POLITECNICO DI TORINO

I Facoltà di Ingegneria
Corso di Laurea in Matematica per le Scienze dell'Ingegneria

Tesi di Laurea

Ottimizzazione topologica di reti di tipo Internet Protocol con il metodo del Local Branching

Relatore:
prof. Roberto Tadei

Candidato:
Flavio Cimolin

Dicembre 2003

Indice

1 Introduzione **2**

2 Il problema in esame **4**
2.1 Descrizione del problema 4
2.2 Modello di programmazione lineare 6
2.3 Implementazione del modello in Xpress Mosel 9

3 Metodi per la Programmazione Lineare Misto-Intera **12**
3.1 Branch and Bound 12
3.2 Euristiche basate su ricerca locale 14
3.3 Local Branching 17

4 Implementazione del metodo ed analisi dei risultati **24**
4.1 Implementazione del metodo 24
4.1.1 Perché il local branching 24
4.1.2 Soluzione iniziale 25
4.1.3 Tagli di local branching 25
4.1.4 Tempo limite 26
4.1.5 Imposizione di un Upper Bound 27
4.1.6 Struttura dell'algoritmo 27
4.1.7 Xpress IVE 28
4.2 Istanze analizzate 29
4.3 Risultati ottenuti 30
4.4 Analisi dei risultati 35

5 Conclusioni e sviluppi futuri **40**

Bibliografia **41**

A Listato del programma in Mosel IVE **43**

B Esempio di Istanza **55**

Capitolo 1

Introduzione

Il lavoro di tesi nasce nel contesto della progettazione di reti ottiche di telecomunicazioni di tipo IP (*Internet Protocol*). Il problema di *Network Design* consiste essenzialmente nell'ottimizzare la topologia di una rete, cioè nel determinare l'installazione di cavi tra coppie di nodi della rete che minimizzi il costo complessivo e che consenta di soddisfare un insieme di richieste di traffico.

Lo sviluppo di reti di telecomunicazioni di tipo IP è attualmente un problema di grande interesse, in connessione con la corrente e continua espansione di Internet, nel numero di utenti utilizzatori e di servizi offerti. Il futuro non può che prevedere sempre maggiori richieste da parte degli utenti, e dunque sarà importante progettare reti sicure e veloci, oltre che in grado di fornire servizi sempre più all'avanguardia.

La strategia di instradamento più utilizzata nelle reti IP è quella OSPF (*Open Shortest Path First*), secondo la quale ogni pacchetto di dati viene instradato su uno dei cammini minimi fra il nodo di partenza e quello di destinazione, e conseguentemente ogni flusso, costituito da tanti pacchetti di dati, viene suddiviso in parti uguali su tutti i cammini minimi possibili.

La formalizzazione matematica in un modello di Programmazione Lineare Misto-Intera di problemi di Network Design è stata affrontata ampliamente in un precedente lavoro di tesi [5], in cui tra l'altro si è tenuto conto anche della possibilità di studiare reti *robuste*, ovvero in grado di resistere ad eventuali malfunzionamenti di alcuni archi. In questa tesi si è considerato un modello relativamente più semplice, ma ciononostante sempre di notevole complessità computazionale, ed in particolare si è tentato l'approccio risolutivo con un metodo che si colloca in posizione intermedia fra il Branch and Bound esatto ed i metodi euristici. Si tratta del metodo del Local Branching, descritto ampiamente in [1], che consiste in una successione di *soft variable fixing*: non vengono scelte manualmente le variabili da fissare, ma è l'algoritmo stesso che decide quali sia più conveniente mantenere costanti e quali invece far variare.

Questo tecnica può essere implementata in Programmazione Lineare grazie all'inserimento di un particolare tipo di taglio, detto *taglio di Local Branching*, in grado di ridurre ad un piccolo sottoinsieme del politopo originale lo spazio di ricerca delle soluzioni.

Il metodo che ne deriva, pur essendo in linea di principio esatto, si comporta essenzialmente come una meta-euristica, e dunque è in grado di trovare in tempi brevi soluzioni molto prossime all'ottimo, ma senza poterne garantire l'ottimalità. Si propone allora in questa tesi un confronto tra questo metodo innovativo ed il più tradizionale Branch and Bound applicato al problema in esame.

I risultati rispecchiano la grande complessità del problema, che può essere risolto con queste tecniche solo per istanze di 6, 7 nodi, ancora troppo lontane da istanze reali, che dovrebbero tenere in conto almeno 15, 20 nodi. Il Local Branching, opportunamente settato, può però risultare nettamente migliore del Branch and Bound classico, soprattutto quando la ricerca è volta a determinare la migliore soluzione possibile in tempi brevi.

Capitolo 2

Il problema in esame

2.1 Descrizione del problema

Il problema che andremo ad affrontare si può descrivere sinteticamente nel modo seguente:

> *Dato un insieme di nodi collegati tra loro da una serie di archi (ciascuno dei quali con un determinato costo di instradamento), e dato un insieme di richieste fra coppie di nodi, determinare la topologia di costo minimo e l'allocazione della banda che permetta di instradare tutti i flussi nel modo ottimale.*

Andiamo ora a vedere più in dettaglio le varie componenti del problema. Possiamo anzitutto pensare l'insieme dei *nodi* (che chiameremo in seguito N) come a città in una data regione, collegate fra di loro da strade. Una strada fra due città vicine corrisponde ad un arco fra una coppia di nodi. Non necessariamente, dunque, fra ogni coppia di nodi esiste un arco attivabile. Nella formulazione matematica del modello, al fine di ridurre il numero di variabili utilizzate, conviene distinguere fra arco senza direzione fra il nodo i ed il nodo j (che indicheremo con $\{i, j\}$ e chiameremo *spigolo*) ed arco direzionale fra il nodo i ed il nodo j (che indicheremo con (i, j) e chiameremo *arco* vero e proprio). Per questa descrizione iniziale del problema confonderemo ancora archi con spigoli, evidenziando le differenze laddove necessario.

Su ogni via che congiunga una città con un'altra è possibile installare uno o più *link*. Un link è un cavo in fibra ottica che porta con sè una certa banda fissata (non necessariamente uguale su tutti gli archi della rete). Per come è costituito fisicamente, un link installato su un arco fornisce una capacità costante uguale in entrambe le direzioni, e il traffico che circola in un verso non influenza quello che circola nel verso opposto. Per ogni arco (bidirezionale, cioè per ogni spigolo) viene dunque assegnata la capacità dei

link che vi si possono installare (chiameremo tali valori $q_{\{ij\}}$) ed il relativo costo unitario (che chiameremo $c_{\{ij\}}$).

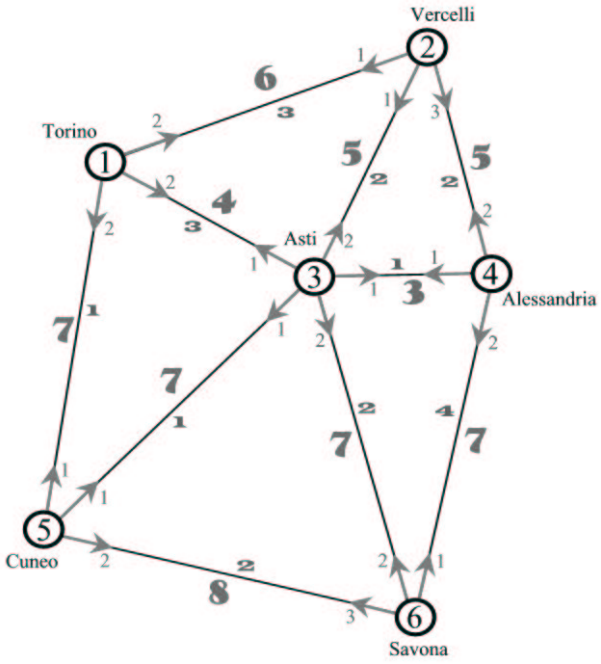

Figura 2.1: Esempio di grafo per il problema

Per ogni arco potenziale sono inoltre definiti dei pesi amministrativi (nel modello saranno chiamati w_{ij}) e, questa volta, non necessariamente tali costi sono simmetrici nelle due direzioni, cioè è possibile che un arco abbia un costo se percorso in un verso, ed un altro costo differente se percorso in senso inverso. E' importante notare che questi "costi di instradamento" non sono delle vere e proprie spese che vadano ridotte al minimo, ma devono piuttosto essere considerati come dei pesi, attraverso i quali il gestore della rete cerca di far fluire il traffico su certi archi piuttosto che su altri, al fine di ottimizzare alcuni parametri di QoS (*Quality of Service*). L'operazione di calibrazione dei pesi viene però effettuata in fase di simulazione, e di conseguenza nell'ambito del problema in esame ci limitiamo a considerare questi valori come dei dati del problema stesso, che rientreranno nel nostro modello in qualità di pesi per il calcolo dei cammini minimi.

In Figura 2.1 è rappresentato un esempio di grafo relativo al problema in esame, sul quale sono anche riportati i dati di una particolare istanza costruita in modo da essere il più possibile somigliante ad un caso reale (il file di dati contenente le matrici come vengono lette dal programma è visibile in Appendice B). I nodi sono allora costituiti da 6 città, e sono collegati da una struttura di 10 archi potenziali. Sugli archi sono indicati in

carattere grande i valori $c_{\{ij\}}$ corrispondenti al costo di installazione di un link (e sono proporzionali alla distanza reale fra le città), e più in piccolo i valori $q_{\{ij\}}$ relativi alle capacità dei link da installare. Inoltre i pesi w_{ij} sono stati rappresentati con delle freccette, dato che corrispondono al costo di attraversamento di un arco in una specifica direzione.

Il costo effettivo della rete così strutturata, che si vorrà appunto andare a minimizzare, è dato allora dalla somma dei costi di ogni singolo link installato. Per farlo, sarà necessario scegliere su quali archi installare dei link e quanti installarne. Per questo parleremo inizialmente solo di archi *potenziali*, che diventeranno archi *effettivi* della rete solo una volta che vi sia stato installato almeno un link.

L'insieme delle *richieste* (che indicheremo in seguito con K) è costituito da terne contenenti un nodo di origine (o_k), uno di destinazione (d_k), ed una capacità del flusso in questione (p_k). La topologia che verrà stabilita, oltre a dover naturalmente garantire che tutti i flussi possano giungere a destinazione, deve soddisfare alcuni principi generali che andremo ora a descrivere, e che devono essere tenuti in considerazione all'interno del modello. Anzitutto la strategia di instradamento che consideriamo è costituita dall'Open Shortest Path First protocol (OSPF), consistente nella richiesta che ciascun pacchetto di dati debba essere instradato su uno dei cammini minimi fra il nodo di origine ed il nodo di destinazione (cammino minimo calcolato in base ai costi w_{ij} degli archi della topologia che devono essere percorsi). Dato che ogni richiesta rappresenta un flusso costituito da molti pacchetti di dati, ciascun cammino minimo verrà utilizzato. In particolare, una approssimazione comunemente accettata è data dall'Equal Commodity Multi-flow principle (ECM): ogni flusso che giunge in un nodo in cui vi siano più (di uno) archi uscenti appartenenti a cammini minimi dal nodo stesso alla destinazione, viene suddiviso uniformemente su ognuno di essi. E' infatti possibile che per arrivare dal nodo di origine $\bar{o}$ al nodo di destinazione $\bar{d}$ ci siano vari percorsi alternativi tutti con lo stesso costo (minimo possibile). Allora la regola ECM impone che il flusso $\bar{p}$ venga suddiviso in parti uguali su ognuno di questi possibili cammini minimi.

2.2 Modello di programmazione lineare

Riportiamo di seguito il modello di PLMI del problema sopra descritto, come presentato in [2]. Consideriamo anzitutto i seguenti **insiemi di base**:

- N: insieme dei nodi
- K: insieme delle richieste
- E: insieme degli spigoli potenziali
- A: insieme degli archi potenziali

Utilizzeremo la notazione $\{i, j\}$ per indicare uno spigolo (senza direzione) fra il nodo $i \in N$ ed il nodo $j \in N$, e la notazione (i, j) per indicare un arco (direzionale) fra il nodo $i \in N$ ed il nodo $j \in N$.

Per rendere più agevole la scrittura del problema faremo uso inoltre delle seguenti notazioni:

- $o_k \in N$: nodo origine della richiesta k
- $d_k \in N$: nodo destinazione della richiesta k
- $D_K = \{v \in N : \exists k \in K,\ v = d_k\} \subset N$: insieme dei nodi che sono destinazione di qualche richiesta
- $O_d = \{v \in N : \exists k \in K,\ v = o_k \wedge d = d_k\} \subset N$: nodi che sono origine di una richiesta che ha come nodo di destinazione d
- $k_{od} = \{k \in K : o_k = 0,\ d_k = d\} \subset K$: richieste che hanno come origine il nodo o e come destinazione il nodo d

Definiamo ora le **costanti fondamentali** del problema:

- $c_{\{ij\}}$: costo di installazione di un link sullo spigolo $\{i, j\} \in E$
- $q_{\{ij\}}$: capacità di un link installato sullo spigolo $\{i, j\} \in E$
- w_{ij}: peso di instradamento per l'arco $(i, j) \in A$
- p^k: capacità della richiesta $k \in K$

Sia inoltre definita per ogni nodo $d \in D_K$ la costante M_d somma delle richieste dirette al nodo d:

$$M_d = \sum_{o \in O_d} p^{k_{od}}$$

Per ogni spigolo $\{i, j\} \in E$ definiamo poi la costante $M^{\Lambda}_{\{ij\}}$ che indichi il numero massimo di link installabili sullo spigolo $\{i, j\}$.

Consideriamo infine due costanti M^{Ω} sufficientemente grande ed ϵ sufficientemente piccola, relative alla ricerca dei cammini minimi. M^{Ω} è posta uguale alla somma degli $n - 1$ archi più lunghi della rete (dove n è il numero di nodi), mentre ϵ è posta uguale a 1. Questi valori sono rispettivamente un upper bound ed un lower bound della massima e della minima distanza tra due nodi della rete.

Passiamo ora a definire le **variabili** del problema:

- $y_{\{ij\}}, \forall \{i,j\} \in E$: variabile binaria della topologia,
  vale $\begin{cases} 1 & \text{se } \{i,j\} \text{ appartiene alla topologia} \\ 0 & \text{altrimenti} \end{cases}$
- $x_{\{ij\}},\ \forall \{i,j\} \in E$: variabile intera che indica il numero di links da installare sullo spigolo $\{i,j\}$
- $y_{ij}^d,\ \forall (i,j) \in A,\ \forall d \in D_K$: variabile binaria, vale 1 se l'arco (i,j) è su almeno uno dei cammini minimi da un qualunque nodo in O_d verso il nodo d, 0 altrimenti
- $f_{ij}^d,\ \forall (i,j) \in A,\ \forall d \in D_K$: variabile reale che indica l'ammontare totale dei flussi diretti al nodo d che passano attraverso l'arco (i,j)
- $\pi_i^d,\ \forall i \in N,\ \forall d \in D_K$: variabile reale che indica il costo del cammino minimo dal nodo i al nodo d

Consideriamo a questo punto la seguente **funzione obiettivo**:

$$\min \sum_{\{i,j\} \in E} c_{\{ij\}} x_{\{ij\}} \tag{2.1}$$

Andiamo ora a descrivere una per una le varie classi di **vincoli**:

Vincoli di capacità:

$$\sum_{d \in D_K} f_{ij}^d \leq q_{\{ij\}} x_{\{ij\}} \qquad \forall (i,j) \in A \tag{2.2}$$

Vincoli di conservazione del flusso:

$$\begin{aligned}
\sum_{j \in N/(i,j) \in A} f_{ij}^d - \sum_{j \in N/(j,i) \in A} f_{ji}^d &= p^{k_{id}} \quad \text{se } i \in O_d \\
\sum_{j \in N/(j,i) \in A} f_{ji}^d - \sum_{j \in N/(i,j) \in A} f_{ij}^d &= M_d \quad \text{se } i = d \qquad\qquad \forall d \in D_K,\ \forall i \in N \\
\sum_{j \in N/(i,j) \in A} f_{ij}^d - \sum_{j \in N/(j,i) \in A} f_{ji}^d &= 0 \qquad \text{altrimenti}
\end{aligned} \tag{2.3}$$

Vincoli di equa suddivisione dei flussi secondo la regola OSPF-ECM:

$$f_{ij}^d \leq f_{il}^d + M_d(2 - y_{ij}^d - y_{il}^d) \qquad \forall d \in D_K,\ \forall (i,j) \in A,\ \forall (i,j) \in A \tag{2.4}$$

$$f_{ij}^d \leq M_d y_{ij}^d \qquad \forall d \in D_K,\ \forall (i,j) \in A \tag{2.5}$$

Vincoli per il calcolo dei cammini minimi:

$$\pi_i^d - \pi_j^d \leq w_{ij} + \epsilon(y_{ij}^d - 1) + M^{\Omega}(1 - y_{\{ij\}}) + (\epsilon - w_{ij} - w_{ji})y_{ji}^d \qquad (2.6)$$
$$\forall (i,j) \in A,\ \forall d \in D_K$$

Vincoli relativi agli archi aperti:

$$y_{ij}^d + y_{ji}^d \leq y_{\{ij\}} \qquad \forall d \in D_K,\ \forall \{i,j\} \in E \qquad (2.7)$$

Vincoli di rafforzamento del modello (sono già presenti implicitamente all'interno dei precedenti, ma la loro aggiunta permette di escludere alcune soluzioni non intere e quindi non ammissibili nello spazio delle soluzioni):

$$x_{\{ij\}} \leq M_{\{ij\}}^{\Lambda} y_{\{ij\}} \qquad \forall \{i,j\} \in E \qquad (2.8)$$

$$y_{\{ij\}} \leq x_{\{ij\}} \qquad \forall (i,j) \in E \qquad (2.9)$$

Riepiloghiamo infine i **domini delle variabili**:

$$\begin{aligned}
y_{\{ij\}} &\in \{0,1\} && \forall \{i,j\} \in E \\
x_{\{ij\}} &\in \mathbb{N} && \forall \{i,j\} \in E \\
y_{ij}^d &\in \{0,1\} && \forall d \in D_k,\ \forall (i,j) \in A \\
f_{ij}^d &\in \mathbb{R} && \forall d \in D_k,\ \forall (i,j) \in A \\
\pi_i^d &\in \mathbb{R} && \forall d \in D_k,\ \forall i \in N
\end{aligned}$$

2.3 Implementazione del modello in Xpress Mosel

L'implementazione in linguaggio Mosel ha dovuto necessariamente tenere conto di una serie di accorgimenti che andremo a descrivere.

Anzitutto, vista la grande mole di dati relativi ad un'istanza del problema (descrizione della topologia della rete, costi di instradamento, costi dei links, richieste), è stato necessario creare un file di dati apposito per ogni istanza su cui applicare il modello. Ad esempio, un file di dati del tipo "istanza_6n_20r.dat" contiene i dati relativi ad un'istanza da 6 nodi e 20 richieste. Il file di dati viene letto dal programma in fase di inizializzazione in modo da acquisire tutte le costanti necessarie. Per un esempio effettivo di file di istanza, vedere l'Appendice B.

Per come è stato formulato il modello, esso presenta casi in cui formalmente sarebbe necessario costuire *array di insiemi*, cosa che non è possibile

effettuare in automatico con i comandi del linguaggio mosel. E' stato dunque necessario aggirare il problema costruendo per ognuno di questi array una matrice strutturata da usare in sostituzione dell'insieme che si voleva descrivere. Per vedere come si è risolto il problema nei vari casi specifici, si rimanda all'Appendice A, ove è riportato il listato del programma con i relativi commenti esplicativi. L'utilizzo di matrici opportunamente strutturate in luogo di insiemi è stato effettuato principalmente per gli insiemi O_d e k_{od}.

Analogamente, al fine di utilizzare anche gli insiemi A degli archi ed E degli spigoli, che nel linguaggio mosel non potrebbero essere costruiti come "coppie di elementi di N" quali sono, si è fatto uso della matrice di incidenza nodi-nodi (nel programma indicata con T). Essa, contenendo i dati relativi alla topologia della rete, può essere utilizzata molto vantaggiosamente per caratterizzare i suddetti insiemi. Per esempio, la scrittura formale

$$\forall\,(i,j) \in A \qquad \ldots$$

può essere inserita nel programma con la sintassi

```
forall(i in N, j in N | T(i,j) = 1) ...
```

dove $T(i,j)$ vale 1 se l'arco (i,j) è in topologia, 0 altrimenti. Quanto all'insieme degli spigoli, per trascrivere la scrittura

$$\forall\,\{i,j\} \in E \qquad \ldots$$

basta allora aggiungere

```
forall(i in N, j in N | i<j and T(i,j) = 1) ...
```

con la convenzione che nel caso ci si riferisca ad uno spigolo $\{i,j\}$ anziché ad un arco (i,j), si consideri $i < j$.

Seguendo la stessa linea di principio, al fine di ridurre il più possibile il numero di variabili effettive del modello (e dunque il costo computazionale), risulta allora opportuno definirle in modo da evitare inutili ripetizioni. In primo luogo questo si verifica quando una variabile definita su un arco non dipende dalla direzione, nel qual caso basta ovviamente memorizzare l'informazione una volta sola. Inoltre vista la possibile mancanza di collegamenti fra due nodi qualsiasi della rete, è molto conveniente memorizzare solamente quelle variabili che davvero sono necessarie per risolvere il problema, e non memorizzare proprio quelle che andrebbero poste a zero. E' per questo che tutte le variabili fondamentali del modello sono state definite attraverso *array dinamici*, che permettono la creazione solo ed esclusivamente degli indici effettivamente utilizzati. Il comando che permette di fare questo è **create**. Vediamo ad esempio il caso dell'inizializzazione delle variabili y, che servono ad indicare se uno spigolo è presente oppure no nella topologia corrente. Si procede allora in questo modo:

```
declarations
    y: dynamic array(N,N) of mpvar
    ...
end-declarations

forall(i in N, j in N | i<j and T(i,j) = 1) do
    create(y(i,j))
    ...
end-do
```

Un'ultima nota riguarda le costanti $M^{\Lambda}_{\{ij\}}$, che comparendo in un vincolo di rafforzamento, non hanno lo scopo di rappresentare una limitazione vera e propria al numero di links installabili su uno spigolo $\{i, j\}$, bensì di irrobustire il modello, e dunque sono state poste convenzionalmente tutte uguali a 100.

Capitolo 3

Metodi per la Programmazione Lineare Misto-Intera

3.1 Branch and Bound

Il metodo del Branch and Bound è stato introdotto agli inizi degli anni '60 ed è attualmente lo strumento base per la ricerca di soluzioni per un problema di Programmazione Lineare Misto-Intera (PLMI). Si tratta di un metodo di *enumerazione implicita*, caratterizzabile da uno schema ad albero che permette di tenere traccia dell'enumerazione delle soluzioni, assieme ad una procedura di valutazione del nodo (*bounding procedure*), grazie alla quale è possibile scartare quei rami che non potranno mai condurre alla soluzione ottima del problema.
L'enumerazione di tutte le possibili soluzioni ammissibili del problema risulterebbe infatti inapplicabile date le enormi dimensioni computazionali con cui si avrebbe a che fare, e dunque metodi di questo tipo possono essere estremamente efficaci in quanto riescono ad eliminare interi sottoinsiemi di soluzioni, riducendo la ricerca solo ai rami che potrebbero contenere una soluzione ottima.

Consideriamo un problema di minimo P_0 contenente sia variabili intere, sia variabili reali (PLMI). Il metodo del Branch and Bound partiziona allora l'insieme delle soluzioni ammissibili in sottoinsiemi via via più ristretti, e per ciascuno calcola un limite inferiore della funzione obiettivo. Per calcolare tale *lower bound* si usano 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. La tecnica più utilizzata è il *rilassamento continuo*: i vincoli eliminati sono quelli di interezza delle variabili. Al problema P_0 viene così associato il problema PL_0 a variabili continue, che dunque può essere risolto in un tempo molto

più breve. Nel caso la soluzione di PL_0 sia anche soluzione di P_0, cioè abbia le variabili richieste intere, allora abbiamo già determinato la soluzione, altrimenti si effettua un *branch*, cioè si sceglie una variabile x_h con valore frazionario $\bar{x}_h$ è si costruiscono i due sottoproblemi P_1 e P_2 uguali a P_0 ma con in aggiunta il vincolo:

$$x_h \leq \lfloor \bar{x}_h \rfloor \qquad \text{per } P_1$$

$$x_h \geq \lceil \bar{x}_h \rceil \qquad \text{per } P_2$$

dove $\lfloor \bar{x}_h \rfloor$ e $\lceil \bar{x}_h \rceil$ sono rispettivamente l'arrotondamento per difetto e per eccesso di $\bar{x}_h$.

La soluzione intera del problema P_0 dovrà a questo punto trovarsi necessariamente o in P_1 oppure in P_2: abbiamo ridotto il problema in due sottoproblemi leggermente più semplici.

L'operazione di *branch* può essere rappresentata visivamente come una biforcazione che a partire da un nodo padre genera due nodi figli, come in Figura 3.1.

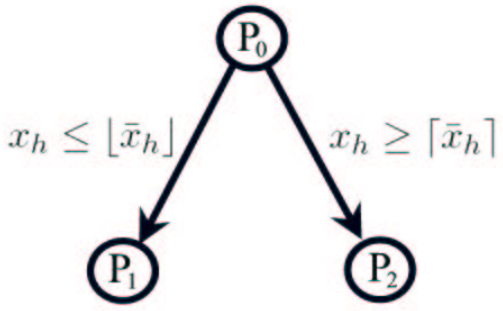

Figura 3.1: Operazione di branch

Lo stesso procedimento viene poi applicato ad ogni nodo figlio, ed in questo modo si riescono ad ottenere sottoproblemi sempre più vincolati e dunque più "facili" da risolvere, oltre che aventi con maggiore probabilità una soluzione intera.

Esistono poi dei criteri, detti *criteri di fathoming* (potatura) che permettono, in fase di costruzione dell'albero di ricerca, di stabilire se il nodo corrente debba essere chiuso oppure no. Le possibilità che si presentano dopo la risoluzione del rilassamento continuo in un generico nodo sono le seguenti:

- **inammissibilità della soluzione**: non esiste soluzione ammissibile del problema rilassato, dunque il nodo può essere chiuso

- **non interezza della soluzione**: il problema viene suddiviso in problemi figli come descritto in precedenza, dunque è un possibile candidato per soluzioni intere
- **soluzione intera**: la soluzione del problema rilassato è intera, dunque il nodo può essere chiuso perché non necessita di ulteriori ricerche; se tale soluzione è migliore della migliore soluzione intera finora ottenuta, quest'ultima viene aggiornata
- **assenza di soluzione migliorante**: se il lower bound del nodo (che costituisce peraltro una stima ottimistica del valore della migliore soluzione ottenibile a partire dal nodo corrente) è maggiore del valore della funzione obiettivo della migliore soluzione intera finora determinata, allora sicuramente sarà impossibile trovare soluzioni miglioranti fra i nodi figli del nodo corrente, e dunque anche in questo caso esso può essere chiuso

Infine è necessario definire la strategia di esplorazione dell'albero di ricerca (in profondità, con la regola *depth first*, oppure in qualità, con la regola *best first*), ed il tipo di rilassamento effettuato per il calcolo del lower bound (oltre al *rilassamento continuo* descritto, si può usare ad esempio il cosiddetto *rilassamento lagrangiano*). Per un'analisi più approfondita relativa all'applicazione di rilassamenti ad un problema di Network Design simile a quello presentato in questa tesi si veda [6].

Per ulteriori informazioni sul Branch and Bound e sugli altri metodi per la Programmazione Lineare Misto-Intera si veda [3], mentre per la presentazione generale dei metodi matematici per la Programmazione Lineare si può far riferimento a [4], oltre che allo stesso [3].

3.2 Euristiche basate su ricerca locale

Il metodo del Branch and Bound è un metodo esatto, ovvero dopo un numero finito di passi determina la soluzione del problema, ammesso che essa esista. L'inconveniente è che quando si ha a che fare con problemi PLMI, al crescere delle dimensioni del problema, i tempi computazionali crescono esponenzialmente, e questo pone in effetti un notevole limite alla sua applicazione in problemi di grandi dimensioni.

Assumendo allora di non riuscire a determinare la soluzione ottima in un tempo accettabile, è auspicabile per lo meno riuscire a trovarne una il più possibile prossima ad essa in un tempo relativamente breve, ed è in questo ambito che si inquadrano le procedure *euristiche*. Un'euristica vuole dunque essere un metodo veloce per risolvere un problema di ottimizzazione combinatiorica, senza la garanzia di raggiungere il vero e proprio ottimo. Le euristiche si suddividono in *euristiche costruttive polinomiali* ed *euristiche*

basate su ricerca locale. Le prime cercano di "costruire" in modo progressivo soluzioni "buone" facendo ad ogni passo una scelta a caso fra le migliori k possibili. Quelle basate sulla ricerca locale si propongono invece di costruire un "intorno" di una soluzione al fine di ricercare al suo interno una soluzione migliore. Ci concentriamo su questo secondo tipo di euristiche, perché è qui che si inquadra il metodo del Local Branching.

Chiamiamo X lo spazio delle soluzioni ammissibili del problema, e supponiamo di poter determinare una soluzione ammissibile qualsiasi $x_1 \in X$. Occorre allora definire un *vicinato* di x_1, ovvero un insieme $V(x_1) \subset X$ che sia sufficientemente piccolo da poter essere esplorato proficuamente. Si cerca quindi di trovare una soluzione $x_2 \in V(x_1)$ migliore di x_1, e poi si reitera il processo costruendo un secondo intorno $V(x_2)$ e cercando lì una soluzione ancora migliore, e così via...

Possiamo riassumere l'algoritmo nei seguenti passi:

1. *Inizializzazione*: si sceglie una soluzione iniziale ammissibile x_1 come soluzione corrente e si calcola il valore della sua funzione obiettivo
2. *Generazione del vicinato*: si seleziona un vicinato $V(x_i)$ della soluzione corrente x_i
3. *Miglioramento della soluzione*: si cerca nel vicinato $V(x_i)$ una soluzione x_{i+1}, possibilmente migliore della precedente
4. *Test di terminazione*: dopo un certo numero di passi predeterminato o dopo un tempo limite fissato l'algoritmo si ferma, altrimenti si ritorna al punto 2 con la soluzione x_{i+1} come soluzione corrente

La procedura fin qui descritta è estremamente generale, ed occorre poi studiare caso per caso a seconda del problema quali siano le scelte migliori da effettuare per implementarla. Anzitutto occorre definire un concetto di *vicinato*, e tanto meglio si riesce a caratterizzarlo, tanto più proficua sarà la resa dell'euristica. E' auspicabile che esso sia relativamente piccolo, in modo da poterlo magari scandire interamente per trovare la migliore soluzione al suo interno, ma neanche troppo piccolo, altrimenti aumenta troppo il rischio di rimanere intrappolati in minimi locali, magari lontani dal minimo globale della funzione obiettivo. Nella valutazione del vicinato è poi possibile scegliere essenzialmente due tipi differenti di strategie:

- *Strategia Steepest Descent*: esplora interamente il vicinato della soluzione corrente e sceglie come soluzione successiva la migliore fra tutte (esclusa la soluzione generatrice del vicinato, che per convenzione si suppone non appartenente ad esso)
- *Strategia First Improvement*: esplora il vicinato della soluzione corrente e non appena trova una soluzione migliore termina la ricerca e sceglie questa come soluzione per il passo successivo

Un modo per ovviare all'inconveniente degli intropolamenti in minimi locali può essere quello di utilizzare tecniche di *diversificazione*, cioè far sì che di tanto in tanto vengano accettate anche soluzioni *peggioranti*, che localmente faranno puntare in direzione opposta a quella dell'ottimo, ma che potrebbero consentire di aggirare ostacoli. In Figura 3.2 è mostrato l'esempio di una soluzione x_n con vicinato $V(x_n)$ troppo piccolo affinché l'algoritmo possa uscire dal minimo locale ed andare verso quello globale.

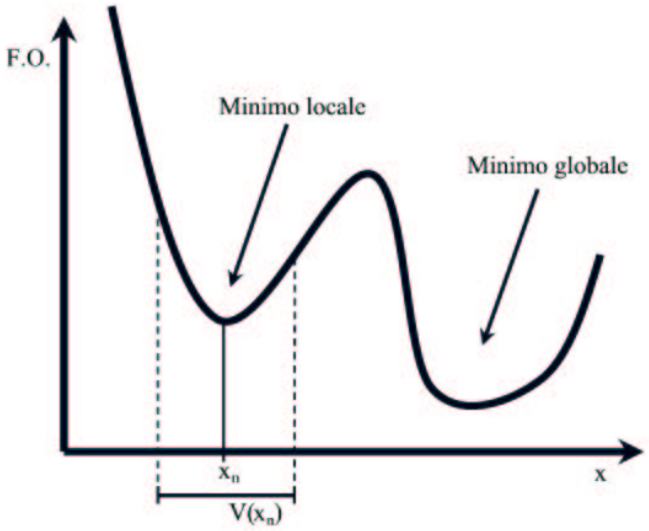

Figura 3.2: Possibile intrappolamento in minimi locali

Si utilizza il termine *meta-euristiche* basate su ricerca locale per indicare metodi che prevedano una serie di accorgimenti atti ad evitare che la ricerca torni sempre verso lo stesso insieme di soluzioni. Ad esempio, il *Tabu Search* prevede la costruzione di un *tabu list*, cioè una lista di mosse "tabu" che sia proibito effettuare (questo evita di ripetere una stessa successione di soluzioni più di una volta: se si ritorna ad una soluzione già analizzata in precedenza, la scelta successiva deve essere diversa da quella della prima volta). Un'altra possibilità è quella di diminuire progressivamente la probabilità di accettare soluzioni peggioranti fino ad annullarla verso la fine della ricerca. L'idea deriva dalla termodinamica e dalla metallurgia: quando il metallo fuso è raffreddato molto lentamnete, tende a solidificare in una struttura di minima energia. Il metodo che ne deriva va sotto il nome di *Simulated Annealing*).

3.3 Local Branching

Benché in letteratura siano state proposte tutta una serie di euristiche messe a punto appositamente per specifiche classi di problemi, vi sono solo pochi articoli riguardanti euristiche applicabili a generici problemi PLMI. Il recente lavoro presentato in [1] si propone appunto di proporre un metodo sufficientemente generale da poter essere applicato ad una vasta classe di problemi di programmazione lineare misto-intera.

Il punto saliente dei metodi euristici consta nella determinazione del vicinato di una soluzione di partenza $\bar{x}$. Tale scelta è tutt'altro che banale, ed è la chiave per ottenere un metodo efficiente. Uno schema comunemente usato è di scegliere un certo numero di variabili della soluzione di riferimento $\bar{x}$ che verranno fissate, e lasciarne libere altre. Questo modo di procedere è detto *hard variable fixing*, e presenta il difficile problema della scelta di quali variabili sia opportuno fissare e quali lasciare libere. In contrapposizione a questa tecnica è possibile pensare ad un differente modo di procedere, il cosiddetto *soft variable fixing*, che consiste invece nel far sì che un certo numero di variabili restino immutate, ma senza conoscere a priori quali siano, lasciando che sia il solver ad effettuare la scelta ottimale fra tutte quelle possibili.

Andiamo ad evidenziare meglio questo schema logico a partire dal quale deriva il metodo del *Local Branching*. Supponiamo di avere il seguente generico problema PLMI:

$$
\begin{aligned}
(P) \qquad & \min c^T x && && (3.1)\\
& Ax \geq b && && (3.2)\\
& x_j \in \{0,1\} && \forall j \in \mathcal{B} && (3.3)\\
& x_j \geq 0,\ \textit{intere} && \forall j \in \mathcal{G} && (3.4)\\
& x_j \geq 0 && \forall j \in \mathcal{C} && (3.5)
\end{aligned}
$$

L'insieme degli indici delle variabili $\mathcal{N}$ è stato suddiviso in $(\mathcal{B}, \mathcal{G}, \mathcal{C}$, dove $\mathcal{B} \neq \emptyset$ è l'insieme di indici delle variabili *binarie*, mentre gli insiemi (in teoria anche vuoti) $\mathcal{G}$ e $\mathcal{C}$ corrispondono rispettivamente alle variabili intere e continue.
Data una soluzione ammissibile di riferimento $\bar{x}$ di (P), sia

$$\bar{S} = \{j \in \mathcal{B} / \bar{x}_j = 1\}$$

il supporto binario di $\bar{x}$, ovvero l'insieme degli indici corrispondenti alle variabili binarie che per $\bar{x}$ valgono 1. Una volta stabilito un parametro intero positivo k, definiamo il *k-vicinato* $\mathcal{N}(\bar{x}, k)$ di $\bar{x}$ come l'insieme delle soluzioni ammissibili di (P) che in aggiunta soddisfano il seguente *vincolo di local branching*:

$$\Delta(x, \bar{x}) := \sum_{j \in \bar{S}} (1 - x_j) + \sum_{j \in \mathcal{B} \setminus \bar{S}} x_j \leq k \tag{3.6}$$

E' allora possibile formulare un semplice algoritmo di ricerca locale che, a partire dalla soluzione di riferimento $\bar{x}$, ne cerchi una migliore all'interno del vicinato $\mathcal{N}(\bar{x}, k)$, e poi proceda iterativamente allo stesso modo con le nuove soluzioni via via determinate. Si può anche pensare allo stesso procedimento come un metodo esatto, inquadrandolo come un tipo molto particolare di branch and bound: i tagli di local branching rappresentano i veri e propri tagli di biforcazione dell'albero di ricerca. Con riferimento alla Figura 3.3, dato un nodo $\bar{x}$ dell'albero, si creano due rami aggiungendo il vincolo:

- $\Delta(x, \bar{x}) \leq k$ per il ramo di sinistra
- $\Delta(x, \bar{x}) \geq k + 1$ per il ramo di destra

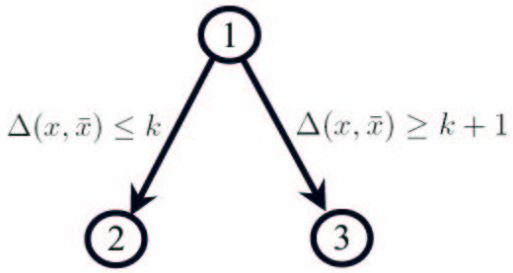

Figura 3.3: Biforcazione con taglio di Local Branching

Espresso in questo modo il local branching appare come un metodo esatto, in cui ogni ramo che va verso sinistra porta alla costruzione di un intorno supposto "sufficientemente piccolo" affinché possa essere determinata la migliore soluzione intera. Immaginando di possedere una *black box* che risolva all'ottimo un problema PLMI non troppo complesso, si può rappresentare l'intero algoritmo con un "albero a scala" che continua sempre verso il nodo di destra, come rappresentato in Figura 3.4. Per ogni ramo che va verso sinistra si risolve interamente il problema con un *Tactical Branching*, che non è nient'altro che il Solver di cui si dispone (la black box), rappresentato con un triangolino con una T dentro.

Tramite il Tactical Branching si determina l'ottimo all'interno del vicinato definito dal taglio di local branching corrispondente, e dopo aver invertito quest'ultimo vincolo di local branching seguendo il ramo di destra (in questo modo si eliminano le soluzioni ammissibili già testate) si costruisce una seconda biforcazione in base alla nuova soluzione determinata. Se

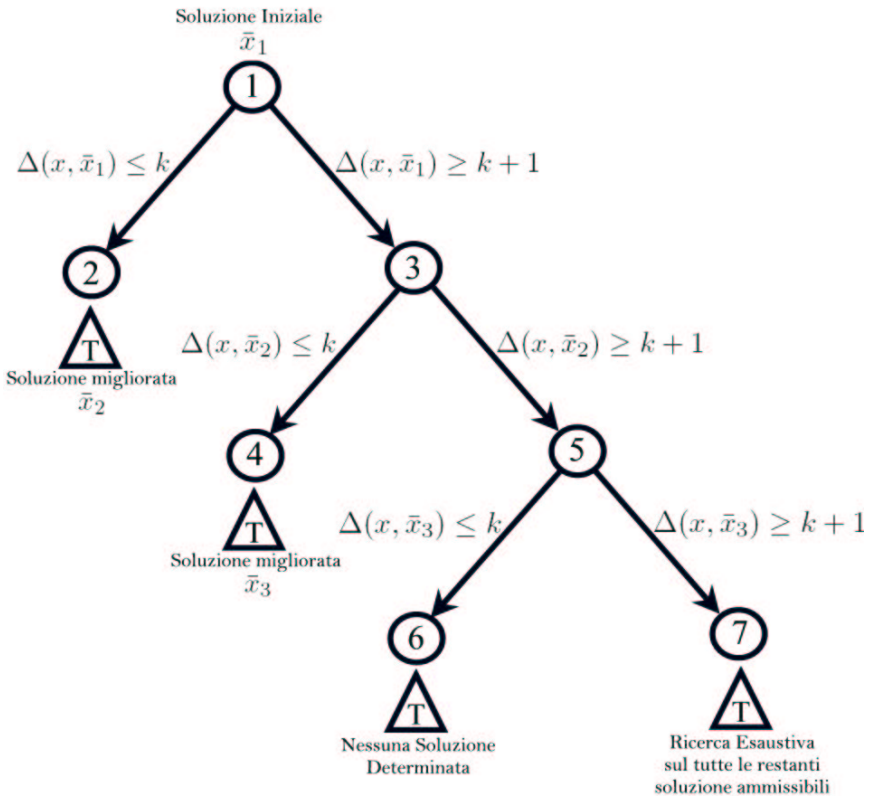

Figura 3.4: Albero di ricerca per il metodo del Local Branching

dopo l'analisi completa di un ramo di sinistra non si trova una soluzione migliorante, non resta che invertire l'ultimo vincolo di local branching e poi applicare il Tactical Branching a tutto ciò che resta a destra. Quest'ultima operazione generalmente non può essere effettuata in tempi accettabili, dato che a differenza dei rami di sinistra, che per costruzione comportano la risoluzione di problemi relativamente "piccoli', i rami di destra costituiscono un problema generalmente di dimensione troppo grande per essere risolta all'ottimo.

Conviene dunque pensare al metodo come ad un'euristica, in grado di determinare in breve tempo un alto numero di soluzioni sempre migliori ma senza garantire al termine di avere raggiunto l'ottimo.

Per migliorare ulteriormente l'efficienza del metodo, e nel contempo trasformarlo in un'euristica che non si fermi dopo un numero numero troppo limitato di passi, si possono fare le seguenti aggiunte:

Imposizione di un tempo limite sui rami di sinistra

Occorre tenere conto del fatto che talvolta neanche i nodi di sinistra sono sondabili interamente in un tempo accettabile. Si può allora pensare di dare un tempo limite alla ricerca, e se questo viene raggiunto, si possono considerare due scenari possibili:

1. Può essere stata comunque determinata una soluzione migliore della precedente, anche se non vi è la certezza che essa sia ottima. Si tratta sempre di un miglioramento, ma l'inconveniente è che non avendo controllato tutto il vicinato, non si può essere certi che al suo interno non ci possa essere l'ottimo, e dunque non è corretto dal punto di vista logico invertire il taglio di local branching precedente. Allora si va semplicemente ad aggiungere un altro ramo corrispondente ad un taglio di local branching relativo alla nuova soluzione appena determinata. Se analizzando il nodo che ne deriva si riuscirà a determinare l'ottimo, allora si potrà invertire quest'ultimo taglio e procedere con il metodo. Lo schema logico di questo scenario è rappresentato graficamente in Figura 3.5.

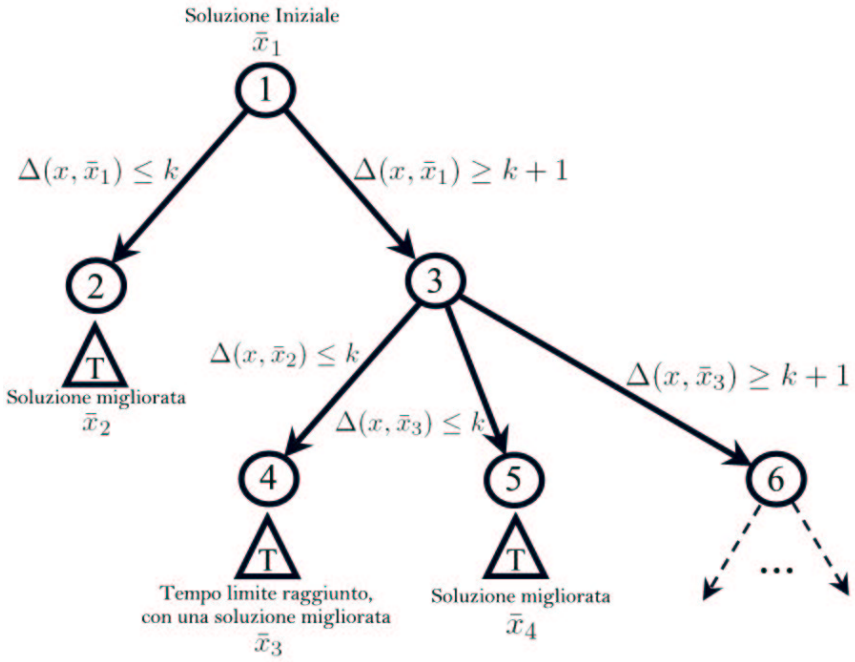

Figura 3.5: Tempo Limite raggiunto: caso (1)

2. Se invece nell'intervallo di tempo il solver non ha trovato una soluzione migliore, conviene procedere diversamente, restringendo il vicinato

corrente in modo da rendere più semplice la sua scansione completa al fine di trovare l'ottimo al suo interno. Per esempio, se il taglio era $\Delta(x, \bar{x}) \leq k$, si aggiungerà un altro ramo con il taglio $\Delta(x, \bar{x}) \leq [\frac{k}{\alpha}]$, con α costante positiva maggiore di 1. Anche in questo caso bisogna tenere conto del fatto che non si può invertire un taglio di local branching se il nodo corrispondente non è stato interamente analizzato. La rappresentazione grafica di questo secondo scenario è data dalla Figura 3.6.

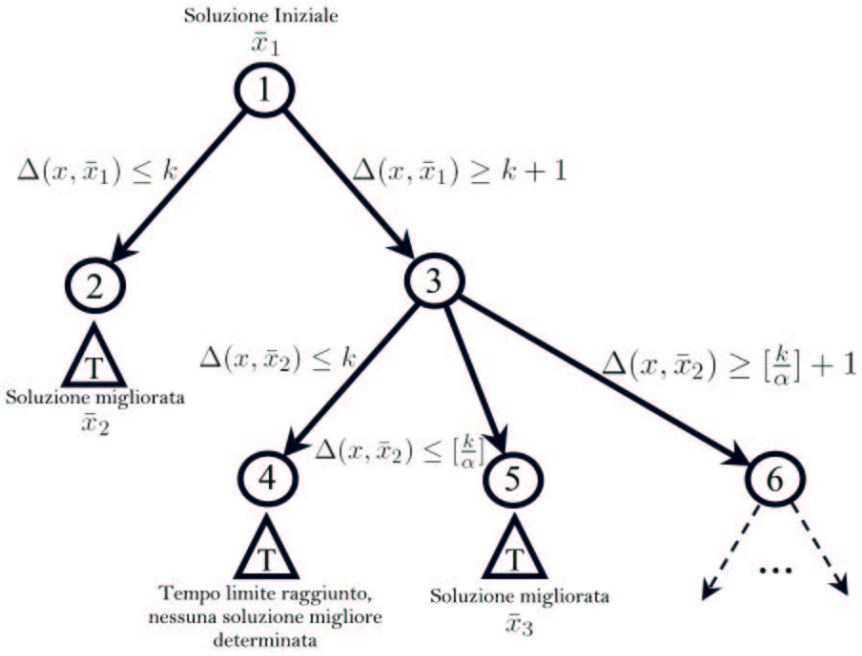

Figura 3.6: Tempo Limite raggiunto: caso (2)

Diversificazione

Qualora un intero ramo non conduca ad una soluzione migliore, al fine di non far terminare il procedimento con una pesante ricerca sul ramo corrente di destra, può convenire implementare una serie di tecniche di diversificazione ben note nell'ambito delle cosiddette meta-euristiche basate su ricerca locale. Tali metodi si rivelano particolarmente utili per uscire da possibili "intrappolamenti" in minimi locali, già descritti in precedenza. Anzitutto è possibile applicare un primo tipo di diversificazione, tentando di cercare una soluzione in un vicinato leggermente più largo del precedente. Ad esempio si può pensare di moltiplicare la dimensione k del vicinato corrente per una costante $\beta > 1$, e dunque considerare a fianco dell ramo che non ha

generato soluzioni, un nuovo ramo costituito dal taglio di local branching $\Delta(x, \bar{x}) \leq k\beta$. Se neanche in questo modo si riesce a determinare una nuova soluzione migliore, si applica un secondo tipo di diversificazione più forte, che consiste nell'allargare ulteriormente il vicinato (ma sempre cercando di non allontanarsi troppo dalla soluzione corrente) e poi, anziché cercare l'ottimo al suo interno, accontentarsi di scegliere la prima soluzione ammissibile determinata. Ad esempio, considerando sempre una costante $\beta > 1$, si può considerare il taglio $\Delta(x, \bar{x}) \leq k\beta^2$, con l'appunto di limitarsi a trovare una qualsiasi soluzione intera (cioè la prima determinata) al suo interno. Quest'ultimo tipo di diversificazione comporterà di accettare momentaneamente (e probabilmente anche per alcuni passi successivi) delle soluzioni molto peggiori di quella precedente, ma d'altro canto presenta il notevole vantaggio di permettere la potenziale uscita da minimi locali. In riferimento alla Figura 3.7, vediamo che solo allargando l'intervallo e solo accettando soluzioni che non siano le migliori possibili, è possibile "saltare" un ostacolo verso il minimo globale.

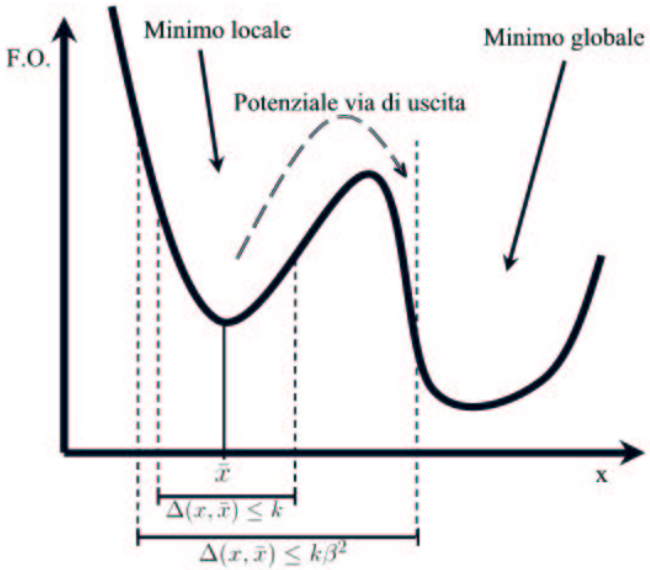

Figura 3.7: Possibile uscita da minimi locali

Tenendo conto dell'inserimento di un tempo limite e della possibilità di effettuare diversificazioni di vario tipo, si va allora a creare un algoritmo ben più complesso di quello del local branching vero e proprio. Infatti ponendo un limite alto (oppure non ponendo affatto limite) al numero delle possibili diversificazioni che si possono fare, si abbandona completamente lo schema concettuale proprio del metodo esatto che avevamo descritto inizialmente, per andare ad inquadrarsi in una procedura di ricerca meta-euristica. Non

si compirà quindi una ricerca esaustiva finalizzata alla determinazione della soluzione ottima, bensì ci si muoverà all'interno dello spazio delle soluzioni amissibili, spostandosi di soluzione intera in soluzione intera. In particolare, questi salti si rivelano efficaci al fine di determinare in breve tempo soluzioni intere molto vicine all'ottimo, che risulta effettivamente lo scopo principale quando si affronta un problema troppo complesso per essere risolto all'ottimo.

Capitolo 4

Implementazione del metodo ed analisi dei risultati

4.1 Implementazione del metodo

4.1.1 Perché il local branching

Nella sezione 3.3 è stato presentato il metodo del local branching in un quadro sufficientemente generale da far sì che esso possa essere applicato ad una vasta classe di problemi PLMI. In [1] esso viene applicato a tutta una serie di problemi di localizzazione con discreto successo, ma per problemi di Network Design il metodo non è ancora stato studiato con sufficiente profondità. Con questa tesi si è voluto appunto tentare un approccio del metodo al significativo problema di progettazione di reti di telecomunicazioni che abbiamo descritto.

E' da notare che tutti i problemi su cui è stato sperimentato dagli autori avevano la caratteristica di possedere un elevato numero di variabili binarie ed un numero di vincoli generalmente minore o confrontabile con il numero di variabili. Il problema in esame presenta caratteristiche decisamente differenti, in quanto le variabili binarie su cui andremo ad effettuare il local branching sono relativamente poche (esse crescono come n^2, dove n è il numero di nodi), mentre il numero dei vincoli rispetto ad esse è estremamente grande (cresce come n^4). Si tratta dunque di una situazione non analizzata in [1], e dunque interessantre da studiare.

Abbiamo detto che, affinché il metodo del Local Branching possa lavorare in piena efficienza, è auspicabile che nell'insieme delle variabili un ruolo fondamentale sia giocato dalle variabili binarie. Un problema di questo tipo è appunto quello che stiamo trattando, in cui le variabili binarie sono estremamente importanti perché indicano essenzialmente se un arco della topologia deve venire aperto oppure no. E' importante notare che nel problema in esame compaiono due tipi differenti di variabili binarie:

- le $y_{\{ij\}}$, che nel caso peggiore crescono come n^2, dove n è il numero di nodi
- le y^d_{ij}, che nel caso peggiore crescono come n^3

A prima vista sembrerebbe logico effettuare il local branching sulle variabili del secondo tipo, che sono in numero largamente maggiore, tuttavia occorre tenere in considerazione il fatto che le prime sono variabili principali, mentre le seconde sono strettamente correlate con le prime: fissata una configurazione delle variabili $y_{\{ij\}}$, il problema diventa "facile", in quanto la determinazione delle y^d_{ij} si trasforma in una semplice ricerca di cammini minimi, che potrebbe essere effettuata in tempi brevissimi con algoritmi ad hoc come ad esempio quello di Dijkstra. E' questa la ragione per cui, scegliendo di provare la strada del local branching per la determinazione di soluzioni il più vicine possibili a quella ottima, sono state scelte come variabili su cui effettuare il branch proprio le $y_{\{ij\}}$.

Naturalmente, essendo il metodo descritto nella sezione 3.3 un modello generale su cui basarsi, nell'implementazione dell'algoritmo sul problema in esame ci si è discostati leggermente da esso, nel modo che andremo a descrivere, che sembra il più appropriato per questo tipo di problema.

4.1.2 Soluzione iniziale

Come tutti i metodi euristici, per partire anche il Local Branching necessita di una soluzione iniziale di riferimento. Nel nostro caso è abbastanza semplice determinarne una rapidamente, in quanto assegnando una determinata topologia (cioè assengando le $y_{\{ij\}}$), il problema che rimane è come abbiamo già detto "semplice", e può essere risolto in tempi estremamente brevi.

Tuttavia resta da stabilire il criterio per la costruzione di questa soluzione iniziale, infatti anche tale scelta può incidere sull'andamento globale del metodo. Trovare il procedimento migliore riguardo alla sua costruzione senza andare per tentativi è un'impresa ardua, che andrebbe studiata molto meglio di quanto facciamo qui. In questa tesi infatti sono stati utilizzati essenzialmente due tipi differenti di soluzioni iniziali, in un certo senso agli antipodi tra di loro. Si tratta della *maglia completa* (tutti gli archi possibili vengono aperti) e dell'*albero ricoprente di costo minimo* (viene aperto il minor numero possibile di archi, aprendo quelli il cui costo $c_{\{ij\}}$ sia complessivamente minimo). La figura 4.1 rappresenta appunto le due possibili soluzioni iniziali per l'istanza di 6 nodi.

4.1.3 Tagli di local branching

Come abbiamo detto, per il nostro problema è logico effettuare i tagli sulle variabili decisionali $y_{\{ij\}}$, e dunque nell'algoritmo che è stato sviluppato si è

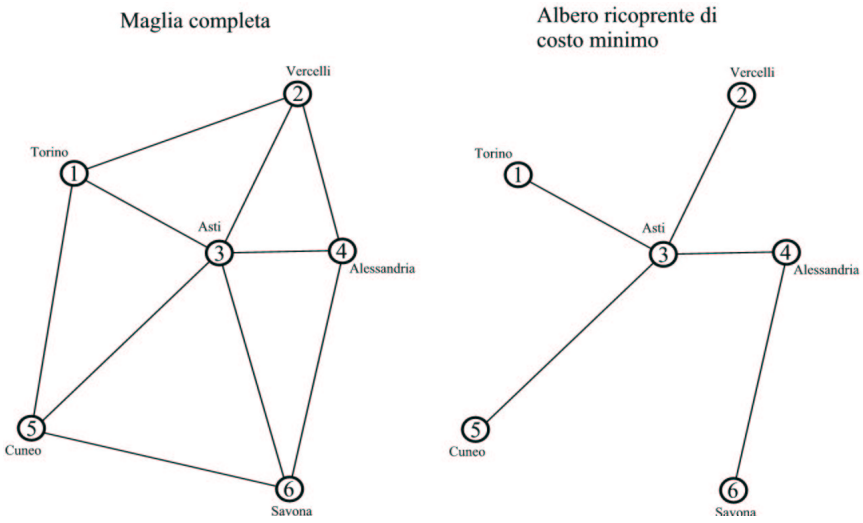

Figura 4.1: Soluzioni iniziali per l'istanza a 6 nodi

provveduto a fare proprio questo. Ad ogni passo è stato inserito un vincolo di local branching del tipo 3.6, che relativamente a queste variabili diventa:

$$\Delta(y, \bar{y}) := \sum_{\{i,j\} \in \bar{S}} \Big(1 - y_{\{ij\}}\Big) + \sum_{\{i,j\} \in \mathcal{B} \setminus \bar{S}} y_{\{ij\}} \leq k \tag{4.1}$$

dove $\bar{y}$ è la soluzione di riferimento e, se $\mathcal{B}$ è l'insieme delle variabili binarie $y_{\{ij\}}$,

$$\bar{S} = \Big\{ \{i,j\} \in \mathcal{B} / \bar{x}_{\{ij\}} = 1 \Big\}$$

Il parametro k sta quindi ad indicare quanti sono gli archi che possono cambiare stato (passare da chiusi ad aperti o viceversa) dalla soluzione $\bar{y}$ alla nuova soluzione y.

A seconda delle dimensioni delle istanze studiate, sarà poi interessante vedere quale scelta del parametro k condurrà alla maggiore efficienza. Per questo sono stati studiati sia valori di k piccoli sia valori più elevati, relativamente alle dimensioni delle istanze considerate.

4.1.4 Tempo limite

Dato che già per problemi di dimensioni moderate la ricerca integrale nel nodo di sinistra risulta estremamente dispendiosa, è stata introdotta la possibilità di utilizzare un tempo limite. Xpress permette di settare un tempo oltre il quale fermarsi nel caso sia già stata trovata almeno una soluzione intera, oppure continuare fino alla determinazione della prima. Si tratta di

due differenti strategie, che permettono l'una di avanzare molto più velocemente, ma con decrementi della funzione obiettivo relativamente piccoli (*first improvement*), l'altra di fare "salti" più rilevanti nella discesa verso l'ottimo, ma impiegando più tempo (*steepest descent*). Dato che non sembra esserci una scelta ottimale, si è ritenuto interessante valutare entrambe le possibilità, imponendo in un caso il tempo limite uguale (convenzionalmente) ad 1 secondo, e facendo dunque sì che il solver si fermasse sempre alla prima soluzione intera determinata, nell'altro caso imponendo un tempo limite confrontabile con i tempi del problema (che variano a seconda dell'istanza), facendo sì che almeno nei primi passi il solver ricercasse una soluzione più vicina a quella ottima.

Naturalmente, come già descritto in linea teorica nella sezione 3.3, qualora venga raggiunto il tempo limite, e non venga dunque dimostrata l'ottimalità della soluzione determinata, non si può invertire l'ultimo taglio di local branching, e dunque occorre eliminare il taglio vecchio per poi aggiungerne uno nuovo relativo alla soluzione aggiornata che è appena stata determinata.

4.1.5 Imposizione di un Upper Bound

Dato che ogni taglio di local branching impone al solver la risoluzione di un problema per lui completamente nuovo, conviene impostare un upper bound sulla soluzione migliore finora determinata. Infatti ripartendo ogni volta dall'inizio il solver non ha l'informazione sulla soluzione determinata nell'ottimizzazione precedente, che è proprio la soluzione in base alla quale è stato creato il vicinato con il taglio di local branching corrente. Allora può convenire imporre un altro vincolo che possiamo chiamare "vincolo di Upper Bound", definito semplicemente come:

$$\text{Funzione Obiettivo } \leq \widetilde{UB} - 1 \tag{4.2}$$

dove $\widetilde{UB}$ è il valore della funzione obiettivo migliore finora determinata. E' stato inserito inoltre il -1 in quanto nel problema la Funzione Obiettivo è sempre un numero intero, e dunque, nel cercare all'interno del vicinato corrente una nuova soluzione migliore, essa dovrà avere un valore necessariamente minore di $\widetilde{UB}$, ed essendo esso intero, dovrà allora essere minore o uguale di $\widetilde{UB} - 1$. Tra l'altro questo vincolo aggiuntivo fa sì che in nessun caso possa venire accettata di nuovo una soluzione già determinata in precedenza, in particolare impone che la soluzione in base alla quale è stato costruito il vicinato non faccia parte del vicinato stesso.

4.1.6 Struttura dell'algoritmo

L'algoritmo che deriva dalle considerazioni fin qui fatte, e che è poi effettivamente quello che è stato scritto in Mosel IVE e sulla base del quale

sono stati determinati i risultati che andremo a riportare nella sezione 4.3, è dunque un tipo di local branching standard, con l'aggiunta di un tempo limite nella ricerca. Considerando il tipo di taglio effettuato nel contesto del problema (che non ha un numero particolarmente elevato di variabili binarie su cui fare il local branching), con valori di k non troppo bassi è abbastanza facile che il metodo conduca all'ottimo anche senza l'utilizzo di tecniche di diversificazione. Inoltre vale la pena ricordare che per istanze reali (e dunque grandi), il metodo ha essenzialmente lo scopo di funzionare come un'euristica, in grado cioè di trovare in tempi brevi soluzioni vicine all'ottimo, e dunque almeno in questa versione preliminare, in cui si vuole ottenere un primo confronto del metodo con il Branch and Bound tradizionale, si è ritenuto sufficiente procedere nel senso fin qui descritto, senza l'ulteriore aggiunta di tecniche di diversificazione.

Occorre ancora spendere alcune parole relative alla terminazione del metodo. Dato che non sono state introdotte diversificazioni e che la Funzione Obiettivo deve necessariamente scendere dopo ogni taglio a causa dell'imposizione dell'ulteriore vincolo di Upper Bound, in un numero finito di passi si arriva a esplorare un vicinato che non contiene più soluzioni migliori. In questo caso il problema viene dichiarato "Unfeasible" e quindi il metodo si ferma. Non è però garantito che sia stato trovato l'ottimo, in quanto occorre ancora invertire l'ultimo taglio di local branching ed effettuare una ricerca esaustiva su tutto il restante spazio delle soluzioni. Nell'algoritmo è stato implementato anche quest'ultimo passo, anche se nelle istanze di dimensioni significative non è mai stato portato a termine vista l'elevata complessità computazionale. Del resto, come abbiamo già detto, l'unico interesse è di raggiungere in tempi brevi una soluzione vicina all'ottimo, e dunque possiamo benissimo tralasciare quest'ultimo passo che sarebbe troppo oneroso. Nella sezione 4.3, dove elencheremo i risultati veri e propri, ometteremo dunque volutamente tutto quello che accade dopo che il metodo del local branching raggiunge l'ultimo nodo di sinistra, e ci fermeremo anzi al passo precedente, cioè all'ultima soluzione intera determinata dal metodo.

4.1.7 Xpress IVE

Il software Xpress della Dash Optimization [7], con il relativo linguaggio di programmazione Mosel, è particolarmente utile per quanto riguarda la Programmazione Lineare Misto-Intera, dato che con la sua flessibilità permette all'utente di implementare algoritmi anche molto sofisticati per la ricerca delle soluzioni. E' per questo che il listato che è stato scritto per implementare il metodo sul problema in esame, presente in Appendice A, è costituito da due parti che possono essere pensate praticamente separate. In primo luogo è stato descritto il problema nella sua formulazione in PLMI (definizione variabili, vincoli, funzione obiettivo,...), e poi, anziché lasciare che il solver lo risolva con il metodo del Branch and Bound, è stato scritto

un algoritmo che prima trovi una soluzione iniziale, e poi effettui i passi del Local Branching.

Questa seconda parte, oltre ad usare le funzioni basilari del linguaggio Mosel per creare cicli e condizioni, ha richiesto anche l'utilizzo di funzioni specifiche che consentono di settare i parametri dell'ottimizzatore. Fra questi sono fondamentali i seguenti:

- `minimize(...)` è il comando che "lancia" il solver con il problema che è stato finora impostato al fine di minimizzare la Funzione Obiettivo che viene indicata all'interno delle parentesi;
- `getobjval` restituisce il valore della Funzione Obiettivo.
- `getsol(...)` permette di leggere i valori delle variabili correnti, dopo che il solver ha terminato di lavorare;
- `sethidden(vinc, true)` consente di "nascondere" un vincolo che non è più valido;
- `setparam("XPRS_MAXTIME",TL)` imposta il tempo limite per la successiva ottimizzazione a TL;
- `getprobstat` restituisce lo stato di ottimizzazione (Optimality, Infeasibility,
- `gettime` fornisce un valore sempre crescente corrispondente al tempo in secondi del clock;

Per una descrizione accurata di questi e degli altri comandi che sono stati utilizzati, si rimanda alla documentazione di Xpress [7].

4.2 Istanze analizzate

Dallo studio del modello in esame si capisce che la complessità di un'istanza dipende in primo luogo dal numero di nodi presenti, ed in secondo luogo non dal numero di richieste, ma dal numero di *nodi di destinazione*. Infatti, sia nelle dichiarazioni delle variabili, sia nei vincoli, compare sempre l'insieme D_K dei nodi che sono destinazione di almeno una richiesta, e mai l'insieme K delle richieste vere e proprie. L'insieme K delle richieste (in particolare attraverso i suoi sottoinsiemi O_d) può modificare solo la struttura di alcuni vincoli (quelli di conservazione del flusso), ma non il loro numero. Nella pratica sarà dunque quasi indifferente far variare il numero delle richieste se i nodi di destinazione rimangono sempre gli stessi: il numero di variabili e di vincoli del problema resterebbe identico.

Fatta questa premessa, diaciamo che un'istanza verosimile del problema ha come insieme D_K tutto o quasi tutto l'insieme dei nodi N. Chiameremo

completa un'istanza che abbia come richieste tutte quelle possibili fra ogni coppia di nodi. In particolare, nello studiare problemi di complessità intermedia, ovvero con un numero di destinazioni minore del numero di nodi, faremo sì che relativamente a quelle destinazioni l'istanza sia completa, cioè che ad ogni nodo di destinazione arrivino flussi da tutti gli altri nodi della rete.

Una serie di tentativi iniziali hanno mostrato che il problema con istanze complete da 3, 4 e 5 nodi è decisamente semplice (un'istanza completa da 5 nodi contiene 20 richieste), e viene risolto dal solver stesso in frazioni di secondo. Il solver avverte invece pesantemente il salto computazionale fra delle istanze complete da 5 e da 6 nodi (un'istanza completa da 6 nodi comprende invece 30 richieste): addirittura per il secondo caso esso non riesce a risolvere all'ottimo il problema in 24 ore.

E' stato allora naturale andare a studiare il comportamento sia del solver sia dell'algoritmo con il metodo del Local Branching per istanze non complete da 6 e da 7 nodi. In particolare, l'istanza da 6 nodi studiata è quella rappresentata in figura 2.1.

4.3 Risultati ottenuti

Visto l'elevato numero di parametri in gioco (il numero di destinazioni, il tipo di istanza iniziale, il valore di k, il tempo limite), anche la semplice stesura dei risultati richiede qualche accorgimento. Inoltre, per capire laddove sia davvero utile il metodo, è necessario confrontare i risultati con quelli che si otterrebbero senza alcun algoritmo particolare, ma lasciando che il Solver ricerchi l'ottimo utilizzando l'algoritmo del Branch and Bound incorporato.

Una prima tabella raccoglierà dunque i risultati relativi alla ricerca di base, per il problema a 6 nodi e poi a 7 nodi, al variare del numero di destinazioni. I risultati relativi al metodo andranno invece a seguire, suddivisi anche a seconda del tipo di soluzione iniziale utilizzata, e per ogni classe di destinazioni, saranno presenti i valori di $k = 1,\ 3,\ 5$ (corrispondenti a dimensioni piccole, medie e grandi dei vicinati) ed i due casi di tempo limite già descritti.

La notazione che utilizzeremo per indicare il modo in cui la soluzione migliora è di indicare i valori della funzione obiettivo che si aggiorna col passare del tempo. Ad esempio la scrittura:

$$311 \xrightarrow{1.2} 297 \xrightarrow{27.4} 289 \xrightarrow{6.1} \underline{288} \cdots$$

indica che partendo dalla soluzione iniziale 311, dopo 1.2 s il primo taglio di local branching conduce alla soluzione 297, poi dopo altri 27.4 s la risoluzione con il secondo taglio conduce a 289, e dopo 6.1 s si arriva a 288. Per mostrare che si è arrivati all'ottimo, qualora questo sia noto, l'ultimo valore è stato sottolineato (infatti esistono casi in cui non sempre si giunge all'ottimo).

Branch and Bound del Solver

Utilizzeremo qui la stessa notazione appena descritta, anche se naturalmente non si tratta più di tagli di local branching, ma semplicemente di "passi" che il Solver esegue al suo interno. Dato il nostro interesse alle soluzioni intere, è comunque utile rappresentare quelle che vengono via via trovate allo stesso modo.

Ove possibile, si è arrivati anche a provare l'ottimalità della soluzione, quando essa può essere raggiunta in tempi non troppo elevati. In tal caso, dopo aver determinato l'ottimo, il solver deve completare la procedura di analisi in un certo lasso di tempo, oltre il quale può garantire che non esistano altre soluzioni migliori.

6N-1D	$\infty \xrightarrow{0.10} 120 \xrightarrow{0.05} 119 \xrightarrow{0.05} \underline{115} \xrightarrow{0.8}$ Ottimalità Provata
6N-2D	$\infty \xrightarrow{0.6} 229 \xrightarrow{5.6} \underline{203} \xrightarrow{2.5}$ Ottimalità Provata
6N-3D	$\infty \xrightarrow{28} 327 \xrightarrow{0.5} 304 \xrightarrow{1.5} 288 \xrightarrow{14} 257 \xrightarrow{38} \underline{256} \xrightarrow{23}$ Ottimalità Provata
6N-4D	$\infty \xrightarrow{98} 431 \xrightarrow{30} 406 \xrightarrow{26} 372 \xrightarrow{16} 335 \xrightarrow{21} 318 \xrightarrow{1847} 313 \xrightarrow{4887} 311 \xrightarrow{248} \underline{306} \xrightarrow{5054}$ Ottimalità Provata
6N-5D	
6N-6D	$\infty \xrightarrow{12211} 406 \xrightarrow{9487} 382 \overset{>}{\longrightarrow}{}^{40000}?$
7N-1D	$\infty \xrightarrow{0.10} \underline{103} \xrightarrow{0.8}$ Ottimalità Provata
7N-2D	$\infty \xrightarrow{1} \underline{212} \xrightarrow{20.5}$ Ottimalità Provata
7N-3D	$\infty \xrightarrow{371} 615 \xrightarrow{347} 364 \xrightarrow{25} 356 \xrightarrow{93} 345 \xrightarrow{1297} 344 \xrightarrow{1501} 343 \xrightarrow{50} 339 \xrightarrow{1895} \underline{338} \xrightarrow{2246}$ Ottimalità Provata
7N-4D	$\infty \xrightarrow{102} 525 \xrightarrow{1557} 522 \xrightarrow{250} 508 \xrightarrow{269} 477 \xrightarrow{8424} 461 \overset{>}{\longrightarrow}{}^{32000}?$
7N-5D	
7N-6D	
7N-7D	$\infty \xrightarrow{8551} 725 \xrightarrow{2135} 669 \overset{>}{\longrightarrow}{}^{100000}?$

Local Branching su grafo a 6 nodi con soluzione iniziale maglia completa

1D	k=1	TL=1	$145 \xrightarrow{1.1} 137 \xrightarrow{1.1} 130 \xrightarrow{0.7} 125 \xrightarrow{0.1} 120 \xrightarrow{0.1} \underline{115}$
		TL=5	$145 \xrightarrow{2.9} 137 \xrightarrow{1.4} 130 \xrightarrow{0.7} 125 \xrightarrow{0.1} 120 \xrightarrow{0.1} \underline{115}$
	k=3	TL=1	$145 \xrightarrow{1.1} 125 \xrightarrow{0.1} \underline{115}$
		TL=5	$145 \xrightarrow{1.8} 125 \xrightarrow{0.1} \underline{115}$
	k=5	TL=1	$145 \xrightarrow{0.1} \underline{115}$
		TL=5	$145 \xrightarrow{0.1} \underline{115}$
2D	k=1	TL=1	$236 \xrightarrow{304} 235 \xrightarrow{5.7} 225 \xrightarrow{1.7} 210 \xrightarrow{1.1} 208 \xrightarrow{1.2} \underline{203}$
		TL=10	$236 \xrightarrow{310} 235 \xrightarrow{11.2} 225 \xrightarrow{10.9} 210 \xrightarrow{11.0} 208 \xrightarrow{2.4} \underline{203}$
	k=3	TL=1	$236 \xrightarrow{16.4} 215 \xrightarrow{2.5} 208 \xrightarrow{5.6} \underline{203}$
		TL=10	$236 \xrightarrow{16.9} 215 \xrightarrow{7.5} \underline{203}$
	k=5	TL=1	$236 \xrightarrow{1.1} 217 \xrightarrow{2.8} 214 \xrightarrow{5.3} \underline{203}$
		TL=10	$236 \xrightarrow{8.1} \underline{203}$
3D	k=1	TL=1	$287 \xrightarrow{64.9} 285 \xrightarrow{99} 282 \xrightarrow{1.2} 278 \xrightarrow{12.8} 274 \xrightarrow{7.3} 256 \xrightarrow{1.8} \underline{256}$
		TL=40	$287 \xrightarrow{66} 285 \xrightarrow{99} 282 \xrightarrow{18.1} 257 \xrightarrow{1.3} \underline{256}$
	k=3	TL=1	$287 \xrightarrow{8.2} 260 \xrightarrow{20.4} 257 \xrightarrow{6.3} \underline{256}$
		TL=40	$287 \xrightarrow{42.2} 257 \xrightarrow{63} \underline{256}$
	k=5	TL=1	$287 \xrightarrow{17.1} 257 \xrightarrow{24.7} \underline{256}$
		TL=40	$287 \xrightarrow{42.1} 257 \xrightarrow{34.8} \underline{256}$
...			...

Local Branching su grafo a 6 nodi con soluzione iniziale albero ricoprente di costo minimo

1D	k=1	TL=1	$136 \xrightarrow{0.1} 131 \xrightarrow{0.3} 126 \xrightarrow{0.1} 124 \xrightarrow{0.2} 119$
		TL=5	$136 \xrightarrow{0.1} 131 \xrightarrow{0.3} 126 \xrightarrow{0.1} 124 \xrightarrow{0.2} 119$
	k=3	TL=1	$136 \xrightarrow{0.4} 124 \xrightarrow{0.1} \underline{115}$
		TL=5	$136 \xrightarrow{0.4} 124 \xrightarrow{0.1} \underline{115}$
	k=5	TL=1	$136 \xrightarrow{0.4} 119 \xrightarrow{0.1} \underline{115}$
		TL=5	$136 \xrightarrow{0.4} 119 \xrightarrow{0.1} \underline{115}$
2D	k=1	TL=1	$255 \xrightarrow{0.5} 237 \xrightarrow{1.1} 235 \xrightarrow{4.4} 228$
		TL=10	$255 \xrightarrow{0.5} 237 \xrightarrow{1.4} 235 \xrightarrow{6.7} 228$
	k=3	TL=1	$255 \xrightarrow{6.6} 228 \xrightarrow{2.5} 214 \xrightarrow{2.3} \underline{203}$
		TL=10	$255 \xrightarrow{10.5} 228 \xrightarrow{10.7} 214 \xrightarrow{5.9} \underline{203}$
	k=5	TL=1	$255 \xrightarrow{4.7} 217 \xrightarrow{1.9} 208 \xrightarrow{1.6} \underline{203}$
		TL=10	$255 \xrightarrow{10.6} 217 \xrightarrow{7.8} \underline{203}$
3D	k=1	TL=1	$305 \xrightarrow{0.5} 287 \xrightarrow{1.4} 286 \xrightarrow{4.2} 278$
		TL=40	$305 \xrightarrow{0.5} 287 \xrightarrow{4.7} 286 \xrightarrow{15.6} 278$
	k=3	TL=1	$305 \xrightarrow{9.8} 296 \xrightarrow{11.4} 260 \xrightarrow{19.9} 257 \xrightarrow{6.1} \underline{256}$
		TL=40	$305 \xrightarrow{42} 291 \xrightarrow{41.9} 257 \xrightarrow{45.7} \underline{256}$
	k=5	TL=1	$305 \xrightarrow{138} 285 \xrightarrow{1.2} 257 \xrightarrow{1.1} \underline{256}$
		TL=40	$305 \xrightarrow{138} 285 \xrightarrow{29} \underline{256}$
4D	k=1	TL=1	$354 \xrightarrow{1.1} 336 \xrightarrow{3.6} 335$
		TL=40	$354 \xrightarrow{1.1} 336 \xrightarrow{14.6} 335$
	k=3	TL=1	$354 \xrightarrow{11.2} 343 \xrightarrow{11.4} 317 \xrightarrow{1447} 312 \xrightarrow{638} \underline{306}$
		TL=40	$354 \xrightarrow{30.6} 332 \xrightarrow{508} 317 \xrightarrow{1447} 312 \xrightarrow{638} \underline{306}$
	k=5	TL=1	$354 \xrightarrow{160} 340 \xrightarrow{265} 333 \xrightarrow{17.3} 325 \xrightarrow{881} 317 \xrightarrow{751} 313 \xrightarrow{1450} 307 \xrightarrow{158} \underline{306}$
		TL=40	$354 \xrightarrow{160} 340 \xrightarrow{265} 333 \xrightarrow{17.3} 325 \xrightarrow{881} 317 \xrightarrow{751} 313 \xrightarrow{1450} 307 \xrightarrow{158} \underline{306}$
...			...
6D	k=3	TL=300	$390 \xrightarrow{310} 380 \xrightarrow{25015} 364 \xrightarrow{> 60000} ?$

Local Branching su grafo a 7 nodi con soluzione iniziale maglia completa

1D	k=1	TL=1	$142 \xrightarrow{1.2} 134 \xrightarrow{1.2} 123 \xrightarrow{0.7} 116 \xrightarrow{0.4} 109 \xrightarrow{0.2} 104$
		TL=5	$142 \xrightarrow{5.6} 134 \xrightarrow{2.2} 123 \xrightarrow{0.7} 116 \xrightarrow{0.4} 109 \xrightarrow{0.2} 104$
	k=3	TL=1	$142 \xrightarrow{1.2} 119 \xrightarrow{0.2} \underline{103}$
		TL=5	$142 \xrightarrow{2.5} 116 \xrightarrow{0.2} 104$
	k=5	TL=1	$142 \xrightarrow{0.4} 104 \xrightarrow{0.1} \underline{103}$
		TL=5	$142 \xrightarrow{0.4} 104 \xrightarrow{0.1} \underline{103}$
2D	k=1	TL=1	$241 \xrightarrow{1.2} 233 \xrightarrow{21.7} 231 \xrightarrow{3.2} 220 \xrightarrow{3.1} 216$
		TL=10	$241 \xrightarrow{10.5} 233 \xrightarrow{21.7} 231 \xrightarrow{10.6} 220 \xrightarrow{10.5} 216$
	k=3	TL=1	$241 \xrightarrow{6.8} 221 \xrightarrow{24.8} 218 \xrightarrow{3.0} 217 \xrightarrow{11.2} 213 \xrightarrow{13.1} \underline{212}$
		TL=10	$241 \xrightarrow{10.7} 221 \xrightarrow{24.8} 218 \xrightarrow{10.8} 217 \xrightarrow{11.2} 213 \xrightarrow{13.1} \underline{212}$
	k=5	TL=1	$241 \xrightarrow{1.2} 217 \xrightarrow{11.5} 213 \xrightarrow{14.4} \underline{212}$
		TL=10	$241 \xrightarrow{10.7} 216 \xrightarrow{11.2} 213 \xrightarrow{15} \underline{212}$
3D	k=1	TL=1	$374 \xrightarrow{191} 371 \xrightarrow{214} 353 \xrightarrow{158} 343 \xrightarrow{957} \underline{338}$
		TL=40	$374 \xrightarrow{191} 371 \xrightarrow{214} 353 \xrightarrow{158} 343 \xrightarrow{957} \underline{338}$
	k=3	TL=1	$374 \xrightarrow{17.8} 366 \xrightarrow{99} 362 \xrightarrow{1093} 361 \xrightarrow{801} 360 \xrightarrow{246} 359 \xrightarrow{48.7} 346 \xrightarrow{1234} 343 \xrightarrow{4515} 339 \xrightarrow{9316} \underline{338}$
		TL=40	$374 \xrightarrow{42.1} 366 \xrightarrow{99} 362 \xrightarrow{1093} 361 \xrightarrow{801} 360 \xrightarrow{246} 359 \xrightarrow{48.7} 346 \xrightarrow{1234} 343 \xrightarrow{4515} 339 \xrightarrow{9316} \underline{338}$
	k=5	TL=1	$374 \xrightarrow{112} 352 \xrightarrow{843} 349 \xrightarrow{6172} 346 \xrightarrow{352} 344 \xrightarrow{13564} 339 \xrightarrow{5215} \underline{338}$
		TL=40	$374 \xrightarrow{112} 352 \xrightarrow{843} 349 \xrightarrow{6172} 346 \xrightarrow{352} 344 \xrightarrow{13564} 339 \xrightarrow{5215} \underline{338}$
...			...

Local Branching su grafo a 7 nodi con soluzione iniziale albero ricoprente di costo minimo

1D	k=1	TL=1	$\underline{103}$
		TL=5	$\underline{103}$
	k=3	TL=1	$\underline{103}$
		TL=5	$\underline{103}$
	k=5	TL=1	$\underline{103}$
		TL=5	$\underline{103}$
2D	k=1	TL=1	$252 \xrightarrow{0.5} 240 \xrightarrow{1.0} 231 \xrightarrow{2.5} 227 \xrightarrow{1.2} 223 \xrightarrow{1.1} 217 \xrightarrow{1.0} 216$
		TL=10	$252 \xrightarrow{0.5} 240 \xrightarrow{1.3} 231 \xrightarrow{4.9} 225 \xrightarrow{8.2} 221$
	k=3	TL=1	$252 \xrightarrow{1.6} 234 \xrightarrow{33.6} 227 \xrightarrow{1.1} 220 \xrightarrow{2.9} 213 \xrightarrow{7.5} \underline{212}$
		TL=10	$252 \xrightarrow{10.8} 234 \xrightarrow{32.9} 227 \xrightarrow{10.7} \underline{212}$
	k=5	TL=1	$252 \xrightarrow{5.2} 236 \xrightarrow{4.7} \underline{212}$
		TL=10	$252 \xrightarrow{10.8} 217 \xrightarrow{10.8} 213 \xrightarrow{14.1} \underline{212}$
3D	k=1	TL=1	$386 \xrightarrow{28.2} 353 \xrightarrow{819} 348 \xrightarrow{1557} 344 \xrightarrow{522} 341 \xrightarrow{582} \underline{338}$
		TL=40	$386 \xrightarrow{41.8} 353 \xrightarrow{819} 348 \xrightarrow{1557} 344 \xrightarrow{522} 341 \xrightarrow{582} \underline{338}$
	k=3	TL=1	$386 \xrightarrow{26.3} 353 \xrightarrow{782} 348 \xrightarrow{1577} 344 \xrightarrow{507} 341 \xrightarrow{572} \underline{338}$
		TL=40	$386 \xrightarrow{41.5} 353 \xrightarrow{790} 348 \xrightarrow{1553} 344 \xrightarrow{506} 341 \xrightarrow{569} \underline{338}$
	k=5	TL=1	$386 \xrightarrow{209} 384 \xrightarrow{338} 348 \xrightarrow{2197} \underline{338}$
		TL=40	$386 \xrightarrow{209} 384 \xrightarrow{338} 348 \xrightarrow{2197} \underline{338}$
...			...
7D	k=3	TL=500	$724 \xrightarrow{512} 594 \xrightarrow{> 73000} ?$

4.4 Analisi dei risultati

In linea generale, per istanze di dimensione relativamente elevata, il metodo sembra essere più efficiente del Branch and Bound tradizionale, ma questo accade solo con una accurata scelta dei parametri e della soluzione iniziale, e non sempre in linea con le previsioni teoriche.

Anzitutto un primo discorso va fatto riguardo alla scelta delle soluzioni iniziali a partire dalle quali far iniziare il metodo. La soluzione a maglia completa sembra troppo sbilanciata: con questa scelta sicuramente il primo passo di local branching consta solamente nel *chiudere* degli archi aperti, e spesso la cosa vale anche per il secondo ed il terzo taglio. Questo significa che all'inizio occorre solo far passare da 1 a 0 alcune delle variabili $y_{\{ij\}}$, e

dunque, a differenza di quanto dovrebbe essere vero in linea di principio, più viene allargato il vicinato più il metodo migliora. Per $k = 1$ i primi passi sono decisamente lenti rispetto agli altri valori di k, in particolare nel caso 6 nodi e 2 destinazioni, che viene risolto dal Solver (Branch and Bound puro) in frazioni di secondo.

L'altra soluzione iniziale, costituita dall'albero ricoprente di costo minimo, risulta effettivamente molto più bilanciata e, salvo qualche eccezione, funziona meglio della maglia completa. E' da notare però che nella quasi totalità delle istanze di dimensioni rilevanti, essa parte da un valore della Funzione Obiettivo leggermente più alto e, dunque, risulta svantaggiata per quanto riguarda la partenza vera e propria, anche se poi effettua molto più rapidamente i primi passi. Questo secondo tipo di soluzione iniziale presenta esattamente le caratteristiche che ci si aspettano dalla ricerca locale: più viene ridotta la dimensione del vicinato k più il metodo è veloce.

Naturalmente la scelta della soluzione iniziale influenza solamente i primissimi passi del metodo, ma dato che già al loro termine si ottengono risultati abbastanza vicini all'ottimo, è importante effettuare la scelta migliore anche in questo senso.

Per quanto riguarda il raggiungimento vero e proprio del valore ottimo della Funzione Obiettivo, il metodo funziona bene per i valori $k = 3$ e $k = 5$, mentre spesso con $k = 1$ l'ultima soluzione determinata non è la migliore possibile. Un unico caso anomalo che con $k = 3$ non determina l'ottimo si ha nell'istanza a 7 nodi e 1 destinazione, con soluzione iniziale maglia completa: se il tempo limite viene posto uguale a 1 il metodo raggiunge l'ottimo 103, se viene posto uguale a 5 il metodo si ferma a 104. Si tratta comunque di un'istanza di dimensione molto ridotta (1 sola destinazione), che viene risolta dal Branch and Bound del solver in una frazione di secondo, e dunque non particolarmente significativa. La nostra scelta di non utilizzare tecniche di diversificazione sembra comunque adattarsi bene al tipo di problema che stiamo affrontando, pur tenendo ben presente che con vicinati molto ridotti si rischia l'intrappolamento in minimi locali.

Sempre considerando istanze di dimensioni significative, la scelta di optare per una strategia *first improvement*, corrispondente al settare il tempo limite al minimo possibile, si è rivelata vincente. Indubbiamente quando un vicinato viene scandito con maggiore accuratezza, i salti verso il basso della funzione obiettivo sono più consistenti, ma questo procedimento può essere portato a termine in maniera efficiente solo per le istanze di piccola dimensione e, dunque, non per quelle di nostro interesse. Già per 2 nodi di destinazione, si osserva bene che l'aumento del tempo per la ricerca non conduce a vantaggi sul tempo totale impiegato dal metodo, ma solo in alcuni casi a fare dei salti più ampi. Dunque, dato che il nostro punto di vista è essenzialmente quello di costruire una buona euristica, in questo caso è decisamente più efficiente avanzare ogni volta con la prima soluzione intera determinata, e dunque con tempo limite uguale a 1.

Se per il tempo limite la scelta più vantaggiosa è, come abbiamo detto, porre per il tempo limite il valore 1 e per il tipo di soluzione iniziale l'albero ricoprente di costo minimo, il valore di k più opportuno da scegliere sembra essere 3. Infatti il valore $k = 1$ va escluso dato che troppo spesso non conduce all'ottimo, salvo trovarsi a dover trattare con un'istanza di dimensioni tanto grandi da poter tralasciare la ricerca dell'ottimo vero e proprio. Fra i valori $k = 3$ e $k = 5$, relativamente all'albero ricoprente di costo minimo il primo è più efficiente, in quanto comporta la ricerca in un vicinato non troppo grande.

Occorre però considerare che anche la soluzione iniziale maglia completa non è priva di interesse. Infatti, per problemi di grandi dimensioni e vicini alla completezza (con un grande numero di nodi di destinazione), è possibile che la soluzione ottima presenti un elevato numero di archi aperti, e dunque non si discosti molto dalla soluzione iniziale suddetta. A seconda dei casi da trattare sarebbe dunque doveroso effettuare un'attenta analisi iniziale per capire come settare i parametri del metodo.

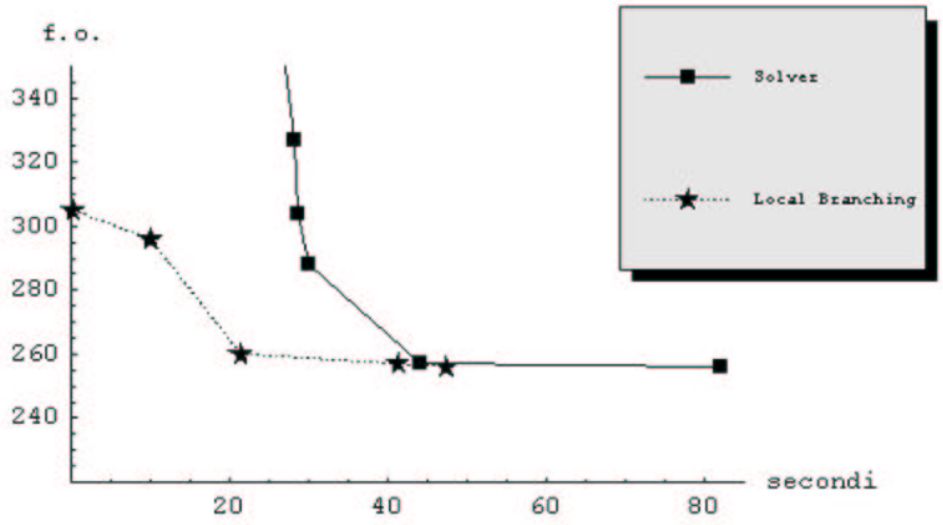

Figura 4.2: Confronto fra il solver ed il Local Branching ($k = 3$, $TL = 1$, soluzione iniziale albero ricoprente di costo minimo) per l'istanza a 6 nodi e 3 destinazioni.

In figura 4.2 è presentato un primo esempio di confronto fra il Branch and Bound del Solver di Xpress ed il metodo del Local Branching implementato. L'istanza è quella di dimensioni medie, ovvero quella a 6 nodi e 3 destinazioni, che il solver risolve in 81 secondi (ed in altri 23 riesce a dimostrare l'ottimalità). Se si effettua un Local Branching partendo dall'albero ricoprente di costo minimo, con i parametri $k = 3$ e $TL = 1$, cioè

con un vicinato di medie dimensioni e con la strategia first improvement, si raggiunge l'ottimo in poco più della metà del tempo rispetto al Solver (44 secondi). E' da notare che non per tutte le scelte di parametri e soluzione iniziale il metodo è migliore del Branch and Bound, e d'altra parte esistono anche altri casi in cui il confronto in favore del Local Branching è ancora più netto. Questo caso può essere dunque una via di mezzo che vuole mostrare che con un'accurata scelta dei parametri, il Local Branching può condurre a soluzioni vicine all'ottimo in tempi più brevi del Solver.

Un secondo esempio molto significativo è riportato in figura 4.3, che rappresenta l'andamento dei due metodi per un'istanza di 7 nodi e 3 destinazioni. Si tratta di un'istanza già più impegnativa, che il solver risolve all'ottimo in circa un'ora e mezza. In questo caso abbiamo provato a confrontarla con il metodo del Local Branching effettuato a partire dalla soluzione iniziale maglia completa, e con i valori dei parametri sempre $k = 3$ e $TL = 1$. Si osserva abbastanza bene quello che viene chiamato effetto "bandiera italiana": immaginando di colorare le tre zone che stanno fra una curva e l'altra con i colori verde, bianco e rosso, in un certo senso si vedrebbe sventolare una bandiera italiana! Al di là di questa fantasiosa interpretazione, il fenomeno è particolarmente significativo, e mostra che vi sono tre fasi nell'andamento dei metodi.

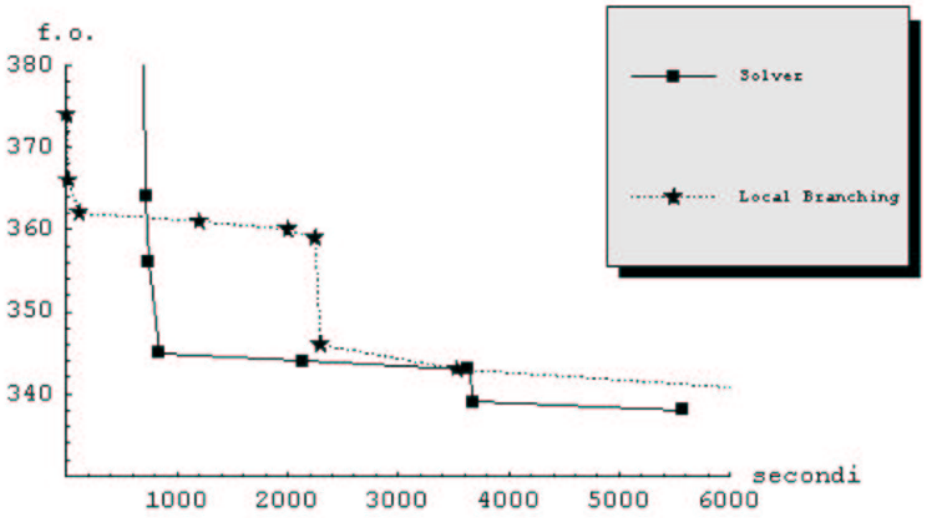

Figura 4.3: Confronto fra il solver ed il Local Branching ($k = 3$, $TL = 1$, soluzione iniziale maglia completa) per l'istanza a 7 nodi e 3 destinazioni.

All'inizio il Local Branching, che si comporta efficacemente come un'euristica, riesce a trovare molto rapidamente soluzioni sempre migliori, mentre il Branch and Bound fatica a trovare le prime soluzioni intere. In una secon-

da fase i due metodi procedono quasi in parallelo, con la stessa velocità, e dunque hanno la medesima efficienza. Nell'ultima fase, quando ci si avvicina alla soluzione ottima, il Branch and Bound supera l'euristica, che risulta decisamente più lenta. Nell'esempio considerato, la prima fase sta all'incirca nell'intervallo di tempo $[0, 700s]$, la seconda in $[700s, 3500s]$, e l'ultima in $[3500s, 17000s]$. Se l'interesse principale è quello di determinare in tempi brevi una soluzione ragionevolmente bassa, il Local Branching riesce a risolvere tale compito in maniera abbastanza soddisfacente.

Per quanto riguarda le istanze complete, i risultati ottenuti sono scarsamente soddisfacenti, dato che il metodo riesce a determinare solamente pochissimi valori, e già arrivato al primo/secondo taglio non riesce a trovare altre soluzioni in tempo utile. Nonostante questo, esso mantiene pur sempre un comportamento decisamente migliore del Branch and Bound, arrivando a determinare una soluzione con Funzione Obiettivo più bassa dell'ultima determinata in tempo utile dal metodo esatto. Visto il notevole tempo impiegato, per queste istanze di grandi dimensioni sono state fatte solamente poche prove particolarmente mirate. Si è scelto il valore $k = 3$ proprio perché era risultato il compromesso migliore nelle istanze di medie dimensioni, e analogamente come soluzione iniziale è stata utilizzata quella costituita dall'albero ricoprente di costo minimo.

Capitolo 5

Conclusioni e sviluppi futuri

Dall'analisi dei risultati effettuata nel capitolo precedente si intuisce che la difficoltà intrinseca del problema permane anche nell'applicazione del metodo del Local Branching che, opportunamente settato, può risultare anche molto migliore del Branch and Bound. I tempi non sono però così soddisfacenti rispetto a quanto ci si poteva aspettare in linea teorica, e questo è dovuto essenzialmente alla scarsa capacità di interazione fra il Solver e l'algoritmo. Infatti sebbene "in teoria" sia ben noto che fissando le variabili $y_{\{ij\}}$ il problema diventa "facile", in quanto richiederebbe solo un calcolo di cammini minimi, nella pratica il Solver non riesce a gestire questa situazione, dato che non riesce a distinguere fra variabili di controllo (le $y_{\{ij\}}$) e variabili derivate.

Tra l'altro questo concetto viene descritto anche in un altro recente lavoro [8], in cui si propone una effettiva suddivisione delle variabili binarie in *primarie* e *secondarie*. Le variabili primarie sono quelle fissate le quali il problema risulta "facile", o almeno "più facile" rispetto a quello originario. Gli autori propongono allora di effettuare dapprima dei tagli di Local Branching sulle variabili primarie, con intorni il più piccolo possibile, e poi, sulla base di questi, effettuare un secondo livello di tagli su quelle secondarie, suddividendo dunque anche concettualmente il problema. E' evidente che la cosa si applicherebbe alla perfezione anche al nostro problema in esame, e potrebbe rappresentare una linea di ricerca futura.

In secondo luogo, si potrebbe avere un miglioramento anche nel caso venisse ampliata maggiormente la flessibilità del Solver, a cui in futuro potrebbero essere specificati esplicitamente quali sono i due tipi di variabili con cui si ha a che fare.

Il metodo, inoltre, potrebbe essere ulteriormente arricchito per esempio considerando la possibilità di introdurre diversificazioni che, per istanze di dimensioni medio/piccole come quelle trattate in questa tesi non sono particolarmente utili, ma che per istanze più grandi potrebbero risultare importanti.

Inoltre andrebbe ampliato lo studio della soluzione iniziale di riferimento in base alla quale far partire il metodo. Abbiamo analizzato cosa deriva dalla scelta di due soluzioni iniziali agli antipodi fra di loro, ed abbiamo constatato che quella costituita dall'albero ricoprente di costo minimo è più efficiente della maglia completa. Probabilmente una scelta intermedia, ad esempio una soluzione iniziale costruita ad hoc tramite un'euristica costruttiva polinomiale, potrebbe risultare ancora migliore.

Per quando riguarda le istanze di dimensioni grandi, ancora molte cose andrebbero studiate al fine di ottenere risultati soddisfacenti. Qui abbiamo studiato solamente dei casi particolari per quanto riguarda istanze complete da 6 e da 7 nodi, in particolare abbiamo usato come dimensione del vicinato il valore $k = 3$, che sembrava il più opportuno. Dato che si vuole cercare di evidenziare però più che altro un comportamento euristico, al fine di riuscire a trovare un numero maggiore di soluzioni intere, potrebbe essere utile studiare anche intorni più piccoli, scegliendo $k = 1$ o anche $k = 2$.

Bibliografia

[1] M. Fischetti, A. Lodi, *Local Branching*, Rapporto Tecnico, Maggio 2002. E' possibile scaricarlo, assieme alla versione aggiornata del Dicembre 2002 e ad una presentazione interattiva di A. Lodi effettuata al MIC 2003 di Kyoto (25-28 Agosto) sul sito: http://www.or.deis.unibo.it/research_pages/ORinstances/MIPs.html.

[2] L. De Giovanni, B. Fortz, M. Labbé, *A compact MILP model for the IP Network Design Problem*, Rapporto Interno, Politecnico di Torino, 2003.

[3] R. Tadei, F. Della Croce, *Ricerca Operativa ed Ottimizzazione*, seconda edizione, Esculapio, 2002.

[4] G. Dantzig, M. Thapa, *Linear Programming*, Springer, 1997.

[5] D. Fransos, *Modelli matematici per la progettazione di reti robuste di telecomunicazioni*, Tesi di Laurea, Dicembre 2002.

[6] S. Amico, *Rilassamenti di un modello matematico per la progettazione di reti IP robuste*, Tesi di Laurea, Marzo 2003.

[7] *Xpress IVE*, software di ottimizzazione sviluppato dalla Dash Optimization basasto sul linguaggio Mosel, informazioni e manuali di riferimento, relativi sia al Solver sia al linguaggio, reperibili al sito: http://www.dashoptimization.com.

[8] M. Fischetti, C. Polo, M. Scantamburlo, *A Local Branching Heuristic for Mixed-Integer Programs with 2-Level Variables*, Rapporto Tecnico, Aprile 2003.

Appendice A

Listato del programma in Mosel IVE

Di seguito riportiamo il listato del programma che esegue il metodo del Local Branching, nella versione descritta nel capitolo 4. Le variabili utilizzate hanno quasi sempre esattamente gli stessi nomi delle variabili descritte nel modello teorico, salvo le y^d_{ij}, che per evitare l'omonimia con le $y_{\{ij\}}$ prendono il nome di $ips(i, j, d)$.

Per una descrizione dettagliata del linguaggio Mosel e dei comandi di Xpress IVE si rimanda a [7].

```
model localbranching

uses "mmxprs", "mmetc", "mmsystem"

Istanza:= "Istanza_6n_3d.dat"

!----------------------
!Dimensioni del problema
!----------------------
declarations
    Nodi: integer
    Richieste: integer
    tempoinizio: real
end-declarations

initializations from Istanza
    Nodi              !Numero di nodi
    Richieste         !Numero di richieste
end-initializations
```

```
declarations
    N = 1..Nodi       !insieme dei nodi
    K = 1..Richieste  !insieme delle richieste
end-declarations

!----------------------
!Richieste
!----------------------
declarations
    o: array(K) of integer !Origini delle richieste
    d: array(K) of integer !Destinazioni delle richieste
end-declarations

initializations from Istanza
    o
    d
end-initializations

!----------------------
!Topologia
!----------------------
!(1 in (i,j) significa che esiste l'arco (i,j))
declarations
    T: array(N,N) of integer !Matrice di incidenza n-n
end-declarations

initializations from Istanza
    T
end-initializations

!----------------------
!Come utilizzare gli insiemi degli Archi e degli Spigoli
!----------------------

    !Insieme degli archi (i,j)
        !Basta usare la sintassi
        !forall(i in N, j in N | T(i,j) = 1) niente:=1
    !Insieme degli spigoli {i,j} (poniamo i<j)
        !Basta usare la sintassi
        !forall(i in N, j in N | i<j and T(i,j) = 1) niente:=1

!----------------------
!Altri insiemi utili
!----------------------
```

```
declarations
    DK: set of integer
    O: dynamic array(N,N) of integer
    Kod: dynamic array(N,N,K) of integer
end-declarations

forall(i in K) DK += {d(i)}

forall(i in K) O(d(i),o(i)):= 1
        !L'array dinamico O(d,i) contiene in ogni riga d
        !solamente degli 1 in corrispondenza di nodi a partire
        !dai quali esiste un flusso che termina in d

    !Per utilizzare l'insieme Od useremo la sintassi
    !d:=4
    !forall(i in N | exists(O(d,i))) niente:=1

forall(i in K) Kod(o(i),d(i),i):= 1
        !L'array dinamico Kod(o,d,k) contiene per ogni
        !coppia (o,d) solamente degli 1 in corrispondenza
        !di richieste che partono da o e terminano in d

    !per utilizzare l'insieme Kod useremo la sintassi
    !o:=1
    !d:=2
    !forall(k in K | exists(Kod(o,d,k))) niente:=1

!----------------------
!Costanti
!----------------------
declarations
    c(!i,j!): array(N,N) of integer
        !costi di installazione link per {i,j} in E
    q(!i,j!): array(N,N) of integer
        !capacità di un link su {i,j} in E
    w(!i,j!): array(N,N) of integer
        !costi di instradamento per (i,j) in A
    p(!k!): array(K) of integer
        !capacità della richiesta k in K
    Md(!d!): array(N) of integer
    ML(!{i,j}!): dynamic array(N,N) of integer
    MO: integer
    eps: integer
end-declarations
```

```
initializations from Istanza
    c
    q
    w
    p
end-initializations

!Definisco Md(i) sommando tutte le richieste dirette al nodo d
forall(i in N) Md(i):= sum(j in N | exists(O(i,j)))
    sum(l in K | exists(Kod(j,i,l))) p(l)

!Faccio sì che su tutti gli spigoli possano essere
!installati al più 100 links
forall(i in N, j in N | i<j and T(i,j) = 1) ML(i,j):= 100

!La costante MO assume come valore la somma degli
!N-1 archi più lunghi della rete
declarations
    tt: array(N) of integer
    tc: integer
    tz: integer
    tk: integer
    te: integer
end-declarations
tc:=1
forall(i in N, j in N | T(i,j)=1) do
    if (tc<Nodi) then
        tt(tc):= w(i,j)
        tc := tc + 1
    else
        tz:=1
        tk:=w(i,j)
        repeat
            if (tt(tz)<tk) then
                te:= tk
                tk:= tt(tz)
                tt(tz):= te
            end-if
            tz:= tz + 1
        until (tz>=Nodi)
    end-if
end-do
tt(Nodi):= 0
```

```
MO:= sum(i in N) tt(i)

!Fisso il valore della costante eps a 1
eps:= 1


!----------------------
!Variabili
!----------------------
declarations
    y: dynamic array(N,N) of mpvar
    x: dynamic array(N,N) of mpvar
    f: dynamic array(N,N,N) of mpvar
    ips: dynamic array(N,N,N) of mpvar
    pi: dynamic array(N,N) of mpvar
end-declarations

!Creo solo le variabili che corrispondono ad archi
!che esistono realmente
forall(i in N, j in N | i<j and T(i,j) = 1) do
    create(y(i,j))
    create(x(i,j))
    y(i,j) is_binary
    x(i,j) is_integer
end-do

forall(i in N, j in N, l in DK| T(i,j) = 1) do
    create(f(i,j,l))
    create(ips(i,j,l))
    ips(i,j,l) is_binary
end-do

forall(i in N, l in DK) create(pi(i,l))


!----------------------
!Funzione Obiettivo
!----------------------
FunzioneObiettivo:=
    sum(i in N, j in N | i<j and T(i,j) = 1) c(i,j)*x(i,j)


!----------------------
!Vincoli
```

```
!----------------------

!vincolo (2)
conto:=1
forall(i in N, j in N | T(i,j) = 1) do
    if (i<j) then
        vincolo2(conto):= sum(l in DK) f(i,j,l) <= q(i,j)*x(i,j)
    else
        vincolo2(conto):= sum(l in DK) f(i,j,l) <= q(j,i)*x(j,i)
    end-if
    conto += 1
end-do

!vincolo (3)
conto:=1
forall(l in DK, i in N) do
  if (exists(O(l,i))) then
    vincolo3(conto):= (sum(j in N | T(i,j) = 1) f(i,j,l)) -
        (sum(j in N | T(j,i) = 1) f(j,i,l)) =
        sum(h in K | exists(Kod(i,l,h))) p(h)
  elif (i=l) then
    vincolo3(conto):= (sum(j in N | T(j,i) = 1) f(j,i,l)) -
        (sum(j in N | T(i,j) = 1) f(i,j,l)) = Md(l)
  else
    vincolo3(conto):= (sum(j in N | T(i,j) = 1) f(i,j,l)) -
        (sum(j in N | T(j,i) = 1) f(j,i,l)) = 0
  end-if
  conto += 1
end-do

!vincolo (4)
conto:=1
forall(i in N, j in N, l in N, h in DK |
                                T(i,j) = 1 and T(i,l) = 1) do
    vincolo4(conto):=
      f(i,j,h) <= f(i,l,h) + Md(h)*(2 - ips(i,j,h) - ips(i,l,h))
    conto += 1
end-do

!vincolo (5)
conto:=1
forall(i in N, j in N, l in DK | T(i,j) = 1) do
    vincolo5(conto):= f(i,j,l) <= Md(l)*ips(i,j,l)
    conto += 1
```

```
end-do

!vincolo (6)
conto:=1
forall(i in N, j in N, l in DK | T(i,j) = 1) do
  if (i<j) then
    vincolo6(conto):=
        pi(i,l)-pi(j,l) <= w(i,j) + eps*(ips(i,j,l)-1) +
            MO*(1-y(i,j)) + (eps-w(i,j)-w(j,i))*ips(j,i,l)
  else
    vincolo6(conto):=
        pi(i,l)-pi(j,l) <= w(i,j) + eps*(ips(i,j,l)-1) +
            MO*(1-y(j,i)) + (eps-w(i,j)-w(j,i))*ips(j,i,l)
  end-if
  conto += 1
end-do

!vincolo (7)
conto:=1
forall(i in N, j in N, l in DK | i<j and T(i,j) = 1) do
    vincolo7(conto):= ips(i,j,l) + ips(j,i,l) <= y(i,j)
    conto += 1
end-do

!vincoli (8) e (9)
conto:=1
forall(i in N, j in N | i<j and T(i,j) = 1) do
    vincolo8(conto):= x(i,j) <= ML(i,j)*y(i,j)
    vincolo9(conto):= y(i,j) <= x(i,j)
    conto += 1
end-do


!----------------------
!Algoritmo per il Local Branching
!----------------------

!Definisco le variabili che serviranno per l'algoritmo
declarations
    !Variabili contenenti la soluzione precedente
    ys: dynamic array(N,N) of integer

    !Variabili contenenti la migliore soluzione
    yb: dynamic array(N,N) of integer
```

```
    !Variabili contenenti la soluzione che va sul taglio
    yv: dynamic array(N,N) of integer

    !Variabili di Upper Bound
    ub: integer
    ubb: integer

    !Variabili della soluzione iniziale
    yi: dynamic array(N,N) of integer
end-declarations

!Inizializzo le nuove variabili
forall(i in N, j in N | i<j and T(i,j) = 1) do
    ys(i,j):= 0
    yc(i,j):= 0
    yb(i,j):= 0
    yi(i,j):= 0
end-do

!Leggo da un file di dati la soluzione iniziale (albero
!ricoprente di costo minimo oppure maglia completa)
diskdata(ETC_IN, "yi.dat", yi)

!Cerco una soluzione intera di partenza con i dati
!sulla soluzione iniziale
conto:= 0
forall(i in N, j in N | i<j and T(i,j) = 1) do
    conto += 1
    ys(i,j):= yi(i,j)
    viniz(conto):= y(i,j) = round(yi(i,j))
end-do
tempoinizio:= gettime
minimize(FunzioneObiettivo)
writeln("Tempo impiegato: ", gettime-tempoinizio)
ub:= round(getobjval)
writeln("Soluzione intera di partenza")
writeln("Funz.Obiett.: ", ub)
writeln("--------")
forall(i in N, j in N | i<j and T(i,j) = 1) do
    writeln("y{",i,",",j,"} = ", getsol(y(i,j)))
end-do
writeln("---")
forall(i in 1..conto) do
```

```
    sethidden(viniz(i), true)
end-do

!Inizializzo anche la soluzione migliore
ubb:= ub
forall(i in N, j in N | i<j and T(i,j) = 1) do
    yb(i,j):= ys(i,j)
end-do


!Procedura di Local Branching
declarations
    tot: integer
    contatore: integer
    k: integer
    stato: array({XPRS_OPT,XPRS_UNF,0,XPRS_UNB}) of string
    TL: real
    temporicercafinale: integer
end-declarations
stato:= ["Ottimo determinato", "Non terminato",
                    "Infeasible", "Illimitato"]
k:= 3
contatore:=0
tot:= 20
TL:= 10
temporicercafinale:= 0
repeat
  contatore += 1

  !Copio la soluzione precedentemente determinata in
  !quella che andrà a costituire il vincolo
  forall(i in N, j in N | i<j and T(i,j) = 1) do
    yc(i,j):= ys(i,j)
  end-do

  !Aggiungo il taglio di local branching
  locbray(2*contatore - 1):=
    (sum(i in N, j in N | i<j and T(i,j)=1 and yc(i,j)=1)
                                    (yc(i,j)-y(i,j))) +
    (sum(i in N, j in N | i<j and T(i,j)=1 and yc(i,j)=0)
                                    (y(i,j)-yc(i,j))) <= k

  !Pongo l'upper bound
  UB(contatore):=
```

```
  sum(i in N, j in N | i<j and T(i,j) = 1)
                              c(i,j)*x(i,j) <= ubb - 1

!Controllo parametri
setparam("XPRS_MAXTIME",TL)

!Trovo la nuova soluzione
tempoinizio:= gettime
minimize(FunzioneObiettivo)
writeln("Tempo: ", gettime - tempoinizio)
writeln("Stato: ", stato(getprobstat))
if ((stato(getprobstat) <> "Ottimo determinato") and
               (stato(getprobstat) <> "Infeasible")) then
  !Inverto l'ultimo taglio e poi faccio una
  !ricerca esaustiva su quello che resta
  sethidden(locbray(2*contatore - 1),true)
  locbray(2*contatore):=
    (sum(i in N, j in N | i<j and T(i,j)=1 and yc(i,j)=1)
                              (yc(i,j)-y(i,j))) +
    (sum(i in N, j in N | i<j and T(i,j)=1 and yc(i,j)=0)
                              (y(i,j)-yc(i,j))) >= (k + 1)
  setparam("XPRS_MAXTIME",temporicercafinale)
  tempoinizio:= gettime
  minimize(FunzioneObiettivo)
  writeln("---")
  writeln("Soluzione dopo la ricerca finale")
  writeln("Tempo: ", gettime - tempoinizio)
  writeln("Stato: ", stato(getprobstat))
  if ((stato(getprobstat) = "Ottimo determinato") or
          (stato(getprobstat) = "Infeasible")) then
      ub:= round(getobjval)
      writeln("Funz.Obiett.:", ub)
      writeln("--------")
      forall(i in N, j in N | i<j and T(i,j) = 1) do
          writeln("y{",i,",",j,"} = ", getsol(y(i,j)))
      end-do
      writeln("---")
      if (ub < ubb) then
          ubb:= ub
          forall(i in N, j in N | i<j and T(i,j)=1) do
              yb(i,j):= round(getsol(y(i,j)))
          end-do
      end-if
  end-if
```

```
    break
  end-if
  ub:= round(getobjval)
  writeln("Soluzione dopo il taglio ",contatore)
  writeln("Funz.Obiett.: ", ub)
  writeln("--------")
  forall(i in N, j in N | i<j and T(i,j) = 1) do
    writeln("y{",i,",",j,"} = ", getsol(y(i,j)))
  end-do
  writeln("---")

  !Ottimo determinato nel tempo prestabilito
  if (stato(getprobstat) = "Ottimo determinato") then
    !Inverto il taglio
    sethidden(locbray(2*contatore - 1),true)
    locbray(2*contatore):=
      (sum(i in N, j in N | i<j and T(i,j)=1 and yc(i,j)=1)
                              (yc(i,j)-y(i,j))) +
      (sum(i in N, j in N | i<j and T(i,j)=1 and yc(i,j)=0)
                              (y(i,j)-yc(i,j))) >= (k + 1)

  !Tempo terminato
  elif (stato(getprobstat) = "Infeasible") then
    !Elimino il taglio ma non posso invertirlo
    sethidden(locbray(2*contatore - 1),true)
  end-if

  !Elimino anche il taglio di Upper Bound,
  !che verrà aggiornato
  sethidden(UB(contatore), true)

  !Controllo se la nuova soluzione determinata è la migliore
  if (ub < ubb) then
    ubb:= ub
    forall(i in N, j in N | i<j and T(i,j)=1) do
        yb(i,j):= round(getsol(y(i,j)))
    end-do
  end-if

  !Salvo le variabili binarie in nelle corrispondenti variabili s
  forall(i in N, j in N | i<j and T(i,j)=1) do
    ys(i,j):= round(getsol(y(i,j)))
  end-do
until (contatore>=tot)
```

```
!Riscrivo la migliore soluzione trovata
writeln("La migliore soluzione trovata è")
writeln("--------")
writeln("Funz.Obiett.: ", ubb)
forall(i in N, j in N | i<j and T(i,j) = 1) do
    writeln("y{",i,",",j,"} = ", yb(i,j))
end-do

end-model
```

Appendice B

Esempio di Istanza

Mostriamo a titolo di esempio un file di istanza che viene letto dal programma. Si tratta di un'istanza completa da 6 nodi (la stessa visualizzata graficamente in figura 2.1), quindi con 30 destinazioni. Il file che la contiene si chiamerebbe `istanza_6n_6d.dat`

```
Nodi: 6
Richieste: 30

T:
[0 1 1 0 1 0
 1 0 1 1 0 0
 1 1 0 1 1 1
 0 1 1 0 0 1
 1 0 1 0 0 1
 0 0 1 1 1 0]

c:
[0 6 4 0 7 0
 6 0 5 5 0 0
 4 5 0 3 7 7
 0 5 3 0 0 7
 7 0 7 0 0 8
 0 0 7 7 8 0]

q:
[0 3 3 0 1 0
 3 0 2 2 0 0
 3 2 0 1 1 2
 0 2 1 0 0 4
 1 0 1 0 0 2
 0 0 2 4 2 0]
```

```
w:
[0 2 2 0 2 0
 1 0 1 3 0 0
 1 2 0 1 1 2
 0 2 1 0 0 2
 1 0 1 0 0 2
 0 0 2 1 3 0]

o: [1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6]
d: [2 3 4 5 6 1 3 4 5 6 1 2 4 5 6 1 2 3 5 6 1 2 3 4 6 1 2 3 4 5]
p: [4 2 1 5 7 8 4 1 2 3 4 8 6 6 7 9 4 2 1 3 2 7 5 6 1 7 2 3 2 1]
```