UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dipartimento di Tecnologie dell'Informazione
Corso di Laurea in Informatica

UN ALGORITMO DI PROGRAMMAZIONE MATEMATICA PER IL MAX EDGE-WEIGHTED CLIQUE PROBLEM WITH MULTIPLE CHOICE CONSTRAINTS

Relatore: Prof. Giovanni Righini

Tesi di Laurea di:
Alberto Bosio
Matricola 641326

Anno Accademico 2004/2005

Ringraziamenti

Ringrazio in particolare il Professor Giovanni Righini per l'aiuto, la pazienza e il sostegno dimostratomi durante lo sviluppo di questa tesi.

A tutti gli amici che hanno accompagnato il mio cammino universitario.

A Michela che mi è sempre stata vicina.

Ai miei genitori, grazie ai quali ho potuto raggiungere questo obiettivo.

Indice

STRUTTURA DELL'ESPOSIZIONE

Nel capitolo 1 viene esposto in maniera formale il *Max edge-weighted clique problem with multiple choice constraints* e le sue applicazioni.

Il capitolo 2 contiene richiami teorici, riguardanti il *Branch-and-Bound* e il *rilassamento Lagrangeano*.

Nel capitolo 3 viene esposta l'applicazione del *rilassamento Lagrangeano* al problema studiato e viene proposto un algoritmo *Branch-and-Bound* per il calcolo della soluzione ottima.

Infine, nel capitolo 4, sono descritti i risultati sperimentali ottenuti.

Capitolo 1

Introduzione

1.1 Scopo della tesi

Lo scopo di questa tesi è studiare, realizzare e valutare sperimentalmente un algoritmo esatto che risolva il *max edge-weighted clique problem with multiple choice constraints*, problema NP-Hard di ottimizzazione su grafo.
L'algoritmo proposto è un *Branch-and-Bound* basato sul *rilassamento Lagrangeano*.

1.2 Applicazioni

Nell'ambito della biologia computazionale si riscontra frequentemente quello che è definito come *side-chain placement problem* [4]. Il problema consiste nel trovare la conformazione lineare di una proteina in cui l'energia di legame dei singoli aminoacidi (e in generale di singole molecole) sia minima e che, nel contempo, assicuri un'adeguata stabilità strutturale (GMEC).
L'organizzazione spaziale degli atomi di una proteina viene detta conformazione. Questo termine si riferisce a qualsiasi stato strutturale che può, senza rompere nessun legame covalente, interconvertirsi in altro stato. Soltanto una, o poche, delle innumerevoli conformazioni teoricamente possibili di una proteina contenente centinaia di legami singoli è quella che tende a predominare nelle condizioni biologiche. Questa è di solito la conformazione termodinamicamente più stabile, quella che ha minore energia libera di Gibbs (G).
Per una descrizione più approfondita dell'ambito biochimico vedere [2] e [6].

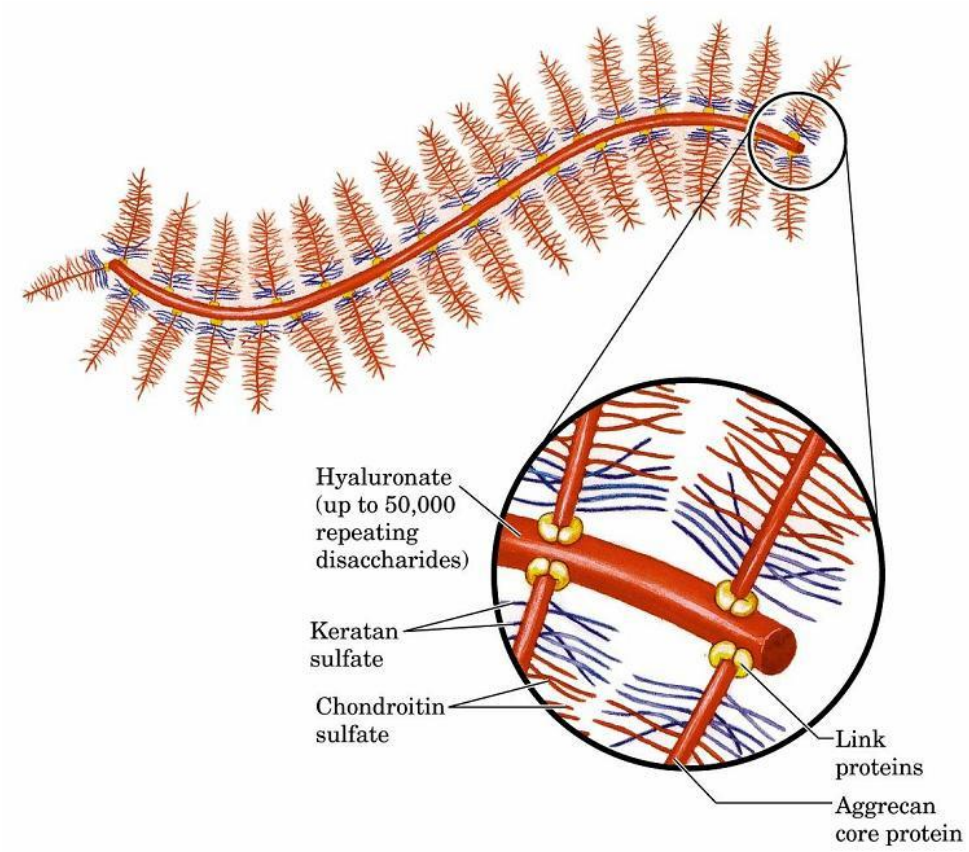

Figura 1.2.1 – aggregato proteoglicanico della matrice extracellulare

Una seconda applicazione dello stesso problema nasce nell'ambito della *management science* e riguarda l'interoperabilità delle imprese. Si considerino N uffici di una grande impresa, ognuno dei quali ha la facoltà di scegliere, a dati costi, uno tra diversi possibili protocolli per interagire con gli altri uffici. I costi di interazione tra due uffici dipendono dai due protocolli scelti, che possono risultare più o meno compatibili tra loro. L'obiettivo di questo problema è minimizzare i costi complessivi cioè i costi di adozione del protocollo scelto per ogni ufficio più i costi di interazione per ogni coppia di uffici.

Una terza applicazione riguarda il settore delle telecomunicazioni e in particolare la localizzazione dei nodi di *switch*. Si consideri una rete formata da diversi *clusters* di nodi. I nodi appartenenti al medesimo *cluster* comunicano tra loro direttamente, senza bisogno di *switches*. La comunicazione tra nodi in *clusters* diversi avviene invece indirettamente, tramite appositi *switches* che sono localizzati uno in ogni *cluster* e sono collegati direttamente tra loro come mostrato in figura.

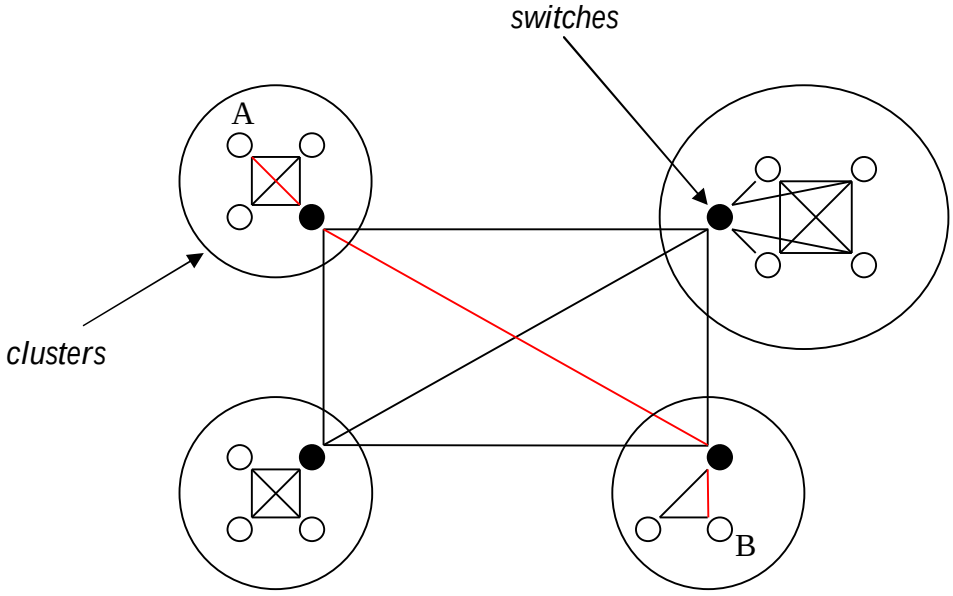

Figura 1.2.2 – struttura della rete ed esempio di connessione (in rosso) tra i vertici A e B appartenenti a *clusters* diversi

Il problema consiste nel localizzare gli *switches* su ogni *cluster* in modo da minimizzare i costi complessivi, che dipendono sia dalla composizione di ogni singolo *cluster* sia dai costi di interconnessione per ogni coppia di *switches*.

1.3 Formulazione del problema

Tutti i problemi descritti informalmente nella sezione precedente, possono essere formulati come il *max edge-weighted clique problem with multiple choice constraints*.

DATI:

Sia $G = (V, E)$ un grafo completo non orientato, dove V è l'insieme dei vertici ed E è l'insieme delle coppie di vertici (spigoli o *edges*). Ad ogni vertice $i \in V$ è associato un peso $c_i \in \Re$. Ad ogni spigolo $[i, j] \in E$ è associato un costo $d_{ij} \in \Re$. L'insieme V è partizionato in n sottoinsiemi $k \in \{1,...,n\}$.

VARIABILI:

Ad ogni vertice $i \in V$ è associata una variabile binaria x_i. Ad ogni spigolo $[i, j] \in E$ è associata una variabile binaria y_{ij}. Un vertice $i \in V$ e uno spigolo $[i, j] \in E$ si dicono attivi quando fanno parte del sottografo di $G = (V, E)$: $C = (U, F)$ dove $U \subset V$ ed $F = \{[i, j] \in E : i \in U \wedge j \in U\}$. Il sottografo in questione dovrà essere completo, cioè dovrà avere uno spigolo per ogni coppia di vertici. Un sottografo completo è una *clique*.

Quindi:

$$x_i = \begin{cases} 1 \text{ se il vertice } i \in U \\ \\ 0 \text{ altrimenti} \end{cases}$$

e

$$y_{ij} = \begin{cases} 1 \text{ se lo spigolo } [i, j] \in F \\ \\ 0 \text{ altrimenti} \end{cases}$$

VINCOLI:

In ogni sottoinsieme V_k un solo vertice $i \in V_k$ deve essere attivo.

Lo spigolo $[i, j] \in E$ è attivo se e solo se i nodi $i \in V$ e $j \in V$ sono attivi.

FUNZIONE OBIETTIVO:

L'obiettivo è massimizzare la somma dei pesi dei vertici e degli spigoli attivi.

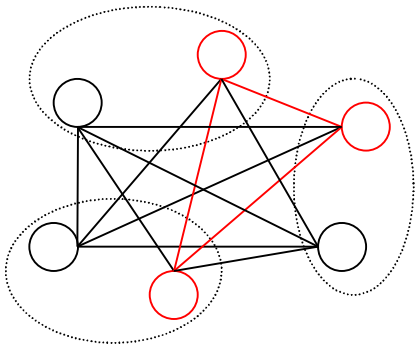

Figura 1.1.1 - grafo di esempio del problema

La figura 1.1 mostra i vertici e gli spigoli attivi in colore rosso. I cerchi tratteggiati sono i sottoinsiemi V_k del grafo. Si possono trascurare gli spigoli $[i, j] \mid i \in V_k$ e $j \in V_k$.

MODELLO:

Una formulazione ILP del *max edge-weighted clique problem with multiple choice constraints* è quindi la seguente:

$$\max \sum_{i \in V} c_i x_i + \sum_{[i,j] \in E} d_{ij} y_{ij} \qquad (1.1)$$

s.t.

$$x_i + x_j - 1 \leq y_{ij} \qquad \forall [i, j] \in E, \qquad (1.2)$$

$$y_{ij} \leq x_i \qquad \forall [i, j] \in E, \qquad (1.3)$$

$$y_{ij} \leq x_j \qquad \forall [i, j] \in E, \qquad (1.4)$$

$$\sum_{i \in V_k} x_i = 1 \qquad \forall k \in \{1, \ldots, n\}, \qquad (1.5)$$

$$x_i \in \{0,1\} \qquad \forall i \in V,$$

$$y_{ij} \in \{0,1\} \qquad \forall [i, j] \in E.$$

L'obiettivo (1.1) è massimizzare il costo dei vertici attivi e degli spigoli attivi.
Il vincolo (1.2) impone che due vertici non possono essere attivi se lo spigolo che li connette non è attivo.
I vincoli (1.3) e (1.4) impongono che uno spigolo può essere attivo solo se un vertice incidente in esso è attivo.
Il vincolo (1.5) impone che un solo vertice appartenente ad ogni sottoinsieme V_k deve essere attivo.
Il *max edge-weighted clique problem with multiple choice constraints* è una variante, non ancora studiata, del *max edge-weighted clique problem* (MEWCP). Una formulazione completa del MEWCP e la dimostrazione che è NP-Hard può essere trovata in [7]. A meno che P=NP, non è possibile trovare un algoritmo efficiente per risolvere il problema all'ottimo, ma diviene necessario ricorrere all'enumerazione di tutte le soluzioni. L'algoritmo, proposto in questa tesi, ne effettua una enumerazione implicita, attraverso una strategia *Branch-and-Bound* (vedi sezione 3.5).

Sorensen [12] ha sviluppato un algoritmo *Branch-and-Cut* per il MEWCP.

Hunting , Faigle e Kern [5] hanno adottato il *rilassamento Lagrangeano* per risolvere il MEWCP.

Park K., Lee e Park S. [9] hanno studiato una formulazione estesa del MEWCP attraverso l'uso di variabili addizionali per l'insieme dei vertici e di variabili naturali per quello degli spigoli.

Capitolo 2

Richiami teorici

2.1 Branch-and-bound

Il problema proposto è un problema NP-hard per cui, a meno che NP=P, per ottenere la soluzione ottima è necessario ricorrere all'enumerazione di tutte le possibili soluzioni. Questa enumerazione viene effettuata in maniera implicita ricorrendo ad algoritmi di *branching*, ovvero esplorando un albero in cui ogni nodo corrisponde ad un sottoproblema. La radice è il problema iniziale, in cui tutte le variabili sono libere. L'insieme di soluzioni corrispondenti (S_0) può essere partizionato in F sottoinsiemi S_k, che rappresentano le soluzioni di F sottoinsiemi $(S_1 \ldots S_F, U_{k=1\ldots F} S_k = S_0)$, fissando opportunamente il valore di una o più variabili libere. Ogni sottoproblema è radice di un nuovo albero; ai suoi figli corrispondono F sottoinsiemi dell'insieme di soluzioni S_k. Le foglie dell'albero di *branching* rappresentano tutte le possibili soluzioni del problema.

Attraverso l'utilizzo di algoritmi euristici è possibile trovare una soluzione iniziale ammissibile, non necessariamente ottima, che rappresenterà il *Bound Primale*; nel caso di un problema di massimizzazione sarà un *Lower Bound,* mentre in un problema di minimizzazione un *Upper Bound*. Gli algoritmi euristici permettono inoltre di completare le soluzioni parziali definite in ciascun sottoproblema.

Per ogni sottoproblema viene valutato il valore di una soluzione ammissibile duale, nel caso di massimizzazione sarà un *Upper Bound* mentre in un problema di minimizzazione un *Lower Bound*, sicuramente non peggiore di qualsiasi sua soluzione ammissibile primale. Questo valore viene detto *Bound Duale*. Per calcolare un *Bound Duale* si utilizzano dei rilassamenti. Un rilassamento di un problema si ottiene ignorando uno o più vincoli del problema stesso in modo da renderlo più semplice, al fine di trovare una soluzione in un tempo breve. Le tecniche di rilassamento più conosciute sono: il *rilassamento continuo*, il *rilassamento Lagrangeano* e il *rilassamento surrogato*. Se in un

sottoproblema il *Bound Duale* non risulta migliore del *Bound Primale*, la valutazione del nodo dell'albero di *branching* corrispondente e di tutti i suoi nodi-figlio può essere tralasciata.

E' necessario, infine, definire una strategia di ricerca. I metodi possibili sono l'espansione in larghezza (*breadth-first*), l'espansione in profondità (*depth-first*) e l'espansione sul problema più promettente (*best-first*).

Nell'espansione in larghezza si esaminano prima i nodi ad uno stesso livello. Tuttavia questa non è una tecnica molto utilizzata in quanto non si raggiungono velocemente le foglie, sebbene possa essere utile in alcuni casi.

L'esplorazione in profondità prevede di analizzare per primi gli ultimi sottoproblemi generati. In questo modo si raggiungono velocemente alcune foglie, ottenendo delle soluzioni ammissibili, che costituiscono un *Bound Primale*. Questa tecnica può essere utilizzata per raggiungere velocemente una foglia e passare poi ad altre tecniche.

Con la politica *best-first*, invece, viene data precedenza ai sottoproblemi che presentano un *Bound Duale* più promettente e che, probabilmente, conterranno soluzioni migliori.

2.2 Introduzione al rilassamento Lagrangeano

Il *rilassamento Lagrangeano* è una tecnica molto efficace utilizzata per rilassare problemi aventi una particolare struttura dei vincoli. L'applicazione della tecnica Lagrangeana ai problemi di ottimizzazione combinatoria e, più in generale, ai problemi di programmazione matematica intera, assume importanza negli anni '70. L'intuizione alla base dell'approccio Lagrangeano consiste nel vedere molti dei problemi di ottimizzazione combinatoria "difficili" come problemi "facili", complicati da un insieme, in genere di dimensione ridotta, di vincoli. Il *rilassamento Lagrangeano*, consiste nell'eliminazione dei vincoli "indesiderati", detti talvolta vincoli complicanti, e nell'inserimento di questi nella funzione obiettivo in modo tale che, nella soluzione del problema rilassato, se ne tenga conto in qualche misura. Alla funzione obiettivo verrà sottratto il valore della violazione di un vincolo complicante opportunamente pesato con un coefficiente detto *moltiplicatore Lagrangeano*, in modo da penalizzare le soluzioni inammissibili.
Dato un generico problema (in forma di massimizzazione):

$$\text{IP)} \quad \max z = cx$$

$$s.t. \begin{cases} Ax \le a \\ Bx \le b \\ x \in Z_+^n \end{cases}$$

dove il primo vincolo è costituito da m vincoli complicanti mentre il secondo da q vincoli semplici.
Otteniamo il rilassamento:

$$LR(\lambda) \quad \max z_{LR} = cx + \lambda(a - Ax)^T$$

$$s.t. \begin{cases} Bx \le b \\ x \in Z_+^n \end{cases}$$

dove il primo vincolo è un vincolo semplice e $\lambda \in \Re_+^m$ è un vettore di coefficienti, detti appunto *moltiplicatori Lagrangeani*. Poiché $\lambda_i \ge 0$, le violazioni su vincolo di tipo "$\le$" si risolvono in una penalizzazione nella funzione obiettivo.
Per mostrare che $LR(\lambda)$ è un rilassamento è sufficiente mostrare che ogni soluzione ammissibile per IP lo è anche per $LR(\lambda)$ e che per ogni soluzione x ammissibile per IP

si ha $z_{LR}(x) \geq z(x)$, e questo si può verificare sapendo che $\lambda_i \geq 0$ e $Ax \leq a$, da cui $\lambda(a - Ax) \geq 0$ e pertanto $cx + \lambda(a - Ax) \geq cx$.

$LR(\lambda)$ è un *Upper Bound* di IP per qualsiasi $\lambda \in \Re_+^m$. Il minimo *Upper Bound* è dato dalla soluzione del seguente problema, detto *Lagrangeano Duale* di IP:

$$\text{LD)} \quad z_{LD} = \min z_{LR}(\lambda)$$

$$s.t. \quad \lambda \in \Re_+^m$$

Per una descrizione formale del rilassamento Lagrangeano vedere [13].

Capitolo 3

Applicazioni al problema

3.1 Il rilassamento Lagrangeano

Il *rilassamento Lagrangeano* si adatta perfettamente al nostro problema. I vincoli (1.2), (1.3) e (1.4) possono essere infatti definiti come vincoli "complicanti". Possiamo, quindi, introdurli nella funzione obiettivo, sottraendone la violazione di ciascuno degli ij vincoli. Nel caso specifico, la violazione di ciascuno degli ij vincoli per (1.2) vale $(x_i + x_j - 1 - y_{ij})$, per (1.3) vale $(y_{ij} - x_i)$ e per (1.4) vale $(y_{ij} - x_j)$. (1.3) e (1.4) sono simmetrici e possono essere racchiusi in un unico vincolo che vale $(2y_{ij} - x_i - x_j)$. $(x_i + x_j - 1 - y_{ij})$ e $(2y_{ij} - x_i - x_j)$ sono moltiplicati rispettivamente per i *moltiplicatori Lagrangeani* $\lambda_{ij} \geq 0$ e $\mu_{ij} \geq 0$. Si ottiene così il nuovo problema:

LR:
max $z =$

$$\sum_{i \in V} c_i x_i + \sum_{[i,j] \in E} d_{ij} y_{ij} - \sum_{[i,j] \in E} \lambda_{ij} (x_i + x_j - 1 - y_{ij}) - \sum_{[i,j] \in E} \mu_{ij} (2y_{ij} - x_i - x_j) \tag{3.1}$$

s.t.

$$\begin{aligned} &\sum_{i \in V_k} x_i = 1 && \forall k \in \{1,\ldots n\}, \\ &x_i \in \{0,1\} && \forall i \in V, \\ &y_{ij} \in \{0,1\} && \forall [i,j] \in E. \end{aligned} \tag{3.2}$$

Eseguendo una riorganizzazione dei termini della funzione obiettivo (3.1), otteniamo la formulazione finale del nuovo problema:

$$LR:$$
$$\max z = \sum_{[i,j]\in E}(c_i - \lambda_{ij} + \mu_{ij})x_i + \sum_{[i,j]\in E}(c_j - \lambda_{ij} + \mu_{ij})x_j + \sum_{[i,j]\in E}(d_{ij} + \lambda_{ij} - 2\mu_{ij})y_{ij} + \sum_{[i,j]\in E}\lambda_{ij} \tag{3.3}$$

s.t.

$$\sum_{i\in V_k} x_i = 1 \qquad \forall k \in \{1,\ldots n\}, \tag{3.4}$$
$$x_i \in \{0,1\} \qquad \forall i \in V,$$
$$y_{ij} \in \{0,1\} \qquad \forall [i,j] \in E.$$

La funzione LR è definita *rilassamento Lagrangeano* del problema originario.
Per trovare il minimo *Upper Bound* dobbiamo risolvere il *Lagrangeano Duale* del problema iniziale ovvero:

$$\text{LD)}\ z_{LD} = \min z_{LR}$$

s.t.

$$\lambda_{ij} \in \Re_+$$
$$\mu_{ij} \in \Re_+$$

3.2 Il sottogradiente

Per ottenere un *Upper Bound* del sottoproblema che si sta analizzando tale che si avvicini il più possibile alla soluzione ottima, è necessario scegliere con accuratezza i *moltiplicatori Lagrangeani*. Per far ciò esistono diversi algoritmi; l'algoritmo applicato nel mio caso è quello del Sottogradiente o *Subgradient Optimization*. Questo metodo consiste in iterazioni successive in cui i *moltiplicatori Lagrangeani* vengono modificati, mediante un sottogradiente, verso una direzione possibile di miglioramento. L'algoritmo procede aumentando il valore dei *moltiplicatori Lagrangeani* per i vincoli che tendono ad essere violati, e diminuendolo per gli altri. Ad ogni iterazione t abbiamo due vettori di moltiplicatori: λ^t e μ^t. Con questi possiamo in seguito risolvere il problema LR ottenendo la sua soluzione ottima. Per fare ciò, per ogni sottoinsieme V_k attiviamo il vertice che da il valore maggiore per $(c_i - \lambda_{ij} + \mu_{ij}) \quad \forall [i, j] \in E$. Per quanto riguarda gli spigoli, invece, attiviamo quelli che aumentano il valore di LR. Dalla soluzione ottenuta calcoliamo due sottogradienti:

$$a_{ij}^t = x_i + x_j - 1 - y_{ij} \qquad \forall [i, j] \in E,$$

$$b_{ij}^t = 2y_{ij} - x_i - x_j \qquad \forall [i, j] \in E.$$

Da questi otteniamo il nuovo valore dei *moltiplicatori Lagrangeani*:

$$\lambda_{ij}^{t+1} = \max(0, \lambda_{ij}^t + \theta_t \cdot a_{ij}^t) \qquad \forall [i, j] \in E,$$

$$\mu_{ij}^{t+1} = \max(0, \mu_{ij}^t + \theta_t \cdot b_{ij}^t) \qquad \forall [i, j] \in E.$$

Per determinare un corretto valore di θ_t ho utilizzato la formula proposta da Beasley in [1]:

$$\theta_t = \frac{\pi_t (z_{ub} - LB)}{\sum_{i=1}^{n} (G_i)^2}$$

dove:

π_t è un parametro che assume valori che vanno da 2 a 0;

z_{ub} è il valore della soluzione del rilassamento;

LB è il valore della migliore soluzione ammissibile trovata, inizialmente calcolata tramite un algoritmo descritto successivamente;

$\sum_{i=1}^{n} (G_i)^2$ è la somma dei quadrati dei sottogradienti che modificano i rispettivi moltiplicatori.

Il parametro π deve decrescere con il numero delle iterazioni; per ottenere questo effetto si suggerisce di dimezzare π ogni 30 iterazioni (alle quali non è corrisposto un miglioramento di z_{ub}) e fermarsi quando π è minore di 0.005. Dopo una breve campagna di test ho deciso di far decrescere π moltiplicandolo, ad ogni iterazione, per un parametro $\beta < 1$. Questo parametro varrà 0.98 per il nodo radice, mentre per i nodi-figlio 0.92.

La scelta iniziale dei moltiplicatori non dovrebbe condizionare il risultato dell'algoritmo; tuttavia, essendo lo stesso inserito in un algoritmo *Branch-and-Bound*, i moltiplicatori vengono inizializzati al valore ottimo dei moltiplicatori del nodo padre e, ad ogni miglioramento di z_{ub}, aggiornati. L'unico nodo che ha tutti i *moltiplicatori Lagrangeani* pari a 0, solo per la prima iterazione, è il nodo radice.

In letteratura viene indicato come valore iniziale di π il valore $\pi_0 = 2$; tuttavia, poiché nei nodi-figlio si parte con i migliori moltiplicatori del nodo padre, risulta ragionevole utilizzare un valore iniziale più basso. Dopo numerosi test ho deciso di utilizzare per il nodo radice il valore indicato in letteratura, mentre per i nodi successivi ho assegnato il valore $\pi_0 = 0.3$.

Ho optato, per il nodo radice, a 290 iterazioni. Non ho voluto mettere una condizione di terminazione anticipata, come ad esempio un numero di iterazioni in cui z_{ub} non migliora, perché dai test effettuati ho notato che fino all'ultima iterazione il valore di z_{ub} migliorava. Inoltre la mia finalità era ottenere dei buoni *moltiplicatori Langrangeani* da cui partire. Per i nodi-figlio ho ridotto il numero di iterazioni a 30. Il numero delle iterazioni, come negli altri casi, l'ho scelto dai risultati dei test. Inoltre ho incrementato lo z_{ub} ottenuto del 5%, come proposto da Beasley. I risultati migliori, però, li ho ottenuti incrementandolo fino al 7%. La bontà di una tecnica di rilassamento dipende principalmente infatti da due fattori:

- qualità del bound prodotto;
- velocità di calcolo del bound.

A volte può essere preferibile scegliere bound meno "forti", ma computazionalmente meno onerosi, che bound migliori a fronte di tempi di calcolo elevati.

Quando la soluzione del problema Lagrangeano è ammissibile anche per il problema originario, non possiamo affermare che tale soluzione sia ottima per il sottoproblema che stiamo analizzando.
Se però la soluzione del problema Langrangeano è ammissibile per il sottoproblema e si verificano le seguenti condizioni:

$$\lambda_{ij}(x_i + x_j - 1 - y_{ij}) = 0 \qquad \forall [i, j] \in E,$$

$$\mu_{ij}(2y_{ij} - x_i - x_j) = 0 \qquad \forall [i, j] \in E,$$

allora la soluzione trovata è ottima per quel sottoproblema e possiamo interrompere le iterazioni del sottogradiente anticipatamente. Precedentemente avevo affermato che non ho voluto introdurre una terminazione anticipata, nel nodo radice; in effetti possiamo terminare le iterazioni anticipatamente ma, tranne nel caso che abbiamo delle istanze molto semplici o di dimensioni ridotte, ciò non avviene. La terminazione anticipata si verifica nei nodi-figlio quando circa metà delle variabili risultano fissate. Quando la soluzione ottima, trovata con l'algoritmo del sottogradiente, rispetta le condizioni sopra esposte, il nodo viene chiuso. Quindi se ciò avvenisse nel nodo-radice non si avrebbe la fase di *branching*.

3.3 Il lower bound

Per calcolare in maniera molto veloce un *Lower Bound* iniziale mi sono affidato a due algoritmi. Il primo sceglie un vertice, per ogni sottoinsieme V_k, in modo del tutto casuale; dai vertici scelti viene generato il sottografo completo. Dato che il grafo originario $G = (V, E)$ è completamente connesso, ad eccezione dei vertici dello stesso sottoinsieme V_k, la soluzione generata è sempre ammissibile. Poiché tale soluzione è dettata dal caso, a volte otteniamo dei valori che possono essere anche molto vicini al valore ottimo, mentre in altri casi i valori ottenuti risultano essere piuttosto scarsi. Per evitare di partire con un valore scarso ho adottato un secondo algoritmo che scegliesse il vertice, per ciascun sottoinsieme V_k, con la somma dei pesi degli spigoli, incidenti in esso, e il peso stesso del vertice, che siano i maggiori per quel dato sottoinsieme V_k. C'è da sottolineare che, sebbene questa sia un'ipotesi remota, è possibile definire con sicurezza che un vertice, di un certo sottoinsieme V_k, sarà sicuramente presente nella soluzione ottima, qualora il valore del peso del vertice e di ogni spigolo incidente in esso siano i maggiori in assoluto per quel gruppo. Il *Lower Bound* di partenza sarà il valore maggiore dato dai due algoritmi. Durante l'algoritmo, ogni qualvolta verrà generata una soluzione ammissibile, il valore di quest'ultima verrà confrontato con il valore del *Lower Bound* attuale. Nel caso in cui l'esito del test fosse positivo (cioè il valore della soluzione ammissibile generata è maggiore del *Lower Bound* attuale) il *Bound Primale* verrà aggiornato con il nuovo valore.

3.4 Euristica di completamento

Da ogni soluzione z_{ub} generata, è possibile costruire una soluzione ammissibile. Al termine dell'ultima iterazione dell'algoritmo del sottogradiente, nel caso non si fosse interrotto per il "ritrovamento" della soluzione ottima per quel sottonodo, si hanno dei vertici attivi che rispettano i vincoli del problema iniziale e degli spigoli attivi che, probabilmente, non li rispettano. E' comunque possibile generare una soluzione ammissibile da questi dati tracciando un sottografo completo, partendo dai vertici che sono selezionati all'ultima iterazione. I valori ottenuti mediante questo semplice metodo sono molto buoni: la soluzione ottima viene trovata molto presto, in genere dopo poche fasi di branching quando qualche variabile è già vincolata. La soluzione ottenuta può essere la soluzione migliore per quel sottonodo ma non ne abbiamo la certezza; è sicuramente una sua soluzione ammissibile. In questo modo, però, il *Lower Bound* migliora continuamente favorendo così la generazione di z_{ub} il più vicino possibili alla soluzione ottima per ogni sottoproblema, e la chiusura anticipata dei nodi.

3.5 Enumerazione implicita

Come già descritto nel paragrafo 2.1, l'enumerazione implicita delle soluzioni di un problema equivale alla visita di un albero di *branching*, i cui nodi rappresentano dei sottoproblemi. Per la generazione dei sottoproblemi-figli e per la loro esplorazione si possono adottare differenti metodi.

3.5.1 Strategia di ricerca

La strategia adottata nell'algoritmo sviluppato è quella del *best-first*. Dando la precedenza ai sottoproblemi che presentano un *Upper Bound* più promettente, si ha una maggiore probabilità che questi contengano la soluzione ottima. Adottando inoltre un algoritmo euristico di completamento si possono trovare molte soluzioni ammissibili.

3.5.2 Strategia di branching

Inizialmente il problema non ha alcuna variabile vincolata, fatta eccezione per il caso in cui l'algoritmo che genera la soluzione ammissibile iniziale determini che un vertice è sicuramente presente nella soluzione ottima. Quindi, escludendo il caso sopra descritto, il nodo radice contiene tutte le soluzioni possibili. A questo punto l'algoritmo esegue la procedura del sottogradiente, ed ottiene:

- la soluzione del rilassamento del problema e il suo valore di z_{ub};
- i vettori λ e μ dei *moltiplicatori Lagrangeani* che corrispondono al rilassamento,
- il valore ottimo del sottoproblema (ma solo nel caso in cui vengano rispettate le condizioni poste nel paragrafo 3.2).

Se il valore ottimo del sottoproblema è stato trovato, il nodo viene chiuso, altrimenti si procede con le operazioni in seguito descritte.
Se il *Lower Bound* è maggiore o uguale a z_{ub} il nodo viene chiuso. Questo controllo risulta inutile se effettuato al nodo radice, anche se il *Lower Bound* generato inizialmente fosse la soluzione ottima. Lo z_{ub} generato risulta essere sicuramente maggiore di tale soluzione. Se il nodo non è stato chiuso viene generata una soluzione ammissibile tramite l'euristica di completamento. Il nodo viene poi inserito nella lista dei nodi aperti rispettando la strategia di visita prescelta (3.5.1). Inizia quindi un ciclo che estrae il primo

nodo della lista e controlla se esso contiene ancora soluzioni promettenti. Nel caso in cui il nodo contenga soluzioni promettenti, il problema viene scomposto in sottoproblemi. Il nodo-padre viene scomposto in f nodi-figlio, dove f è il numero dei vertici presenti nel sottoinsieme V_k (la prima fase di branching avviene sul sottoinsieme $V_k \mid k = 1$). In ogni nodo-figlio viene attivato un vertice diverso dagli altri. In questo modo tutti i vertici del sottoinsieme V_k saranno attivi ma in nodi-figlio diversi. Mantenendo una variabile che indica in quale gruppo successivo il nodo-padre deve essere scomposto evitiamo di controllare, tutte le volte che estraiamo un nodo, quali vertici sono stati già attivati. Il ciclo termina quando la lista si svuota. A questo punto il valore della migliore soluzione intera nota rappresenta la soluzione del nostro problema.

Capitolo 4

Risultati sperimentali

4.1 GLPK: GNU Linear Programming Kit

Per verificare la correttezza e la bontà dell'algoritmo realizzato ho adottato un solutore di Programmazione Lineare Intera. Il solutore adottato è GNU Linear Programming Kit Version 2.4 [3], meglio conosciuto come GLPK. Un solutore lineare è un applicativo che riceve in ingresso un modello matematico ed i dati che caratterizzano un'istanza del problema restituendo poi una soluzione ottima. Tipicamente un solutore lineare determina la soluzione con il metodo del simplesso e le sue varianti. In genere può anche risolvere problemi di PLI attraverso algoritmi di *Branch-and-Bound* predefiniti. GLPK, come altri solutori, può essere utilizzato:

1 come libreria esterna che applica il solutore sia a strutture dati condivise che a strutture dati su file;

2 come programma stand alone che processa il modello ed i dati definiti tramite file, grazie ad un generatore algebrico interno.

GLPK, inoltre, ha anche il vantaggio di permettere la definizione del modello tramite il linguaggio GNU Math Prog [8]. GNU Math Prog, o solamente Math Prog, è un sottoinsieme del linguaggio di programmazione matematica AMPL. Math Prog permette di descrivere il modello in modo molto intuitivo e naturale. Gli elementi base di un modello Math Prog sono qui in seguito descritti:

Insiemi: rappresentano il dominio dei valori sia dei parametri che delle variabili;

Parametri: sono i dati, eventualmente organizzati in vettori mono o pluridimensionali, i quali possono essere caricati anche da altri file;

Variabili: descrivono la soluzione ed è il compito del solutore fissarne il valore al termine dell'esecuzione;

Vincoli: definiscono il problema distinguendo tra soluzioni ammissibili e non ammissibili;

Funzioni obiettivo: necessarie a valorizzare le soluzioni generate; il solutore ne ottimizza una sola durante la sua esecuzione, ma al termine valuta il valore di tutte.

Come ci si attende inoltre da un linguaggio ad alto livello, è possibile aggiungere commenti in qualsiasi punto del testo. La grammatica è libera dal contesto o context free. Ogni istruzione è terminata da un punto e virgola. E' possibile spezzare un'istruzione complessa su più righe, permettendo di aggiungere spazi, tabulazioni o righe vuote, al fine di facilitare la lettura del modello stesso.

glpsol è il solutore lineare presente nel pacchetto GLPK. E' un'applicazione richiamabile solo da linea di comando, per la quale non esiste un'interfaccia grafica. Il suo utilizzo base richiede l'indicazione di uno o più file contenenti il modello, i dati ed eventualmente il nome del file sul quale scrivere i risultati. Per fare ciò basta digitare su linea di comando:

```
glpsol --model nome.mod --data nome.dat --output nome.sol
```

Inizialmente glpsol risolve il problema come un problema di PL. Se il modello è effettivamente di PL, il processo termina con la soluzione. Se invece si tratta di un modello di PLI, si è di fatto risolto il rilassamento continuo del problema. In questo modo glpsol può verificare l'eventuale insoddisfacibilità e, eventualmente, determinare un bound primale del problema. Successivamente glpsol, basandosi sulla soluzione rilassata, determina una soluzione ottima del problema di PLI.
Durante il processo, glpsol invia allo standard output un insieme di informazioni, tra le quali i valori della soluzione PL, quelli della migliore soluzione intera trovata fino a quel momento e il gap, espresso in percentuale, tra le due. Al termine del processo, glpsol riporta il tempo impiegato e la dimensione della memoria utilizzata. E' possibile inoltre intervenire sul comportamento di glpsol tramite alcuni parametri. Il parametro più interessante per il mio scopo è la limitazione del tempo di esecuzione. In assenza di indicazioni glpsol prosegue a risolvere il modello finché non trova la soluzione ottima o ne dimostra l'inammissibilità. Per un modello complesso questo può portare ad esecuzioni

che durano ore, se non giorni. Può essere quindi necessario limitare il tempo macchina dedicato alla soluzione del modello. Al termine del tempo prefissato possiamo quindi trovarci di fronte a tre casi:

glpsol trova la soluzione ottima, o dimostra che non esiste soluzione ammissibile, eventualmente terminando prima del limite fissato;

glpsol trova, nel tempo limite fissato, una soluzione ammissibile e una stima della ottimalità di questa;

glpsol non trova nessuna soluzione nel tempo limite fissato.

4.2 Strumenti e dati utilizzati

L'algoritmo *Branch-and-Bound* [10] è stato realizzato in C# [11] , sfruttando il paradigma della programmazione ad oggetti, compilato con Microsoft Visual Studio .NET. I test sono stati condotti su un PC con processore AMD Athlon a 1100 Mhz, dotato di 256 MB di RAM, sistema operativo Windows Xp Professional.

Dato che in letteratura non esistono istanze ho creato un algoritmo che le generi. Per verificare il comportamento dell'algoritmo in presenza di problemi diversi tra di loro ho diviso le istanze in quatto classi:

nella prima classe tutti i vertici e gli spigoli hanno un peso positivo;

dalla prima classe deriviamo poi la seconda classe ponendo a zero il peso di tutti i vertici.

La differenza, quindi, tra la prima e la seconda classe, sta nel valore dei vertici. Con questa operazione un'istanza "facile" per la prima classe può diventare una istanza "difficile" per la seconda classe oppure il contrario. Il valore dei vertici infatti può influenzare la complessità dell'istanza.

Nella terza classe tutti i vertici e gli spigoli hanno un peso che può essere positivo o negativo;

dalla terza classe deriviamo l'ultima classe ponendo, come nel caso della seconda classe, i vertici a zero. Il discorso fatto in precedenza vale anche in questo caso.

Per il calcolo degli spigoli ho adottato il seguente schema:

* $1 \le d_{ij} \le 1000 r^h$ $(1 \le d_{ij} \le 1000)$ per gli spigoli positivi,

* $-1000 r^h \le d_{ij} \le 1000 r^h$ $(-1000 \le d_{ij} \le 1000)$ per gli spigoli positivi e negativi,

dove $h \in \{1,...,5\}$ e $r \in (0,1)$.

Per il calcolo dei vertici, invece, ho seguito il seguente schema:

* $1 \le c_i \le 1000$ per i vertici positivi,

* $-1000 \le c_i \le 1000$ per i vertici positivi e negativi.

Ogni istanza è costituita da un certo numero di vertici. Il numero minimo di vertici è 30. Il numero massimo dei vertici non è definito a priori. Per calcolarlo ho utilizzato la seguente regola, applicata all'algoritmo sviluppato. Ho dato un time-out di 7200 secondi:

- se entro questo limite di tempo la soluzione ottima viene trovata aumento le dimensioni delle istanze;
- se la soluzione ottima non viene invece trovata fornisco la migliore soluzione ammissibile e il gap, in percentuale, tra il *Lower Bound* e il peggior *Upper Bound*; in questo caso però non aumento più la dimensione delle istanze.

Come riferimento ho adottato l'algoritmo sviluppato e non glpsol perché lo scopo dei test è anche verificare la dimensione massima delle istanze risolvibili, dal mio algoritmo, in un tempo limite prefissato. La complessità di un'istanza non è definita solo dal numero dei vertici che la compone, ma anche dalla dimensione dei sottoinsieme V_k. I vertici sono suddivisi, tra ogni sottoinsieme V_k, in ugual misura. Il numero dei vertici appartenenti ad un sottoinsieme V_k è 2 o 3 o 4 o 5. Dal nome del file di un'istanza è possibile risalire alle caratteristiche sopra elencate. La convenzione adottata è la seguente (il simbolo – indica la concatenazione):

numero della classe-

-numero dei vertici totali-

-numero dei vertici per gruppo

Un possibile esempio è il seguente:

1-6-2

Un'istanza con queste caratteristiche può corrispondere al seguente file di input:

NUMERO DEI GRUPPI:
3;
NUMERO DEI VERTICI:
6;
NOME DEI VERTICI:
x1 x2 x3 x4 x5 x6;
PESO DEI VERTICI:
117,5 839,3 474,9 352,1 39,4 680,4;
GRUPPO DI APPARTENENZA:
0 0 1 1 2 2;
MATRICE DI INCIDENZA:

	x1	x2	x3	x4	x5	x6
x1	∞	∞	732,6	616,7	386,5	92,4;
x2	∞	∞	29,4	447,8	899,2	9,5;
x3	732,6	29,4	∞	∞	123,7	15,1;
x4	616,7	447,8	∞	∞	274,9	111,7;
x5	386,5	899,2	123,7	274,9	∞	∞;
x6	92,4	9,5	15,1	111,7	∞	∞;

EOF

Figura 4.2.1 – file di input

Per ogni tipo di istanza sono stati generati cinque file, differenti tra loro per i valori generati. Lo scopo è di fornire un tempo medio per quel determinato tipo istanza. Finché tutte e cinque le istanze non superano il time-out aumento le dimensione del problema.

Nelle tabelle, sotto riportate, sono utilizzati i seguenti valori:

- Nome: indica il tipo dell'istanza;
- T(sec): indica il tempo impiegato, in secondi, per trovare la soluzione ottima; un * significa che il time-out è scaduto;
- Gap %: indica il divario percentuale tra il miglior Lower Bound e il peggior Upper Bound (espresso in percentuale);
- Mem(MB): indica la memoria impiegata in MegaByte .

Il valore che indica il tempo e la memoria è una media dei cinque file calcolati. Per quando riguarda il Gap % ho preferito indicare il range dei valori ottenuti. Nel caso in cui alcuni file superano il time-out ed altri no, verrà indicato il tempo seguito dal numero dei file (tra parentesi rotonde) sul quale è stata calcolata la media. La stessa cosa verrà fatta per il Gap %.

4.3 Risultati

In questa sezione sono raccolti tutti i risultati sperimentali.

Nelle tabelle 4.1, 4.2 ,4.3 e 4.4 riporto i risultati ottenuti con sulle istanze da $V_k = 2$.

Da queste si evince che le prestazioni dell'algoritmo sviluppato sono migliori di glpsol anche se la memoria utilizzata risulta sempre maggiore rispetto a quest'ultimo. Questo vale per tutte le tabelle riportate.

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
1-30-2	4	0	9	14	0	4,9
1-32-2	7	0	13	27	0	5,9
1-34-2	12	0	16	37	0	6,9
1-36-2	43	0	20	119	0	8
1-38-2	57	0	27	236	0	9,8
1-40-2	94	0	29	478	0	11,5
1-42-2	107	0	35	534	0	13
1-44-2	158	0	38	623	0	15
1-46-2	214	0	40	1323	0	17.6
1-48-2	397	0	45	2413	0	18,9
1-50-2	1393	0	50	*	13-15,4	21
1-52-2	5944 (3)	2,8-5 (2)	57	*	17,7-19	25
1-54-2	*	7-10	63	*	21-23	27

Tabella 4.1

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
2-30-2	7	0	10	13	0	4,8
2-32-2	10	0	14	23	0	5,8
2-34-2	11	0	15	27	0	6,8
2-36-2	37	0	25	93	0	7,9
2-38-2	107	0	30	192	0	9,7
2-40-2	121	0	32	427	0	11,2
2-42-2	128	0	35	574	0	12,9
2-44-2	155	0	38	789	0	14,5
2-46-2	224	0	41	1005	0	17
2-48-2	397	0	47	2413	0	18,5
2-50-2	894	0	49	6984	0	20
2-52-2	6751	0	60	*	7-11,3	24,9
2-54-2	*	3-5	63	*	15-19	26,9

Tabella 4.2

Notiamo che sulle istanze di classe 3 e 4 è stato possibile aumentare maggiormente le dimensioni del problema rispetto a quelle di classe 1 e 2. Infatti, anche se la complessità del problema è maggiore, l'algoritmo sviluppato e glpsol trovano più facilmente una soluzione ottima perché i dati, contenendo anche valori negativi, sono più semplici.

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
3-30-2	1	0	4	3	0	4,9
3-32-2	3	0	10	4	0	5,9
3-34-2	4	0	12	6	0	6,9
3-36-2	12	0	15	34	0	8
3-38-2	26	0	19	57	0	9,7
3-40-2	41	0	29	100	0	11,3
3-42-2	65	0	35	269	0	12,8
3-44-2	126	0	38	447	0	14,7
3-46-2	154	0	40	801	0	16,1
3-48-2	197	0	46	1248	0	18,9
3-50-2	512	0	50	2048	0	21,1
3-52-2	1057	0	57	4865	0	24,9
3-54-2	3234	0	61	6152	0	27,5
3-56-2	6589	0	63	*	7-18	32
3-58-2	7115 (1)	5-13(4)	65	*	21-23	36,3
3-60-2	*	14-25	67	*	26-34	41,2

Tabella 4.3

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
4-30-2	4	0	5	3	0	4,8
4-32-2	6	0	7	10	0	5,8
4-34-2	11	0	13	15	0	6,8
4-36-2	17	0	18	23	0	7,9
4-38-2	33	0	25	61	0	9,5
4-40-2	57	0	29	184	0	11,2
4-42-2	71	0	35	289	0	12,7
4-44-2	151	0	40	489	0	14,6
4-46-2	194	0	43	756	0	16
4-48-2	209	0	46	1268	0	18,7
4-50-2	617	0	51	2567	0	21
4-52-2	1259	0	55	5796	0	24,6
4-54-2	3684	0	58	*	3-7	27,2
4-56-2	6994	0	61	*	11-18	31,8
4-58-2	*	11-17	63	*	24-36	36

Tabella 4.4

Nelle tabelle 4.5, 4.6 ,4.7 e 4.8 riporto i risultati ottenuti con sulle istanze da $V_k = 3$.

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
1-30-3	5	0	5	8	0	4,7
1-33-3	12	0	13	20	0	6,2
1-36-3	22	0	18	28	0	7,8
1-39-3	54	0	29	206	0	9,7
1-42-3	87	0	35	439	0	12,2
1-45-3	305	0	48	1249	0	16,4
1-48-3	1111	0	54	4236	0	20
1-51-3	5742	0	60	*	7,5-12,9	25,3
1-54-3	*	6-11	63	*	15-18	30,9

Tabella 4.5

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
2-30-3	6	0	7	7	0	4,5
2-33-3	11	0	12	15	0	6,1
2-36-3	28	0	22	35	0	7,7
2-39-3	77	0	32	356	0	9,5
2-42-3	105	0	38	598	0	12
2-45-3	297	0	45	2236	0	16,1
2-48-3	1541	0	56	5898	0	19,7
2-51-3	6888	0	62	*	5,8-9,9	25
2-54-3	*	9,6-12,3	63	*	12-17,7	29,9

Tabella 4.6

Precedentemente avevo accennato di come i pesi dei vertici potessero modificare il rendimento dell'algoritmo. Nella tabella 4.7 e 4.8 notiamo che nelle istanze con le caratteristiche 3-60-3 sono state risolte dall'algoritmo, mentre quelle con le caratteristiche 4-60-3 hanno superato il time-out. Questo comportamento si è verificato anche nelle tabelle 4.5 e 4.6 in maniera ridotta.

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
3-30-3	1	0	4	2	0	4,7
3-33-3	3	0	5	4	0	6,2
3-36-3	5	0	6	7	0	7,8
3-39-3	12	0	12	15	0	9,7
3-42-3	26	0	21	55	0	12,2
3-45-3	34	0	26	129	0	16,4
3-48-3	61	0	30	291	0	20
3-51-3	133	0	40	544	0	25,3
3-54-3	371	0	47	3587	0	30,9
3-57-3	1596	0	56	6912 (2)	4-8,3(3)	36
3-60-3	3698	0	60	*	10-23	41,3
3-63-3	*	3-11	63	*	21-29	46,8

Tabella 4.7

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
4-30-3	4	0	6	6	0	4,5
4-33-3	9	0	8	11	0	6,1
4-36-3	15	0	10	17	0	7,7
4-39-3	33	0	18	51	0	9,5
4-42-3	65	0	30	216	0	12,1
4-45-3	107	0	32	456	0	16,2
4-48-3	268	0	33	731	0	19,8
4-51-3	389	0	41	4130	0	25,1
4-54-3	2654	0	50	*	5-17	30,6
4-57-3	4236	0	58	*	15-26	35,4
4-60-3	*	9-14	63	*	29-34	41,1

Tabella 4.8

Nelle tabelle 4.9, 4.10 ,4.11 e 4.12 riporto i risultati ottenuti con sulle istanze da $V_k = 4$. In queste tabelle il peso dei vertici non ha influito nel tempo di esecuzione.

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
1-32-4	4	0	5	9	0	5,3
1-36-4	9	0	9	40	0	7,4
1-40-4	127	0	30	212	0	10,1
1-44-4	158	0	35	322	0	13,5
1-48-4	1205	0	45	6014	0	17
1-52-4	3652 (4)	5 (1)	58	*	20-23	32,1
1-56-4	*	12-17	63	*	30-35	38,4

Tabella 4.9

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
2-32-4	7	0	6	9	0	5,2
2-36-4	20	0	9	47	0	7,2
2-40-4	159	0	29	658	0	10
2-44-4	287	0	62	2684	0	13,3
2-48-4	2369	0	53	6958 (1)	7-13(4)	16,8
2-52-4	5789 (4)	3 (1)	60	*	12-26	31,8
2-56-4	*	5-7	63	*	24-35	38

Tabella 4.10

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
3-32-4	1	0	5	3	0	5,3
3-36-4	2	0	7	5	0	7,4
3-40-4	3	0	10	10	0	10,1
3-44-4	58	0	22	105	0	13,5
3-48-4	72	0	30	358	0	17
3-52-4	136	0	35	698	0	32,1
3-56-4	569	0	43	3584	0	38,4
3-60-4	4256	0	58	*	13-20	43
3-64-4	*	2-17	66	*	19-26	48,8

Tabella 4.11

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
4-32-4	2	0	5	5	0	5,2
4-36-4	3	0	10	9	0	7,1
4-40-4	7	0	12	15	0	9,8
4-44-4	34	0	20	79	0	13,2
4-48-4	81	0	28	189	0	16,5
4-52-4	92	0	32	455	0	31,8
4-56-4	689	0	45	3699	0	38,3
4-60-4	5146	0	59	7159 (1)	3-15(4)	43
4-64-4	*	8-15	63	*	20-33	48,7

Tabella 4.12

Nelle tabelle 4.13, 4.14 ,4.15 e 4.16 riporto i risultati ottenuti con sulle istanze da $V_k = 5$. Nelle tabelle 4.13 e 4.14 notiamo che l'algoritmo sviluppato è riuscito a completare due istanze, di dimensioni maggiori, di classe 2 rispetto a quelle di classe 1. I vertici con valore nullo hanno reso il problema più semplice.

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
1-30-5	4	0	5	7	0	4,2
1-35-5	8	0	11	19	0	6,5
1-40-5	27	0	19	79	0	9,7
1-45-5	105	0	30	405	0	14
1-50-5	302	0	43	1422	0	19,8
1-55-5	2364	0	50	6848 (3)	8-13 (2)	25,2
1-60-5	5987 (4)	5,7 (1)	57	*	12-18	32,1
1-65-5	*	4-15	63	*	23-27,3	38

Tabella 4.13

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
2-30-5	3	0	4	5	0	4,1
2-35-5	10	0	12	15	0	6,4
2-40-5	32	0	22	105	0	9,6
2-45-5	127	0	31	512	0	13,8
2-50-5	298	0	43	1568	0	19,6
2-55-5	1548	0	48	4894	0	25
2-60-5	4785 (4)	6,3 (1)	54	*	2-5	31,9
2-65-5	6923 (2)	9-13 (3)	61	*	13-19	37,8
2-70-5	*	15-19	63	*	17-34	45,2

Tabella 4.14

Nelle tabelle 4.15 e 4.16 notiamo che l'algoritmo sviluppato è riuscito a completare due istanze, di dimensioni maggiori, di classe 3 rispetto a quelle di classe 4. A differenza dal caso soprastante i vertici con valore nullo hanno reso il problema più difficile.

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
3-30-5	1	0	4	1	0	4,2
3-35-5	3	0	4	4	0	6,5
3-40-5	6	0	6	11	0	9,7
3-45-5	10	0	12	15	0	14
3-50-5	54	0	27	107	0	19,8
3-55-5	108	0	30	487	0	25,2
3-60-5	348	0	44	1257	0	32,1
3-65-5	417	0	48	3578	0	38
3-70-5	1359	0	51	5871 (4)	5 (1)	46,2
3-75-5	5785 (3)	8-15 (2)	57	*	10-13	51
3-80-5	*	18-39	63	*	23-35	57,1

Tabella 4.15

	Algoritmo sviluppato			glpsol		
Nome	T(sec)	Gap %	Mem(MB)	T(sec)	Gap %	Mem(MB)
4-30-5	3	0	4	5	0	4,1
4-35-5	10	0	6	7	0	6,4
4-40-5	13	0	12	15	0	9,5
4-45-5	17	0	27	23	0	13,8
4-50-5	67	0	30	98	0	19,6
4-55-5	111	0	44	394	0	25
4-60-5	587	0	49	2648	0	31,8
4-65-5	1235	0	50	4871	0	37,9
4-70-5	5265	0	56	*	7-10	46
4-75-5	*	11-18	63	*	34-40	50,4

Tabella 4.16

4.4 Conclusioni

In questa tesi è stato proposto un algoritmo di tipo Branch-and-Bound basato sul rilassamento Lagrangeano per risolvere il max edge-weighted clique problem with multiple choice constraints. Il rilassamento adottato si è rivelato molto efficace ottenendo l'ottimo in breve tempo o un gap ridotto quando il time-out è scaduto. Confrontando i risultati dei test effettuati con l'algoritmo sviluppato e quelli ottenuti con glpsol, notiamo che i tempi di risoluzione sono sempre migliori, di sei o sette volte, e anche il gap ottenuto quando il time-out è scaduto è minore.

E' possibile apportare miglioramenti all'algoritmo. Si può migliorare l'euristica di completamento oppure modificare la fase di *branching*.

BIBLIOGRAFIA

[1] Beasley J.E. Lagrangean relaxation. The Management School Imperial College – Technical Report, 1992.

[2] David L. Nelson, Michael M. Cox. I principi di biochimica di Lehninger.

[3] GNU Linear Programming Kit. www.gnu.org/software/glpk/glpk.html

[4] Hong E., Lozano-Pérez T. Protein side-chain placement: probabilistic inference and integer programming methods.

[5] Hunting M., U. Faigle, and W. Kern. A Lagrangian Relaxation Approach to the Edge-Weighted Clique Problem. EJOR, 131 (2001), 119-131.

[6] Jan Koolman, Klaus-Heinrich Rohm. Testo atlante di biochimica.

[7] Macambira E. M., C. C. de Souza. The edge-weighted clique problem: Valid inqualities, facet and polyhedral computations. EJOR 123 (2000), 346-371

[8] Makhorin A. Modelling Language GNU MathProg. Draft Edition, Gennaio 2005.

[9] Park K., Lee K. and Park S. An extended formulation approach to the edge-weighted maximal clique problem. EJOR 95 (1996), 671-682

[10] Righini G. Guida alla realizzazione di algoritmi branch & bound. Techinical report, Dipartimento di Tecnologie dell'Informazione, Sede di Crema, 9 marzo 2000.

[11] Robinson, Cornes, Glynn, Harvey, McQueen, Moemeka, Nagel, Skinner, Watson. C# guida per lo sviluppatore. Wrox Press, 2001.

[12] Sorensen M. New Facets and a Branch-and-Cut Algorithm for the Weighted Clique Problem. EJOR, 154 (2004), 57-70.

[13] Wosley L.A. Integer programming. John Wiley & Sons, 1988