{j in COLORI} X[i,j] = 1;

 subject to NoColoriUgualiInVerticiAdiacenti
        {(i,k) in LATI, j in COLORI}:
            X[i,j]+X[k,j]<=Y[j];

 #-----DEFINITI I VINCOLI? SI-----
 #-----LE VARIABILI HANNO GIA' ASSUNTO VALORI BINARI,-----
 #-----QUINDI NON C'E' BISOGNO DI NESSUN VINCOLO A-----
 #----- DIFFERENZA DEL MODELLO MATEMATICO-----
```

Fig. 7

Il vincolo consiste nel confronto di due espressioni algebriche tramite relazioni di uguaglianza o disuguaglianza: non è necessario che le variabili siano contenute nella prima espressione del vincolo. L'equazione, o disequazione, può contenere solo variabili e parametri dichiarati nel modello, o valori numerici costanti.

Anche in questo caso, la struttura per la dichiarazione non si allontana da quella di una funzione obiettivo:

subject to <nome_vincolo> : <relazione_vincolo>;

I vincoli delimitano lo spazio delle *soluzioni ammissibili* da quello delle *non ammissibili*, e quindi i possibili valori che la funzione obiettivo può assumere.

4.1.5 Dati

myciel3.dat

```
data;

set COLORI :=
1
2
3
4
5
6
7
8
9
10
11
;

set VERTICI :=
1
2
3
4
5
6
7
8
9
10
11
;

set LATI :=
1   2
1   4
1   7
1   9
2   3
2   6
2   8
3   5
3   7
3   10
4   5
4   6
4   10
5   8
5   9
6   11
7   11
8   11
```

Fig. 8

Una volta che il traduttore ha letto e processato il modello, AMPL è pronto per la lettura dei dati. Come specificato nell'introduzione, al modello astratto descritto in precedenza devono essere combinati i dati degli insiemi e parametri dichiarati.
Successivamente, vedremo che le istanze su cui verranno testati gli algoritmi e valutati i risultati rispettano la struttura standard richiesta da DIMACS; tuttavia AMPL non è capace di leggere i dati secondo questa impostazione, dal momento che richiede una semplice, ma rigida, forma espressiva. Per permettere l'indipendenza logica del modello da un particolare grafo, le istanze vengono salvate in un opportuno file in formato *.dat*.
Tuttavia, i dati, come il modello, devono essere scritti in un file di testo, seguendo precise regole formali: questo riduce la flessibilità del linguaggio. Allo stesso tempo, attraverso semplici righe di comando, è possibile accedere a Input salvati in file Access, Excel, o altre basi di dati, purché opportunamente strutturate.

In AMPL, i dati specificabili sono quelli di parametri e insiemi. Come nel caso preso da esempio nella Fig.8, riportiamo i dati relativi a un semplice grafo G= (11,20). Per assegnare i dati relativi alle entità dichiarate nel modello, dobbiamo riutilizzare la stessa dicitura dell'inizializzazione, ma ad essa seguono i dati specifici dell'insieme o parametro che andiamo a popolare di dati.

entità <nome_entità> := dati;

I dati possono essere semplicemente intervallati da una spaziatura: se l'entità è monodimensionale, tutto ciò che intercorre tra l'assegnamento e il punto e virgola, intervallato da una spaziatura, sarà letto da AMPL come un dato. Se l'entità è invece bidimensionale, o più grande, AMPL leggerà come coppia due dati inseriti successivamente l'un l'altro.

Nonostante il tutto venga salvato in un file .dat, è opportuno specificare che stiamo andando a definire i dati di una particolare istanza, introducendo quindi la dichiarazione con la parola chiave *data*.

Una volta letti i dati con successo, vengono determinati tutti i valori degli insiemi e parametri dichiarati. In questo modo, il traduttore di AMPL potrà strutturare il problema lineare: identificare il numero di variabili da dimensionare, la quantità di vincoli e la funzione obiettivo, e finalmente scrivere l'output utile all'ottimizzatore per risolvere il problema.

4.2 Algoritmo di Upper Bound

Ogni algoritmo parte da dati in ingresso (Input) per produrre dati elaborati in uscita (Output). Nell'algoritmo SEQ applicato al Vertex Coloring Problem, i dati di Input sono il grafo G = (V,E) e i colori $c \in C$, dove, considerando il caso peggiore per cui ad ogni vertice venga assegnato un colore diverso, C deve avere la stessa cardinalità di V. Questi valori si considerano già dichiarati nella definizione del modello VCP.

In questa parte non ci soffermeremo in una dettagliata descrizione dell'algoritmo, ma ci limiteremo commentare quelli che sono i passaggi critici della sua implementazione.

Dato un grafo $G = (V, E)$, dove i vertici sono ordinati cosi che $V = \{v_1, v_2, \dots, v_n\}$, l'algoritmo SEQ si propone di risolvere il problema effettuando tre semplici passi:

- Il primo vertice v_1 è assegnato al colore 1;
- Attribuisci v_{i+1} al numero di colore più basso possibile in modo che non vi appartengano mai due vertici adiacenti;
- Ripeti il passo 2 per $i = \{1, \dots, n-1\}$.

Un vertice può essere assegnato a una classe se e solo si verifica che non esista adiacenza con nessuno dei vertici già assegnati a tale colore. Questo significa che ognuno degli n vertici verrà verificato potenzialmente su $O(n)$ classi di colore, da cui otteniamo la complessità finale dell'algoritmo $O(n^2)$. Al fine di limitare tale operazione, sfruttiamo un parametro binario, dichiarato con il nome di *flag*: esso interrompe il ciclo di ricerca sulla classe di colore nel momento in cui viene verificata un'adiacenza.

Come si può vedere nel ciclo più interno dell'algoritmo, la suddetta verifica è formata da due predicati di ricerca: per interrompere il ciclo, il vertice v non deve essere connesso ad un altro vertice w ($(v, w)\ in\ LATI\ or\ (w, v)\ in\ LATI$) già assegnato ($X[w, c] = 1$) alla classe presa in considerazione. Se la condizione è soddisfatta, allora il parametro *flag* assume il valore zero, interrompendo così il ciclo; altrimenti, l'iterazione continua finché il vertice $v$soddisfa i requisiti per essere assegnato al colore c.

SEQ.run

```
#--in questo file proveremo a sviluppare l'algoritmo di Upper Bound (UB)--
#--------SEQ----------

param flag binary;
param maxcolore default 0;

for {v in VERTICI} {
        for {c in COLORI} {
            if c > maxcolore
                    then {
                        let maxcolore := c;
                        }
            let flag := 1;
            for {w in VERTICI} {
                if ((v,w) in LATI or (w,v) in LATI) and X[w,c] = 1
                        then {
                            let flag := 0;
                            break;
                             }
                    };
            if flag = 1
                    then {
                        let X[v,c] := 1;
                        break;
                        }
            };
    };


display maxcolore;
```

Fig. 9

4.3 Algoritmo di Lower Bound

La struttura dell'algoritmo di Lower Bound risulta più complessa rispetto al precedente, in quanto si compone di due fasi principali:

- definizione del vertice con il grado $\delta(v)$ più alto;
- sviluppo della clique K.

Il grado di un vertice $\delta(v)$, in un grafo non orientato, corrisponde al numero di lati che hanno tale vertice considerato come terminale; non avendo nelle istanze considerate alcun *loop*, corrisponde allora al numero di nodi connessi a v.

Il problema della ricerca della clique più grande di un grafo appartiene anch'esso alla categoria dei *NP-hard problem*. Non essendo la trattazione di tale problema oggetto di questo elaborato, ci

limiteremo a studiare l'algoritmo: esso risulta, infatti, strumentale per dare sostegno allo scopo proposto nell'introduzione.

Nella Fig.10 vengono inizializzati degli insiemi speculari a quello dei vertici per non andare a modificarne irrimediabilmente il contenuto; tali insiemi vengono utilizzati per la ricerca del vertice con il massimo grado, cioè col maggior numero di vertici adiacenti; qualora ne esista una molteplicità, ci assicuriamo che ne venga selezionato solo uno.
Il grado massimo del suddetto vertice, che sarà il primo elemento della clique, viene salvato in un parametro precedentemente inizializzato.

SEQ.run | LB.run

```
#--in questo file proveremo a sviluppare l'algoritmo di Lower Bound (LB)--

set VERTICIP2 default VERTICI;
set VERTICIP default VERTICIP2;
set J default {};
set K default {} ;

param gradovertice{VERTICIP2} default 0;
param gradomassimo;

for {(i,j) in LATI, v in VERTICIP2} {
    let gradovertice[v] := (if i=v or j=v
                                then gradovertice[v]+1
                                else gradovertice[v]);
                        };
# cerco di attribuire a un parametro chiamato gradomassimo
# il grado massimo all'interno del vettore

let gradomassimo := max{v in VERTICIP2} gradovertice[v];

let J := {v in VERTICIP2: gradovertice[v] == gradomassimo};
let K := {max{i in J} i};
```

Fig. 10

Nella Fig.11, invece, viene completata la clique K partendo dal vertice previamente selezionato.

Un sottoinsieme di nodi, per essere definito *clique*, vuole che tutti i suoi elementi siano reciprocamente connessi tramite un arco.
Data questa stringente condizione, è inutile iterare la ricerca, partendo dal vertice con massimo $\delta(v)$, su tutti gli elementi del grafo: risulta infatti più conveniente limitare l'analisi ai nodi connessi al primo elemento della clique.

Dall'algoritmo otteniamo due informazioni fondamentali: la cardinalità di K, e i particolari vertici di cui è costituito. Il primo determina il numero di colori minimo per soddisfare il Vertex Coloring Problem, il secondo per darci una soluzione iniziale da cui partire.

```
set H default {};
set U default {};
param gradomax;
param mincolore integer default 0;

repeat {
let U := {};
for {v in VERTICIP} {
            if forall{z in K} ((z,v) in LATI or (v,z) in LATI)
                then{
                    let H := H union {v};
                    }
    };

if not exists{h in H} h
    then {
        break;
        }

let gradomax := max{h in H} gradovertice[h];
let U := {h in H: gradovertice[h] = gradomax}; #prendo quelli di gradomax
let K := K union {max{u in U} u};
let H := H diff K;
let U := {};
let VERTICIP := {};

for {h in H} {
            if forall{k in K} ((h,k) in LATI or (k,h) in LATI)
                then {
                    let VERTICIP := VERTICIP union {h};
                    }

        };

if not exists{v in VERTICIP} v
    then {
         break;
         }

let gradomax := max{s in VERTICIP} gradovertice[s];
let U := {s in VERTICIP: gradovertice[s] = gradomax};
let K := K union {max{u in U} u};
let VERTICIP := VERTICIP diff K;
let H := {};

};

for {k in K} {
    let mincolore := mincolore +1;
    };

display mincolore;
```

Fig. 11

4.5 Codice Integrato

Gli algoritmi, presi singolarmente, non riportano utili informazioni sul grafo analizzato. Tuttavia, se integrati con il modello di Vertex Coloring Problem, possono essere di grande aiuto nel facilitare la ricerca dell'ottimo, proprio perché, come affermato nei precedenti paragrafi, limitano il campo delle soluzioni analizzate dall'ottimizzatore.

Andremo ora ad analizzare il file contenente le istruzioni di comando per una corretta integrazione tra gli algoritmi previamente descritti e il modello del problema.

Per prima cosa, nel file *batch* carichiamo, in ordine, modello e dati associati, specifici del grafo scelto. Segue la scelta del solver da utilizzare; abbiamo già parlato della grande versatilità di AMPL, capace di mettere in relazione, attraverso il traduttore, modelli e algoritmi (implementati nel suddetto linguaggio) con diversi ottimizzatori. Il comando per scegliere un particolare risolutore è:

option solver <nome_solver>;

AMPL imposta di default MINOS, per cui, seguendo l'istruzione appena descritta, richiamiamo CPLEX. Inoltre, per ogni risolutore, sono a disposizione una serie di direttive per personalizzarne l'attività durante il running-time; definiamo quindi quattro specifiche tra le opzioni disponibili:

- *time=300*, limita il tempo di processamento a 300 secondi, di default praticamente infinito (*time=*e^{75}). Si è scelto di imporre un *time-limit* perché può accadere che l'ottimizzatore restituisca errori del tipo out-of-memory, cioè la memoria occupata dall'elaboratore superi quella ad esso attribuita, fallendo l'operazione;
- *timing = 1*, se l'ottimizzatore termina prima del limite di tempo imposto, con questo comando viene specificato anche il tempo impiegato per la risoluzione (default 0, cioè non specificato);
- *prestats = 1*, riporta prima dell'azione del solver, le statistiche sul numero di vincoli e variabili che CPLEX andrà ad analizzare (default 0);
- *threads =1*, obbliga CPLEX a utilizzare solo uno dei vari risolutori a disposizione al suo interno, affinché il tempo riportato non sia la somma dei tempi di più ottimizzatori.

- Una volta lanciati i due algoritmi, prima di attivare il risolutore bisogna fare alcune valutazioni:
- se l'UB e LB hanno lo stesso valore, allora si è già trovata la soluzione ottima;
- altrimenti, si attribuisce ad ogni elemento della clique un colore diverso, in questo modo CPLEX individuerà più velocemente la soluzione iniziale da cui far partire l'analisi.

```
model C:\Users\Lorenzo\Desktop\AMPL\FileCompleti\VCP.mod;
data C:\Users\Lorenzo\Desktop\AMPL\FileCompleti\DSJC125.9.dat;
option solver cplex;
option cplex_options 'time=300 timing=1 prestats=1 threads=1';
option show_stats 1;

include C:\Users\Lorenzo\Desktop\AMPL\FileCompleti\SEQ.run;
include C:\Users\Lorenzo\Desktop\AMPL\FileCompleti\LB.run;

var Z{k in K} binary;

if mincolore == maxcolore
    then {
        print "la soluzione è:";
        print mincolore;
        }
    else {
        for {v in VERTICI}  {
                for {c in COLORI} {
                    let X[v,c] := 0;
                    };
            };

        let COLORI := 1..maxcolore;
        for {k in K} {
            for {c in COLORI} {
                if Y[c] !=1 and Z[k] = 0
                    then {
                        let X[k,c] := 1;
                        let Y[c] := 1;
                        let Z[k] := 1;
                        }
                    };
                };
        solve;
        }

reset;
```

Fig. 12

5. ESPERIMENTI COMPUTAZIONALI

5.1 Caratteristiche Elaboratore

Modello e codice integrato sono stati eseguiti, per le diverse istanze, su un processore Intel® Core™ i7-5500U CPU @ 2.40GHZ, con 8 GB di RAM installata.

5.2 Caratteristiche Solver

Il solver utilizzato per risolvere il problema di Vertex Coloring è CPLEX. Esso fa parte del pacchetto IBM ILOG CPLEX Optimization Studio 12.6.3.0, un ambiente integrato per il supporto del processo decisionale analitico, utile per sviluppare velocemente modelli di ottimizzazione con l'ausilio della programmazione matematica. Questo *toolkit*, cioè insieme di strumenti software, permette di ottimizzare le decisioni di business con motori di ottimizzazione ad alte prestazioni, creando applicazioni realistiche (simulazioni) in grado di migliorare in modo significativo i risultati aziendali.

Pur interfacciandosi a noi come un unico elemento, CPLEX è al suo interno composto da differenti solver, ciò permette di risolvere un'ampia gamma di classi di problemi:

- i classici problemi di *Integer Linear Programming* (ILP), a cui appartiene, ad esempio, il Vertex Coloring Problem;
- problemi di *Quadratic Programming* (QP), dove la funzione obiettivo contiene incognite elevate al quadrato; se anche i vincoli contengono termini quadratici, allora sono detti problemi di *Quadratically Constrained Programming* (QCP);
- *Mixed-Integer Programming (MIP),* dove le variabili possono includere anche valori semi-continui.

CPLEX, per i problemi di programmazione lineare intera (ILP), utilizza la tecnica del *Branch&Bound*, metodo esatto di esplorazione dello spazio delle soluzioni basata sull' enumerazione implicita, cioè non analizza l'intero poliedro dove è ammessa una soluzione, ma, come dice il nome stesso, la ricerca si compone di due fasi:

- *Branching*: analisi ricorsiva di sottoinsiemi disgiunti di soluzioni: è la tecnica per generare sotto-problemi; dal momento che non vogliamo che diventino troppo numerosi (l'obiettivo è velocizzare la risoluzione del problema) interviene il secondo step;
- *Bounding*: valuta la partizione sulla base di una stima del miglior valore che la funzione obiettivo assume in ciascun sottoinsieme. Se il risultato ottenuto non è meglio di quello che già possiedo, si eliminano i sottoinsiemi che certamente non contengono la soluzione ottima.

Per velocizzare la valutazione della partizione, si considera di effettuare un rilassamento dei vincoli più difficili, quali l'interezza del valore assunto dalle variabili.
Tuttavia, per problemi quali il Vertex Coloring Problem, lo spazio delle soluzioni da esplorare rimane eccessivamente ampio: pur trattandosi di un algoritmo esatto, ovvero che si ha la certezza che riporti un risultato esatto (a meno di errori di valutazione), il tempo o la memoria richiesta per mantenere le possibili soluzioni non risultano sufficienti o adatti alle nostre richieste.

Inoltre, sequenza delle possibili soluzioni prese in considerazione può variare in funzione della strategia scelta per selezionare il successivo nodo da analizzare; infatti, anche solo cambiando l'ordine dei vincoli nella definizione del modello studiato, ciò causerebbe la modifica della strategia di branching, variando il *running-time* necessario alla risoluzione.

Nel momento in cui non riesce a trovare una soluzione al problema, CPLEX riporta il miglior valore della funzione obiettivo fino a quel momento trovato (Upper Bound), e, indirettamente, attraverso un gap, il minimo che deve assumere (Lower Bound).

5.3 Caratteristiche Istanze

Le istanze scelte per questo modello sono diverse, prese dal sito http://mat.gsia.cmu.edu/COLOR/instances.html, rispettano il formato standard DIMACS.

DIMACS (Centre for Discrete Mathematics and Theoretical Computer Science) è una collaborazione tra Rutgers University e Princeton University fondata nel 1989. Il centro universitario promuove la ricerca e la divulgazione di metodi matematici e statistici della matematica discreta e dell'informatica teorica.
DIMACS si pone due obiettivi: finanziare progetti di ricerca per incoraggiare lo sviluppo di nuovi

algoritmi performanti, e stabilire uno standard su cui confrontare i risultati ottenuti dai diversi partecipanti ai programmi. Al 2016, sono quattro i progetti attivi presso il DIMACS, focalizzati su: Cybersecurity, Energia, Crittografia e Condivisione Informazioni-Analisi Dinamica dei Dati.

Un file salvato secondo lo standard DIMACS, deve seguire una precisa costruzione:

- nelle iniziali righe di commento (introdotte dalla lettera "c") forniscono tutte le necessarie informazioni sull'istanza in questione;

    - $c\ This\ is\ an\ example\ of\ a\ comment$

- segue la Problem Line, unica linea in cui si definiscono le proprietà fondamentali del grafo, identificabile perché inizia con "p"; nella riga viene indicata una parola chiave per identificare la classe del problema, il numero di vertici e di archi;

    - $p\ edges\ 12\ 158$

- infine i vari archi sono sempre introdotti dalla lettera "e", seguita da una coppia di valori distanziati tra loro da un solo spazio vuoto.

$$e\ 1\ 25$$

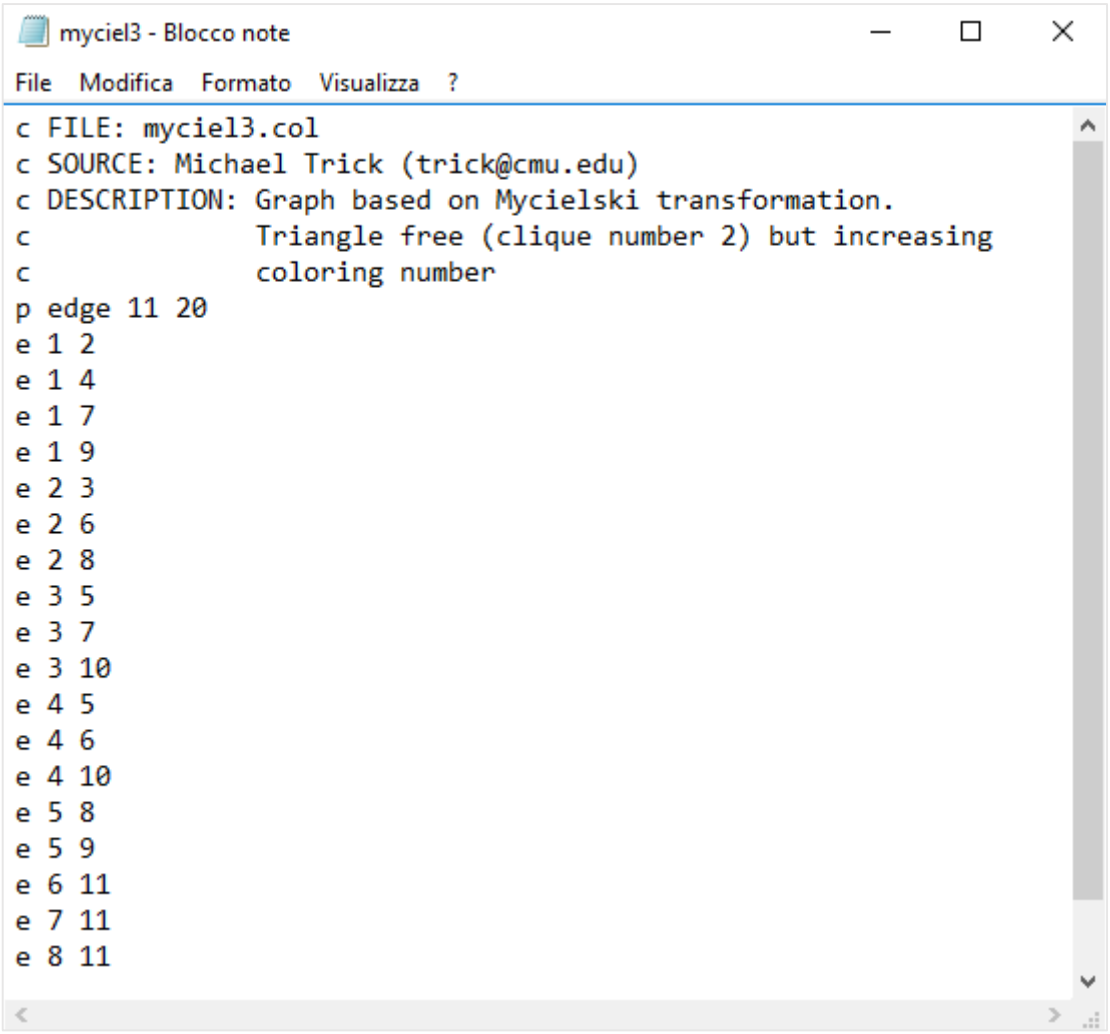


Fig. 13

DIMACS propone una grande varietà di grafi, creati casualmente (random) o con una particolare origine; tra le istanze scelte troviamo:

- grafi provenienti da Don Knuth's Stanford GraphBase; sviluppati da Don Knuth, professore emerito dell'Università di Stanford, traggono ispirazione da molteplici e interessanti tematiche. Le istanze con nomi propri sono "Book Graph": il grafo è ricavato da grandi opere letterarie, ogni nodo è un personaggio dell'opera scelta, due nodi sono connessi se, nel libro, si incontrano reciprocamente. Ad esempio, *anna* sta per Anna Karenina, protagonista dell'omonimo romanzo di Tolstoj, *homer* per Omero, presunto autore dell'Iliade e dell'Odissea.

  I "Miles Graph" sono grafi ricavati da mappe stradali dei collegamenti tra città statunitensi

nel 1947; due nodi (città) sono connessi solo se abbastanza vicini. I "Queen Graph" sono ispirati ai movimenti di una regina su una scacchiera $n \times n$;

- grafi DSJC: sono grafi generati casualmente, utilizzati da David S. Johnson e compaiono nel suo articolo "Optimization by Simulated Annealing: an Experimental Evaluation; Part II";
- "Myciel Graph", così chiamati perché basati sulla trasformazione di Mycielski; sono grafi che risultano difficili da risolvere in quanto *triangle-free*, cioè la clique più grossa che possiamo trovare nel grafo può contenere al massimo 2 elementi.

5.4 Risultati sperimentali

In tabella, vengono riportate le informazioni utili relative ad ogni istanza presa in considerazione: le prime due colonne indicano la cardinalità degli insiemi dei vertici V e dei lati E. Si riportano i risultati ottenuti elaborando solo il modello: CPLEX restituisce sia LB che UB, ma, nel momento in cui hanno lo stesso valore, significa che è stato trovato l'ottimo entro il limite di tempo imposto (300 secondi); viene allegato infine il running-time, ovvero il tempo di elaborazione dei risultati.

Le ultime cinque colonne si riferiscono, invece, all'elaborazione del codice integrato: le prime due sono le soluzioni riportate dagli algoritmi di Lower Bound (LB_i) e Upper Bound (UB_i). Le ultime tre colonne sono i risultati del modello, equivalente a quello precedentemente descritto, con l'integrazione dei due algoritmi greedy.

			Modello			Codice Integrato				
Nome_istanza	**n**	**m**	**LB**	**UB**	**t[sec]**	LB_i	UB_i	LB_f	UB_f	**t[sec]**
anna	138	986	11	11	6.1094	7	12	11	11	0.1250
david	87	406	11	11	1.5469	11	12	11	11	0.0938
homer	561	3259	0	561	OUT	10	15	13	13	1.6875
huck	74	602	11	11	0.4688	5	11	11	11	0.0313
jean	80	508	10	10	0.6094	4	10	10	10	0.0313
miles250	128	774	8	8	5.9844	8	9	8	8	0.0625
miles500	128	2340	20	20	15.0469	16	22	20	20	0.9375
miles750	128	4226	31	31	24.0156	22	34	31	31	2.5938
miles1000	128	6432	42	42	32.6875	32	44	42	42	6.6094
miles1500	128	10396	71	73	OUT	55	76	73	73	36.9688
myciel3	11	19	4	4	0.0469	2	4	4	4	0.0156
myciel4	23	71	5	5	0.8750	2	5	5	5	0.1250
myciel5	47	236	4	6	OUT	2	6	6	6	25.9531
queen5	25	320	5	5	0.1719	5	8	5	5	0.0126
queen6	36	580	6	7	OUT	4	11	7	7	53.9375
queen7	49	952	7	7	52.2812	5	10	7	7	1.0000
DSJC125.1	125	736	0	125	OUT	4	8	5	5	16.6719
DSJC125.5	125	3891	0	125	OUT	9	26	9	26	OUT
DSJC125.9	125	6961	0	109	OUT	27	36	27	36	OUT

Fig.14

6. CONCLUSIONI

A conclusione del progetto di tesi, in questo capitolo verranno analizzati i risultati dei test effettuati sulle due diverse soluzioni sviluppate durante il lavoro di questa tesi, esaminando i tempi di elaborazione ottenuti sulle istanze precedentemente presentate.

Riassumendo il lavoro discusso nei precedenti capitoli, si è focalizzata l'attenzione sull'implementazione del modello di Vertex Coloring Problem nel linguaggio AMPL. Si è data una descrizione del funzionamento dell'ottimizzatore utilizzato, e, in funzione delle sue peculiarità, si è valutato di sviluppare gli algoritmi di Lower Bound e Upper Bound.

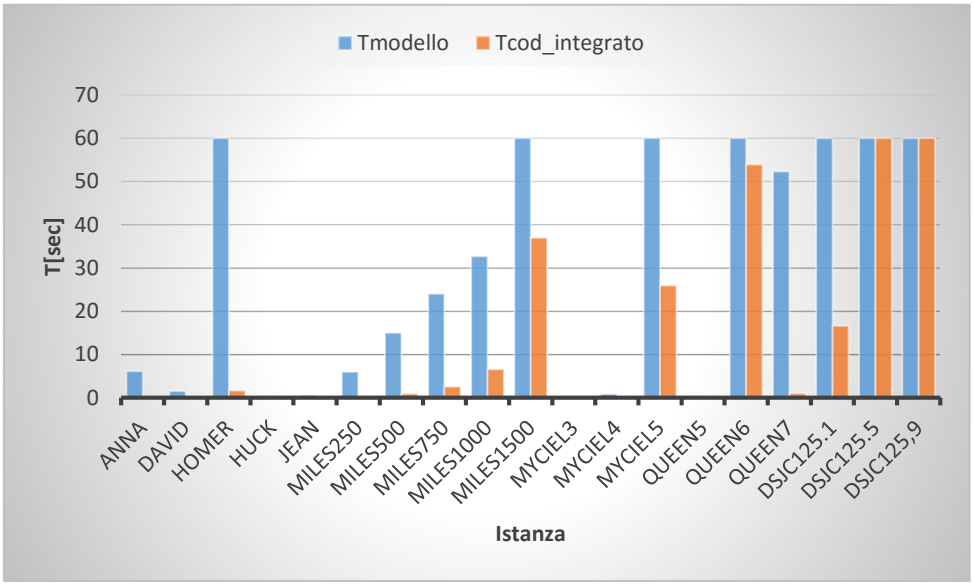
Le suddette tecniche per limitare il poliedro delle soluzioni ammissibili del Vertex Coloring non rientrano tra le più attuali e innovative, dal momento che la performance degli algoritmi non era parte del progetto; esiste a riguardo una ampia letteratura di algoritmi euristici per la risoluzione del Vertex Coloring Problem, che integrano diversi metodi euristici, anche al di fuori della categoria degli algoritmi greedy.

Come affermato nell'introduzione, l'obiettivo di questa tesi è invece dimostrare che, tramite l'utilizzo di semplici accorgimenti (quali i due algoritmi presentati), è possibile aumentare esponenzialmente la possibilità di risolvere problemi, che altrimenti rimarrebbero senza soluzione.

Il fatto che il Vertex Coloring Problem appartenga alla classe dei *NP-hard problem*, implica che non esista un rapporto proporzionale tra le proprietà di una singola istanza: non si può ricercare alcuna relazione di proporzionalità tra numerosità di nodi ed archi e tempo di risoluzione, tantomeno fra i *running-time* del solo modello rispetto al codice integrato. Non è fattibile conoscere, infatti, l'onerosità dell'attività di *Branch&Bound* eseguita dall'ottimizzatore, in termini di valutazione di ogni singolo sottoinsieme analizzato: il rilassamento di una potenziale soluzione ottima potrebbe risultare più difficoltoso rispetto a una qualsiasi altra, a prescindere dalla cardinalità dell'istanza analizzata.

Per i motivi sopra citati, ci limiteremo ad evidenziare i vantaggi che l'integrazione degli algoritmi di Lower Bound e Upper Bound apportano al modello.

Per facilitare la lettura dei dati riportati nella Tab.1, si riporta una rappresentazione grafica dei risultati, confrontando, per ogni istanza, il tempo impiegato nella risoluzione prima del solo modello (blu) e del codice integrato (arancione). Come si può notare, spesso i tempi rappresentati sono così piccoli da non comparire neanche nell'istogramma, paragonandoli agli altri. Dove il tempo raggiunge il valore massimo nel grafo, significa che è stato raggiunto il *time-limit* imposto a ciascun problema.



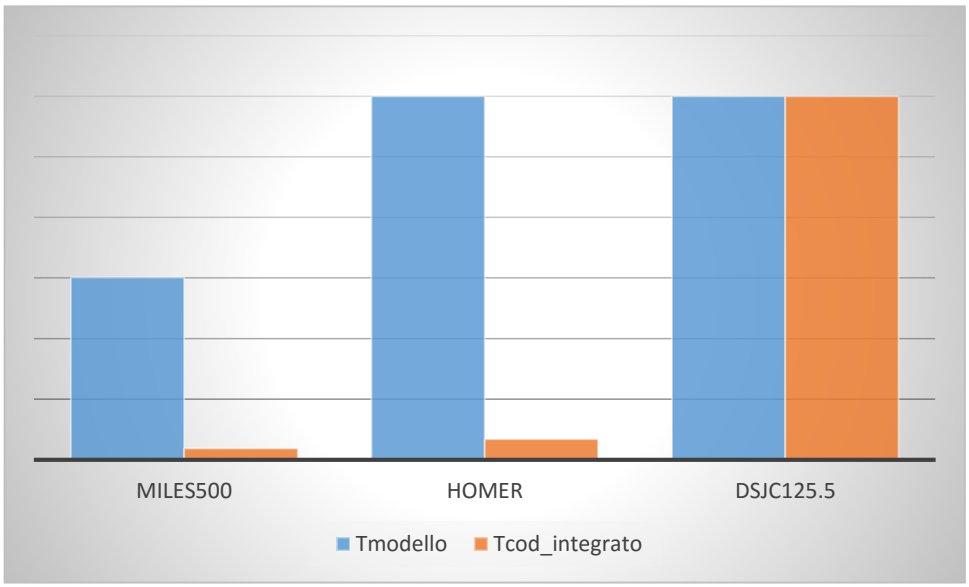
Tab. 1

Al fine di analizzare i 19 grafi selezionati, possiamo innanzitutto dividerli in tre categorie:

- istanze dove anche il modello lanciato da solo riesce, entro il tempo limite, a trovare l'ottimo, tuttavia il tempo impiegato dalla soluzione implementata in questo elaborato risulta nettamente più efficiente;
- istanze in cui il modello supera il tempo limite, mentre il codice integrato trova rapidamente una soluzione;
- istanze dove nessuno dei due trova una soluzione, ma gli algoritmi ci permettono di limitare il gap tra LB e UB, e per lo meno avere il valore di UB come soluzione vicina all'ottimo.

Assumendo tre casi “campione” (riportati in Tab.2), rappresentativi di ognuna delle categorie prima accennate, portiamo come esempio i seguenti grafi:

- miles500, in questo caso, il solver termina entro il *time-limit* la risoluzione del problema, tuttavia l’utilizzo del codice integrato riduce del 1600% il tempo d’elaborazione;
- homer, il tentativo di elaborazione esclusiva del modello supera il tempo limite, mentre il codice integrato viene risolto in tempi molto contenuti, ben sotto ai 2 secondi, trovando rapidamente l’ottimo;
- DSJC125.5, non viene risolto in nessun caso; nell’elaborazione del solo modello, purtroppo CPLEX riporta un gap = 125 (LB = 0, UB=125), invece il codice integrato riduce il gap pari a 17 (LB = 9 e UB= 26).



Tab. 2

Grazie a queste “istanze-benchmark”, proposte secondo lo standard DIMACS, siamo riusciti a dimostrare l’obiettivo proposto nell’introduzione all’elaborato; infatti, con l’ausilio di semplici strumenti, quali gli algoritmi euristici *greedy*, sono facilmente migliorabili le performance e soluzioni del problema. Da evidenziare, il ruolo che AMPL ha giocato nell’implementazione dei suddetti algoritmi: grazie alla sua intuitiva sintassi, permette a qualsiasi utente di prescindere dalla conoscenza delle strutture dati (vettori, matrici etc.) richieste dall’ottimizzatore.

Concludendo, questo approccio ai problemi di modellazione non si dimostra valido solo per il particolare caso del Vertex Coloring Problem affrontato in questa tesi, ma risulta efficace per un qualsiasi problema difficile, del quale l’utente sia in grado di riconoscere la struttura e vincoli di cui si compone.

BIBLIOGRAFIA

Clausen J. 1999, *Branch and Bound Algorithms – Principles Examples* University of Copenhagen

Cordone R. Bruglieri M. Liberti L. 2001, *Appunti sul linguaggio di programmazione AMPL*. Politecnico di Milano

Culberson J.C. , Luo F. 1995, *Exploring the k-colorable Landscape with Iterated Greedy*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science

Erickson J. 2014. *Algorithms and Models of Computation. lecture 30: NP-Hard Problems,*University of Illinois.

Fourer R. Gay D. Kernighan B., 1990 *A Modeling Language for Mathematical Programming.* Management Science 36 pp. 519-554.

Fourer R. Gay D. Kernighan B., 2003, *AMPL: A Modeling language for Mathematical Programming.* Second Edition

IBM Corporation, 2015. *IBM ILOG CPLEX Optimization Studio – Getting Started with CPLEX* (Version 12, Release 6).

Kaminsky P. 2003, *Introduction to AMPL: A Tutorial.* University of California, Berkeley

Malaguti E. 2006, *The Vertex Coloring Problem and its generalizations*. PhD Thesis, Università di Bologna

Malaguti E. Monaci M. Toth P. 2008, *A Metaheuristic Approach for the Vertex Coloring Problem*. INFORMS Journal on Computing Vol. 20, No.2 Spring 2008, pp. 302-316

Malaguti E., Monaci M., Toth P. 2011, *An exact approach fot the Vertex Coloring Problem,* Elsevier, Discrete Optimization pp.174-190

Ostegard P. 2002, *A fast algorithm for the maximum clique problem,* Elsevier Discrete Applied Mathematics pp.197–207

Pesenti R, 2005 *Modelli Lineari Interi/Misti*, Università Ca' Foscari Venezia

Vigo D. 2003, *Algoritmi euristici IV – Analisi delle Prestazioni*. Università di Bologna

Voloshin V. 2009, *Graph Coloring: History, results and open problems*, Alabama Journal of Mathematics

SITOGRAFIA

Algoritmo Greedy: http://www.uniroma2.it/didattica/MMOD1bis/deposito/AlgEuristici.pdf

Graph Coloring Instances: http://mat.gsia.cmu.edu/COLOR/instances.html#XXDSJ

DIMACS: http://mat.gsia.cmu.edu/COLOR/instances.html#XXDSJ

AMPL: http://ampl.com/