Relazione finale del corso di Ricerca Operativa 2
a.a. 2012/13

Alberto Franzin 1012883, Ludovico Minto 626599

10 marzo 2014

Figura 1: Commis-Voyageur tour (1832)

Sommario

In questa relazione presentiamo il lavoro svolto durante il corso di Ricerca Operativa 2, aa 2012/13, riguardante diverse tecniche per la risoluzione di un problema di programmazione mista intera (MIP), applicate al Problema del Commesso Viaggiatore (TSP). Sono stati implementati approcci esatti con branch-and-bound e solver MIP commerciali, e metodi euristici. Le soluzioni proposte sono state testate su istanze note della libreria TSPLIB.

Indice

1 Introduzione **3**

2 Rilassamento Lagrangiano e Branch-and-Bound **5**
2.1 1-Albero Minimo 5
2.2 Rilassamento Lagrangiano 6
2.3 Metodo del Subgradiente 7
2.3.1 Subgradiente PHK 9
2.3.2 Subgradiente PVJ 9
2.3.3 Subgradiente KVJ 13
2.4 Branch-and-Bound 15
2.4.1 B & B Prim-based 17
2.4.2 B & B Kruskal-based 17
2.4.3 Riduzione dei lati 17
2.4.4 Propagazione dei vincoli 18
2.5 Risultati computazionali 18
2.5.1 Subgradiente lagrangiano 18
2.5.2 Branch-and-bound 21
2.6 Commenti 26

3 Risolutori basati sul modello MIP **28**
3.1 Modello MIP del TSP 28
3.1.1 Preprocessing 29
3.2 Risoluzione iterativa 29
3.3 Risoluzione con callback “alla Miliotis” 30
3.4 Risoluzione con callback sulla soluzione frazionaria 30
3.5 Risultati computazionali 31
3.6 Commenti 34

4 Euristici **37**
4.1 Euristici basati sulla formulazione del problema 37
4.1.1 Nearest Neighbour 37
4.1.2 k-opt 37
4.1.3 Branch-and-bound con prefissaggio 40

4.2 Euristici basati su modello MIP 40
4.2.1 Hard Fixing 41
4.2.2 Proximity Search 41
4.2.3 RINS+Polishing 42
4.3 Risultati computazionali 42
4.3.1 NN, RC e k-opt 42
4.3.2 B&B con prefissaggio 45
4.3.3 Euristici basati su solver MIP 48
4.4 Commenti 52

5 Conclusioni **54**

Capitolo 1

Introduzione

Il Problema del Commesso Viaggiatore (Traveling Salesman Problem, TSP) è probabilmente il problema di ottimizzazione più noto, a causa della sua formulazione di immediata comprensione, della sua impostazione molto "visuale" e di un campo di applicabilità che spazia dalla logistica alla bioinformatica che lo rendono adatto come benchmark, anche dal punto di vista didattico, per diverse tecniche di risoluzione.

Obiettivo del TSP è trovare il ciclo hamiltoniano di costo minimo in un grafo completo, cioè il percorso che tocca tutti i nodi del grafo e torna al nodo di partenza, selezionando l'ordine dei nodi (e quindi gli archi attraversati) in maniera tale che la somma totale dei costi degli archi attraversati sia la più bassa possibile.

Il TSP (nella sua versione simmetrica) è matematicamente definito per un grafo completo $G = (V, E)$ come:

$$\min \sum_{e \in E} c_e x_e \tag{1.1}$$

rispetto a

$$\sum_{e \in \delta(v)} x_e = 2 \quad \forall\, v \in V \tag{1.2}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad \forall\, S \subset V,\ S \neq \emptyset \tag{1.3}$$

$$x_e \in \{0, 1\} \quad \forall\, e \in E. \tag{1.4}$$

Nella funzione obiettivo 1.1 le variabili x_e sono variabili indicatrici intere (vincolo 1.4) associate agli archi che assumono valore 1 se l'arco corrispondente è selezionato nella soluzione ottima e 0 altrimenti, e c_e è il costo loro associato. Il vincolo di grado 1.2 impone che su nodo incidano due e solo due archi, mentre la famiglia di vincoli 1.3 vieta la presenza di cicli di dimensione inferiore a $|V|$. Ci sono più modi per specificare questa condizione, oltre

alla formulazione di tipo knapsack qui riportata; questi vincoli prendono il nome di SECs, sigla per subtour elimination constraints.

Il TSP è un problema NP-completo: ciò significa che al giorno d'oggi un algoritmo efficiente non è ancora conosciuto, e forse non esiste nemmeno. Il numero di possibili tour definibili su un grafo cresce in maniera proporzionale al fattoriale del numero di nodi, pertanto un approccio brute-force è impraticabile anche solo per grafi di pochi nodi. Negli anni, moltissimi autori hanno approcciato il problema con varie tecniche spesso innovative, come ad esempio il branch-and-cut in Padberg and Rinaldi [13]; tuttavia, anche l'algoritmo ad oggi più efficiente fallisce quando eseguito su istanze molto grandi. Per istanze di prova di centinaia di migliaia o milioni di nodi, attualmente non è ancora conosciuta la soluzione ottima[1]. Una cronologia dei vari approcci al problema, nonché una vasta presentazione delle tecniche, si trova in Applegate, Bixby, Chvatal, and Cook [2], volume accompagnato dal software Concorde[2] [1], attualmente il più efficace risolutore di istanze di TSP.

In questa relazione esponiamo alcune di queste tecniche, sviluppate durante il corso di Ricerca Operativa 2, accompagnate da risultati computazionali e relative considerazioni. Le varie tecniche sono state implementate in linguaggio `c`, usando nella seconda parte del corso i metodi per usare il solver IBM ILOG CPLEX^{TM}12.5[3]. I test sono stati effettuati su istanze della libreria TSPLIB (Reinelt [14]), che contiene diverse istanze note del TSP, con dimensioni che vanno dalle poche decine alle migliaia di nodi, che viene normalmente usata come riferimento per i lavori su TSP presenti in letteratura. In particolare, le tecniche sviluppate durante il corso vengono testate su un testbed di istanze geografiche ed euclidee (in cui i costi rispettano la disuguaglianza triangolare) di taglia inferiore ai 1000 nodi.

La relazione è strutturata come segue: nel capitolo 2 è descritto il primo approccio è basato sul rilassamento lagrangiano del problema, sul quale si costruisce un algoritmo branch-and-bound. Nel capitolo 3 il problema viene formulato come problema di programmazione mista intera, e vengono presentate alcune tecniche per la sua risoluzione esatta. Nel capitolo 4 invece si presentano alcune tecniche euristiche per la risoluzione del TSP. Infine, esponiamo le conclusioni ottenute.

[1] si veda http://www.math.uwaterloo.ca/tsp/

[2] http://www.math.uwaterloo.ca/tsp/concorde.html

[3] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Capitolo 2

Rilassamento Lagrangiano e Branch-and-Bound

La prima tecnica di risoluzione che affronteremo utilizza un approccio di tipo branch-and-bound, in cui il calcolo di un lower bound alla soluzione ottima si traduce nella computazione di un 1-albero minimo. Dato un insieme di vertici $V = \{1, ..., n\}$, un *1-albero* su V è un grafo non diretto connesso ottenuto aggiungendo ad un albero non diretto sui vertici $V \setminus \{1\}$ due lati distinti incidenti in 1. L'osservazione che sta alla base del metodo qui presentato è la seguente: un ciclo hamiltoniano è un 1-albero in cui ogni vertice ha grado uguale a 2. Dato un grafo $G = (V, E, c)$ connesso, non diretto e pesato, il costo di un ciclo hamiltoniano o *tour* H su un tale grafo è non minore del minimo costo di un 1-albero sullo stesso grafo. In particolare, se H^* è un tour ottimo in G, si ha che:

$$c(H^*) \geq \min\{c(T) : T \text{ 1-albero su } V\} \tag{2.1}$$

2.1 1-Albero Minimo

Un primo lower bound alla soluzione ottima è quello che si ottiene considerando il costo di un 1-albero minimo. La computazione di tale lower bound può essere facilmente ricondotta al calcolo del costo di un minimum spanning tree sui vertici $\{2, ..., n\}$, al quale dovrà essere sommato il costo dei due lati di costo minimo incidenti in 1. La costruzione di un albero ricoprente minimo può essere effettuata utilizzando l'algoritmo Prim, uno tra i più efficenti nel caso si considerino grafi completi. Benchè la complessità computazionale dell'algoritmo sia relativamente ridotta e consenta di ottenere un lower bound in tempi brevi anche per grandi istanze, la scarsa qualità del bound generalmente non permette ad un algoritmo di tipo branch-and-bound di restringere in modo sufficientemente veloce lo spazio di ricerca.

In figura 2.1 due esempi di 1-albero minimo, calcolati sule istanze `lin105` e `lin318`.

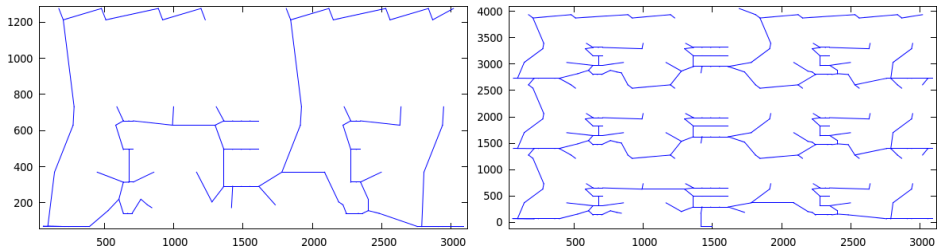

Figura 2.1: 1-albero per `lin105` e `lin318`.

2.2 Rilassamento Lagrangiano

Riprendendo le considerazioni precedenti, possiamo dire che l'eliminazione dei vincoli di grado permette da una parte di poter disporre in modo molto efficiente di un lower bound alla soluzione ottima, dall'altra rende tuttavia il valore ottenuto poco significativo perchè molto al di sotto della soluzione ottima. Il rilassamento lagrangiano che ora andremo a considerare, tenta di migliorare il bound cercando di ristabilire i corretti vincoli di grado per ogni vertice. Il *rilassamento lagrangiano* è una tecnica di rilassamento per problemi di ottimizzazione: l'idea di base è quella di eliminare dal problema originale una parte di vincoli, possibilmente proprio quei vincoli che rendono difficile la risoluzione del problema, inserendoli nella funzione obiettivo per tenerne in qualche modo conto durante la risoluzione.

Nel nostro caso, andremo a considerare il rilassamento lagrangiano che si ottiene eliminando dal modello i vincoli di grado sui vertici $\{2, ..., n\}$, per inserirli nella funzione obiettivo come somma pesata secondo opportuni coefficienti o *moltiplicatori lagrangiani* $\pi_2, ..., \pi_n$. È da notare come i moltiplicatori non siano vincolati in segno, dal momento che i vincoli eliminati sono in forma di equazioni. Il problema rilassato è detto *problema lagrangiano*, e può essere risolto efficientemente: un algoritmo per la costruzione di un 1-albero minimo basato ad esempio sull'algoritmo di Prim per il calcolo di un MST ci porge una facile soluzione.

$$\min \sum_{e \in E} c_e x_e - \sum_{i=2}^{n} \pi_i \Big(\sum_{e \in \delta(v)} x_e - 2 \Big) \tag{2.2}$$

$$\sum_{e \in \delta(1)} x_e = 2 \tag{2.3}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad \forall\, S \subset V \setminus \{1\},\ |S| \geq 2 \tag{2.4}$$

$$\sum_{e \in E \setminus \delta(1)} x_e = n - 2 \tag{2.5}$$

$$x_e \in \{0, 1\} \quad \forall\, e \in E. \tag{2.6}$$

Come si può osservare, sono stati rilassati i vincoli 1.1 per $v \in V \backslash \{1\}$, mentre 2.5, 2.4, 2.5, 2.6 corrispondono al modello matematico di un 1-albero: le soluzioni ammissibili del problema lagrangiano sono tutti e solo gli 1-alberi del grafo. La funzione:

$$L(\pi) = \min \left\{ \sum_{e \in E} c_e x_e - \sum_{i=2}^{n} \pi_i (\sum_{e \in \delta(v)} x_e - 2) : x \in X \right\} \tag{2.7}$$

dove X è l'insieme dei vettori di incidenza degli 1-alberi sul grafo, è detta *funzione lagrangiana*.

Il valore $L(\pi)$ costituisce un lower-bound al costo di una soluzione ottima del problema originario (che ricordiamo essere un problema di minimizzazione), per qualsiasi vettore di moltiplicatori π. Per ottenere un buon lower bound si può quindi cercare di massimizzare il valore di $L(\pi)$ al variare di π, ovvero cercare di risolvere il *problema lagrangiano duale*.

2.3 Metodo del Subgradiente

La funzione lagrangiana $L : \pi \longrightarrow L(\pi)$ è una funzione continua, lineare a tratti e concava. La ricerca del massimo è complicata dal fatto che la funzione non è differenziabile in ogni suo punto: uno degli approcci più seguiti è quello che utilizza il *metodo del subgradiente*, che tenta di trovare una soluzione generando iterativamente una successione di vettori $\pi^0, \pi^1, ..., \pi^k$ convergente all'ottimo, secondo una regola generale del tipo:

$$\pi_i^{k+1} = \pi_i^k + t^k s_i^k \qquad i = 1, ..., n \tag{2.8}$$

dove s^k è un subgradiente di L in π^k, mentre t^k è l'ampiezza dello spostamento che l'algoritmo compie nella direzione determinata dal subgradiente. Con riferimento al modello matematico del problema originario, si dimostra che $(Ax^k - 2)$ è un subgradiente di $L(\pi^k)$ in π^k, dove A è la matrice dei vincoli di grado. In altre parole, la componente i-esima del vettore subradiente all'iterazione k-esima è dato da $s_i^k = d_i - 2$, dove d_i è il numero di lati incidenti sul vertice i dell'1-albero calcolato in quell'iterazione. La scelta del passo, che peraltro condiziona pesantemente la rapidità con cui l'algoritmo si avvicina all'ottimo, risulta al contrario meno chiara. Si dimostra tuttavia in Held and Karp [8] che la scelta di un passo che verifica le condizioni:

$$\lim_{k \to \infty} t^k = 0 \tag{2.9}$$

$$\sum_{k=0}^{\infty} t^k = +\infty \tag{2.10}$$

garantisce la convergenza all'ottimo della successione di punti π^k calcolati ad ogni iterazione seguendo la direzione $s^k / \parallel s^k \parallel$. La formula proposta da

Held e Karp è la seguente:

$$t^k = \alpha^k \frac{UB - L(\pi^k)}{\| s^k \|^2} \tag{2.11}$$

dove UB è un upper bound, possibilmente di buona qualità, al valore della soluzione ottima del problema originario, mentre α^k è uno scalare inizialmente posto uguale a 2, e fatto decrescere al procedere dell'algoritmo.

Nonostante 2.11 rispetti le due condizioni sufficienti per la convergenza 2.9 e 2.10, nessuna garanzia è data circa la velocità con cui l'algoritmo si avvicina all'ottimo. Un metodo alternativo per aggiornare il passo e portare più rapidamente l'algoritmo in prossimità dell'ottimo è invece quello proposto da Volgenant e Jonker. Volgenant and Jonker [16] impongono che la successione di passi t^k debba rispettare il vincolo imposto da una equazione alle differenze del secondo ordine del tipo:

$$t^{k+1} - 2t^k + t^{k-1} = costante \tag{2.12}$$

unitamente alle condizioni $t^1 - t^2 = 3(t^{M-1} - t^M)$ e $t^M = 0$, dove M è il massimo numero di iterazioni che si intende far compiere all'algoritmo. La regola di aggiornamento del passo che ne consegue è la seguente:

$$t^k = t^1 \frac{k^2 - 3(M-1)k + M(2M-3)}{2(M-1)(M-2)} \tag{2.13}$$

Come si può notare, la regola è completamente specificata una volta scelto il numero massimo di iterazioni M e l'ampiezza del primo passo t^1, mentre non richiede, contrariamente alla regola di Held e Karp, la conoscenza di un buon upper bound per poter essere utilizzata. Come vedremo, sarà proprio questa caratteristica a permetterci di sfruttare il metodo del subgradiente anche nei nodi interni dell'albero di ricerca del branch-and-bound. Sempre in [16], viene suggerita la seguente formula per il calcolo della nuova posizione ad ogni iterazione:

$$\pi_i^{k+1} = \pi_i^k + 0.6\, t^k(d_i^k - 2) + 0.4\, t^k(d_i^{k-1} - 2) \qquad i = 1, ..., n \tag{2.14}$$

dove l'inserimento del termine $t^k(d_i^{k-1} - 2)$, ovvero della componente i-esima del subgradiente all'iterazione precedente, ha lo scopo di smorzare le oscillazioni nella direzione di spostamento che incorrono quando l'algoritmo si trova a confine di due o più zone in cui la funzione lagrangiana assume differenti gradienti (fenomeno detto zig-zagging).

Una nostra prima implementazione cerca di risolvere il problema lagrangiano duale utilizzando il metodo del subgradiente e le regole 2.11 e 2.8 per l'aggiornamento del passo e della posizione. Una seconda implementazione segue invece le regole 2.13 e 2.14. In entrambi i casi, la costruzione dell'1-albero viene effettuata sfruttando l'algoritmo di Prim per la

costruzione di alberi di supporto minimo. Proponiamo infine una terza soluzione, in cui l'albero di supporto minimo viene calcolato con l'algoritmo di Kruskal, mentre l'avanzamento dell'algoritmo del subgradiente è regolato da 2.13 e 2.14. Chiameremo di seguito le tre soluzioni rispettivamente come PHK (Prim-Held-Karp), PVJ (Prim-Volgenant-Jonker) e KVJ (Kruskal-Volgenant-Jonker).

Le immagini 2.2, 2.3 e 2.4 mostrano l'andamento dell'aggiornamento del lower bound, vale a dire come varia il bound quando viene individuata una soluzione migliorante. Le immagini 2.5, 2.6 e 2.7, invece, mostrano il valore del bound ad ogni iterazione del subgradiente.

2.3.1 Subgradiente PHK

Il primo metodo è quello proposto inizialmente in [8]. Come anticipato, in questa implementazione la costruzione di un 1-albero ad ogni iterazione segue l'algoritmo di Prim. Una particolare attenzione è stata posta nell'adattare l'algoritmo per renderlo compatibile con la nostra decisione di trattare in modo esplicito i vincoli sui lati del grafo, senza dover ricorrere ad artifici come l'assegnazione di un costo molto grande o molto piccolo rispettivamente per vietare o forzare un lato. Questa scelta ha forse complicato alcune porzioni di codice, ma ci ha permesso di procedere senza dover prestare attenzione ad eventuali comportamenti imprevisti dovuti a combinazioni di costi particolarmente elevati.

La messa a punto di parametri quali il coefficiente α^k o il numero massimo di iterazioni si è rivelata particolarmente ostica, mostrando perlomeno inizialmente risultati poco soddisfacenti, oppure molto buoni ma solo su ristretti gruppi di istanze, su cui i parametri risultavano evidentemente sovradattati. Ricalcando alcune idee utilizzate in [16], è stato deciso di utilizzare un numero massimo M di iterazioni proporzionale a n^2, la dimensione dell'istanza; α viene invece dimezzato ogni qual volta viene superato un certo numero h di iterazioni consecutive senza conseguire alcun miglioramento nella valutazione del massimo della funzione lagrangiana. Il valore di h è stato calcolato come segue: se h fosse il tempo medio di dimezzamento di α, ci sarebbero all'incirca M/h dimezzamenti nel corso dell'algoritmo; imponendo $\alpha_u = 0.001$ all'ultimo aggiornamento $u = M/h$, si trova $h = M/(\log_2 \alpha_1 - \log_2 0.001)$, dove $\alpha_1 = 2$ è il valore assunto inizialmente.

In figura 2.8 mostriamo dei dettagli dell'aggiornamento con PHK per `gr137` come riportato in figura 2.5.

2.3.2 Subgradiente PVJ

In questa implementazione il passo e la direzione sono aggiornati seguendo le regole 2.13 e 2.14, mentre il numero massimo di iterazioni è pari a $n^2/50 + n + 16$, come suggerito in [16]. Il passo di base t^1 che compare in

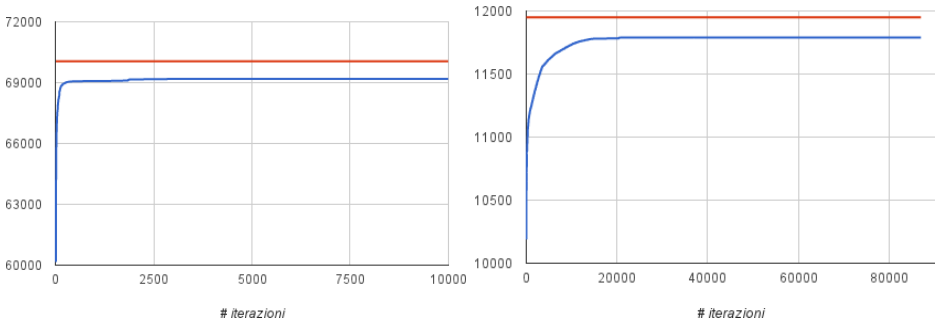

Figura 2.2: Miglioramento del lower bound con PHK per `gr137` e `fl417`.

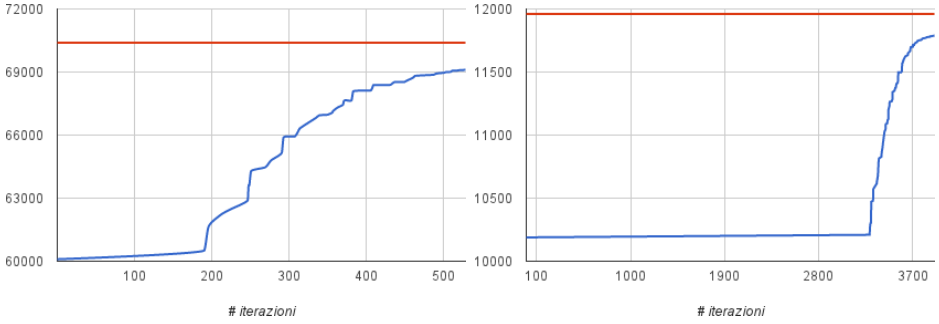

Figura 2.3: Miglioramento del lower bound con PVJ per `gr137` e `fl417`.

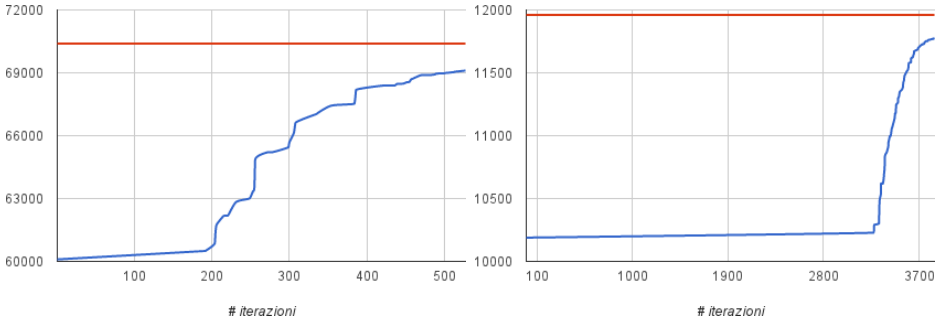

Figura 2.4: Miglioramento del lower bound con KVJ per `gr137` e `fl417`.

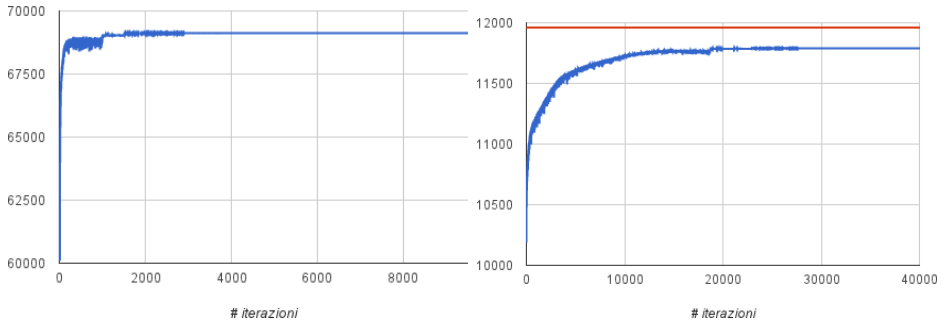

Figura 2.5: Andamento complessivo del lower bound con PHK per `gr137` e `fl417`.

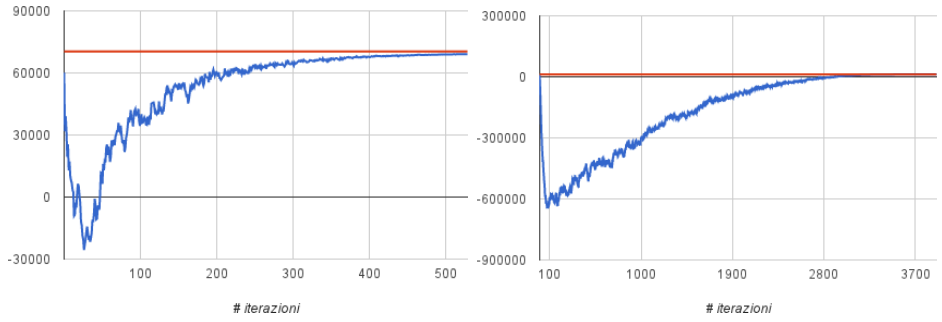

Figura 2.6: Andamento complessivo del lower bound con PVJ per `gr137` e `fl417`.

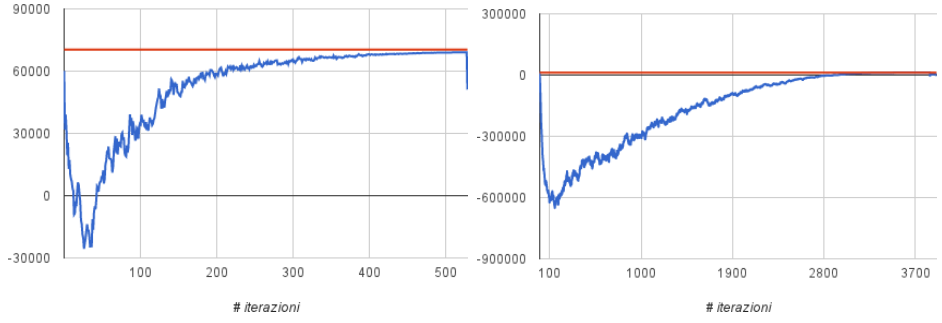

Figura 2.7: Andamento complessivo del lower bound con KVJ per `gr137` e `fl417`.

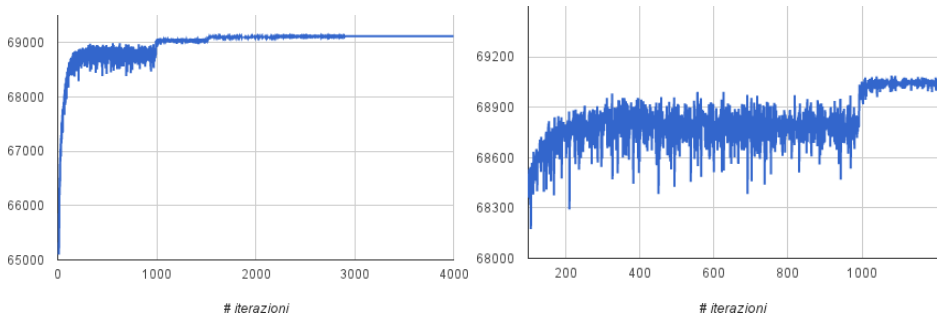

Figura 2.8: Dettagli dell'aggiornamento del bound per `gr137`.

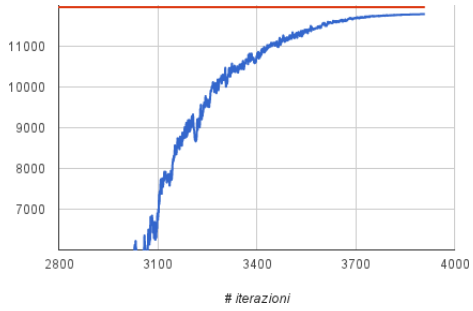

Figura 2.9: Dettaglio dell'aggiornamento del bound per `fl417`.

2.13 è posto inizialmente pari a $L(\pi^1)/n$, ed è successivamente aggiornato ogni qual volta l'algoritmo migliora il valore della funzione lagrangiana [1]. Come si vedrà anche più avanti, esiste per questa implementazione la possibilità di essere eseguita in una modalità leggermente differente, ottimizzata rispetto al caso in cui ci si trovi ad operare in un nodo interno dell'albero di ricerca del branch-and-bound. In questo caso, il numero massimo di iterazioni viene notevolmente ridotto, mentre il valore dello step di base viene calcolato in base al valore dei moltiplicatori corrispondenti alla migliore soluzione trovata dall'algoritmo quando eseguito nel nodo radice dell'albero di ricerca. Sempre con riferimento ad un contesto di branch-and-bound, un accorgimento particolarmente efficace è inoltre quello di interrompere l'esecuzione e uscire dal loop principale di iterazione non appena viene trovato un valore della funzione lagrangiana superiore all'incumbent.

Riportiamo in figura 2.9 il dettaglio delle ultime iterazioni di PVJ su `fl417`.

[1] Il valore scelto in [2] per t^1 è $L(\pi^1) * 0.01$, dove il fattore 0.01 è stato presumibilmente scelto in rapporto alla dimensione delle istanze risolte, che all'epoca difficilmente superava le 100 città.

2.3.3 Subgradiente KVJ

L'implementazione che ora descriveremo, a differenza delle precedenti, utilizza l'algoritmo di Kruskal per il calcolo di un albero ricoprente minimo. La complessità computazionale dell'algoritmo di Prim, quadratica nel numero di nodi dell'istanza, non permette di sfruttare appieno eventuali riduzioni del problema iniziale in termini di numero di lati. L'aggiunta di vincoli iniziali sui lati non è certo trasparente ad un branch-and-bound in cui l'1-albero viene calcolato utilizzando l'algoritmo di Prim, visto che il fissaggio dei lati riduce le possibili strade nell'albero di ricerca; ciò che però si vorrebbe ottenere è anche uno speed-up all'interno di ogni singolo nodo, come conseguenza diretta di una effettiva diminuzione della dimensione dell'istanza. Come indicato dai test effettuati, i vantaggi di questo approccio riescono a bilanciare e superare alcune evidenti limitazioni dell'algoritmo di Kruskal, a partire proprio dalla sfavorevole complessità computazionale nel caso di grafi completi o più in generale densi.

La rapidità dell'algoritmo è principalmente ostacolata dalla difficoltà di effettuare efficientemente le due seguenti operazioni:

1. ordinare i lati del grafo in base al loro costo
2. verificare che l'aggiunta di un lato alla soluzione sia consistente con i vincoli di aciclicità

Per quanto riguarda il punto 2, abbiamo affrontato il problema realizzando una struttura dati di tipo *union-find*, utilizzata ogni qualvolta l'algoritmo cerca di aggiungere un lato (i, j) alla soluzione. Le procedure $find(i)$ e $find(j)$ vengono chiamate per verificare se i e j appartengono o meno alla stessa componente connessa: in caso positivo il lato viene scartato dalla soluzione, altrimenti viene aggiunto e la procedura $union(i, j)$ aggiorna la struttura per tenere traccia della nuova componente connessa formatasi. Dal momento che la procedura *union* viene utilizzata al più $n - 2$ volte, mentre $find$ è utilizzato un numero di volte che è circa pari al numero di lati del grafo, sì è deciso di progettare la struttura in modo tale che $find$ avesse complessità $O(1)$. Per rendere inoltre il costo complessivo delle $n - 2$ operazioni *union* dell'ordine $O(n \log(n))$, l'unione di due componenti separate avviene aggiornando gli elementi della componente con minor numero di elementi. Naturalmente, una volta aggiunti gli $n - 2$ lati corrispondenti alla porzione mst dell'1-albero, non è necessario continuare a processare i rimanenti.

Per quanto riguarda invece la problematica evidenziata al punto 1, una soluzione è stata tentata affidando l'ordinamento dei lati alla funzione *qsort* della libreria standard di C. Come ci si poteva aspettare, la complessità computazionale di quicksort non ha permesso di ottenere significativi miglioramenti rispetto alle soluzioni Prim-based, anche in caso di grafi notevolmente ridotti. È stata invece scartata fin da subito la possibilità di un

ordinamento di tipo merge-sort, non essendo un algoritmo in-place. Si è deciso quindi di sfruttare l'interezza dei costi (nel caso di istanze TSPLIB) per migrare verso l'utilizzo di algoritmi di ordinamento non comparativi, in particolare counting-sort. I lati possono essere facilmente ordinati considerando come chiave il loro stesso costo, e la complessità computazionale di questa operazione è data da:

$$T_{cs} = 2N + K \tag{2.15}$$

dove N è il numero di lati da ordinare e K e il numero di possibili chiavi. Se c_{min} e c_{max} sono rispettivamente il costo minimo e il costo massimo tra i costi dei lati da ordinare, il numero di chiavi è dato da $K = c_{max} - c_{min} + 1$. Sebbene si possa osservare che per la maggior parte delle istanze TSPLIB il range di possibili costi non supera il numero di lati del grafo completo, per alcune istanze la differenza di costo tra i lati da ordinare potrebbe rallentare notevolmente l'ordinamento. Bisogna inoltre ricordare che i costi dei lati durante le varie iterazioni del subgradiente vengono modificati dai moltiplicatori lagrangiani, e il range dei costi potrebbe aumentare. Infine, il peso di K risulta certamente più rilevante quanto più il numero N di lati da ordinare diminuisce, ovvero proprio nei casi in cui vorremmo trarre beneficio dalla nostra implementazione Kruskal-based. Per aggirare questo problema si è reso necessario implementare un ulteriore algoritmo di ordinamento di tipo radix-sort, da utilizzare quando il numero di chiavi supera di una certa soglia il numero di lati da ordinare. La complessità computazionale di radix-sort è data da:

$$T_{rx} = 2Nd + db \tag{2.16}$$

dove $d = \log_b K$. Scegliendo come base $b = N$ si ottiene una complessità pari a $3Nd$, dove $d = \log_N K$.

Descriviamo brevemente le principali fasi attraversate ad ogni iterazione.

1. Viene costruito un 1-albero a partire da una lista di lati ordinati composta all'iterazione precedente.
2. Vengono calcolati i nuovi moltiplicatori: dal momento che intendiamo trattare solamente costi interi, i moltiplicatori vengno arrotondati all'intero più vicino prima di essere utilizzati per aggiornare i costi dei lati del problema lagrangiano. Il prezzo da pagare sarà una generale peggioramento nella qualità del lower bound ritornato.
3. Vengono aggiornati i costi dei lati a partire dai moltiplicatori correnti.
4. I lati vengono ordinati in base al loro costo.

L'algoritmo può essere reso ancora più efficiente se si considera che nel corso della sua evoluzione, tende a modificare con sempre minor frequenza i valori assunti dai moltiplicatori; molti di essi si assestano su pesi che manterranno invariati fino alla fine dell'esecuzione. Come conseguenza, il numero

di lati i cui costi vengono modificati tende generalmente a diminuire al crescere del numero di iterazioni (si vedano le figure in 2.10). L'idea è quella di estrarre dalla lista globale dei lati L, ereditata già ordinata dall'iterazione precedente, quei lati i cui costi sono stati modificati nell'iterazione corrente, formando una lista L_{ch} di lati modificati. I lati rimanenti formeranno invece una lista L_{nch} già ordinata. Ci possiamo quindi limitare l'applicazione degli algoritmi di ordinamento, il fattore computazionalmente più pesante di tutta la procedura, alla sola lista L_{ch}. Una nuova lista di lati ordinata utilizzabile dall'algoritmo di Kruskal all'iterazione successiva viene infine creata in tempo lineare effettuando un semplice merge tra L_{ch} e L_{nch}. Gli algoritmi di ordinamento utilizzati sono, come già detto, counting-sort e radix-sort, unitamente a quicksort per gestire eventuali fasi finali del subgradiente, in cui il numero di lati modificati diminuisce a tal punto rispetto al numero di chiavi utilizzate da rendere poco conveniente l'utilizzo dello stesso radix-sort rispetto a quicksort.

2.4 Branch-and-Bound

Nella sua implementazione di base il branch-and-bound (Land and Doig [10]) è una ricerca in profondità in cui ad ogni nodo viene selezionata una variabile della soluzione frazionaria, e relativamente ad essa viene effettuata una scelta, che porta alla generazione di due sottoalberi: in uno la variabile viene arrotondata all'intero inferiore, nell'altro viene invece arrotondata all'intero superiore. Questa operazione è detta *branching*. In generale, ad ogni nodo viene effettuata una decisione tale da partizionare il sottospazio di ricerca in sottoalberi mutuamente disgiunti, garantendo in questo modo sia la completezza della ricerca che l'unicità del cammino per raggiungere la soluzione. Ad ogni nodo, la soluzione generata viene inoltre confrontata con i bound alla soluzione ottima disponibili a quel punto della ricerca: nel caso si tratti di un problema di minimizzazione, se il costo è superiore o uguale all'upper bound (detto anche *incumbent*), il ramo di branching può essere rimosso e il sottoalbero associato scartato, dal momento che non potrà contenere una soluzione ammissibile di costo minore. Infine, sempre nel caso di problemi di minimizzazione, se ad un nodo viene generata una soluzione ammissibile di costo inferiore all'incumbent, quest'utlimo verrà aggiornato al costo della nuova soluzione.

Discutiamo di seguito alcuni aspetti della nostra implementazione, che fa uso degli algoritmi precedentemente presentati. Tutte le versioni proposte fanno uso della regola di branching proposta da Volgenant e Jonker in [16], e attraversano i rami preferendo nell'ordine i caso in cui i due lati selezionati e_1 ed e_2 vengono entrambi forzati, il caso in cui e_1 viene forzato ed e_2 viene vietato, infine il caso in cui e_1 viene vietato.

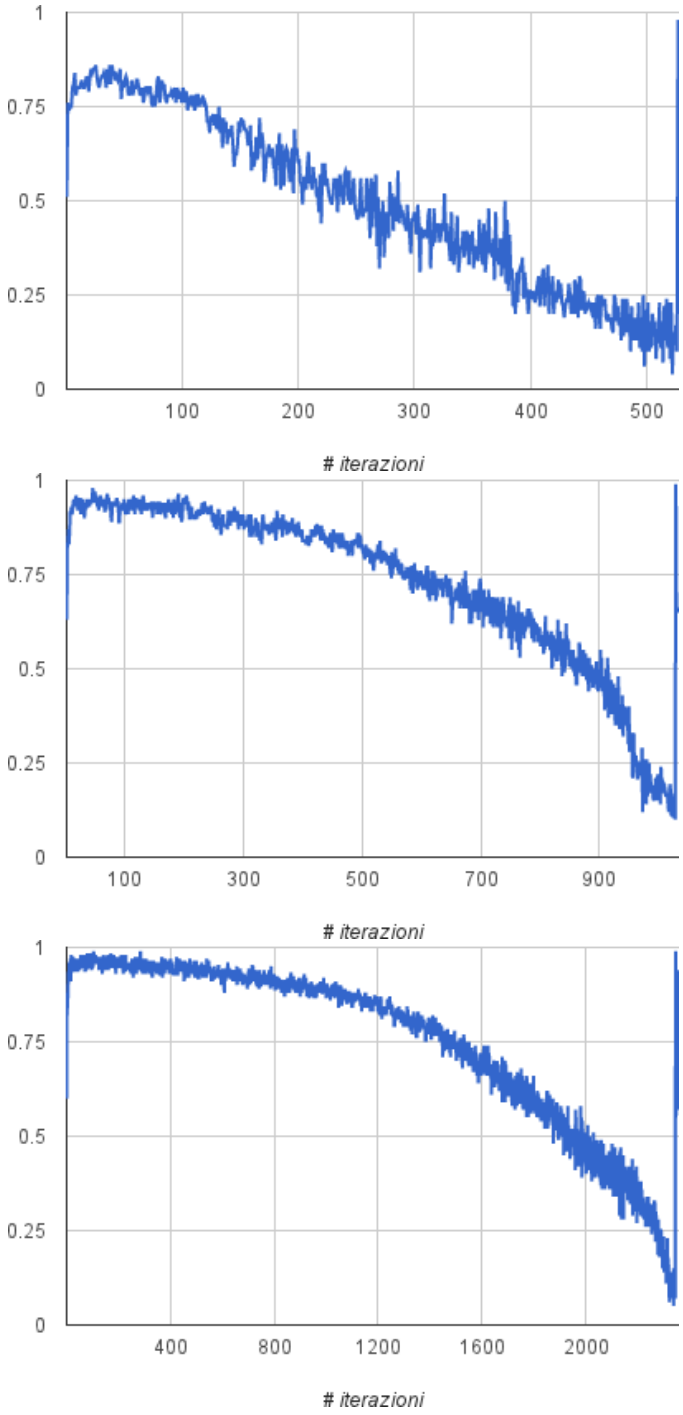

Figura 2.10: Andamento del numero di lati con costi modificati (frazione sul totale) per `gr137`, `gr202`, `lin318`.

2.4.1 B & B Prim-based

Una prima versione che abbiamo implementato utilizza PVJ per la computazione di un buon lower-bound ad ogni nodo. Nonostante PVJ riesca a garantire bound di qualità migliore rispetto a KVJ, la complessità quadratica nel numero dei nodi non permette incrementi nella velocità di esecuzione a seguito di riduzioni. Di questa versione, come della successiva, sono state provate alcune varianti, modificando alcuni parametri o inserendo o meno metodi di riduzione e propagazione.

2.4.2 B & B Kruskal-based

Una seconda versione utilizza invece KVJ, per sfruttare al massimo le riduzioni applicate al grafo di partenza e l'imposizione di vincoli durante la discesa nell'albero di ricerca. In questa implementazione, viene mantenuta una lista concatenata di lati non vincolati, aggiornata ogni volta prima di passare ad un nodo figlio, e viceversa al ritorno per ripristinare le condizioni inizialmente presenti. In entrambe le versioni, generalizzando quanto suggerito in [3], ogni nodo eredita i moltiplicatori lagrangiani associati alla migliore soluzione trovata dal nodo padre: tali moltiplicatori vengono utilizzati come punto di partenza per l'algoritmo del subgradiente, e a partire da essi viene calcolata l'ampiezza dello step iniziale.

2.4.3 Riduzione dei lati

Per rendere vantaggioso l'utilizzo di KVJ rispetto a PVJ, è necessaria l'applicazione al grafo iniziale di una buona riduzione. La riduzione implementata prevede la costruzione di un 1-albero minimo, il calcolo dei *costi marginali* associati a ciascun lato del grafo, e infine l'eliminazione di quei lati i cui costi ridotti superano una certa soglia, con la garanzia che tali lati certamente non appartengono alla soluzione ottima. L'osservazione che sta alla base del meccanismo di riduzione utilizzato, è la seguente: dato un 1-albero minimo T per il grafo G, un lato e può essere eliminato se una sua forzatura costringe l'1-albero minimo ad assumere un valore maggiore rispetto ad un upper bound alla soluzione ottima. Se $T(e)$ è l'1-albero minimo ottenuto forzando il lato e in T, si definisce come costo marginale di e il valore $\bar{c_e} = c(T(e)) - c(T)$, e il lato può essere rimosso senza escludere la soluzione ottima dalla regione ammissibile se:

$$c(T(G) + \bar{c_e} > UB \tag{2.17}$$

La correttezza di tale procedura è garantita ([3, 4]) anche nel caso in cui $T(G)$ sia un 1-albero ottenuto da un rilassamento lagrangiano del tipo proposto da Held e Karp. Nel nostro caso, una riduzione di questo tipo è applicata inizialmente, utilizzando il migliore 1-albero calcolato da PHK.

Abbiamo poi esplorato la possibilità di utilizzare la stessa riduzione anche all'interno dell'albero di ricerca, prestando attenzione a ristabilire i lati rimossi prima di risalire ad un livello superiore. Per evitare di appesantire troppo la computazione, la riduzione viene eseguita in un determinato nodo solamente quando viene rilevato un restringimento del gap tra lower e upper bound rispetto al gap registrato nel nodo padre.

2.4.4 Propagazione dei vincoli

Un'altra tecnica per cercare di ridurre lo spazio di ricerca è quella di aggiungere tutti quei vincoli che possono essere derivati a partire dalla presente disposizione dei vincoli sul grafo, oppure verificare la consistenza dei vincoli che dovranno essere aggiunti per scartare eventuali branch non ammissibili prima di iniziare la computazione nei nodi figli. Una prima osservazione per procedere in tal senso è la seguente: se ci sono due lati forzati incidenti in un vertice, è possibile vietare tutti i rimanenti lati incidenti su quello stesso vertice, dal momento che i vincoli di grado impongono che vi siano esattamente due lati incidenti in ogni vertice della soluzione. Analogamente, se ci sono $n - 3$ lati vietati incidenti in un vertice, è possibile forzare i rimanenti due lati.

L'imposizione di vincoli ai lati incidenti in un nodo con due lati forzati o $n - 3$ lati vietati modifica il numero di lati forzati e vietati incidenti nei nodi vicini, rendendo eventualmente possibile applicare su di essi lo stesso procedimento qualora le condizioni siano verificate. L'idea è quindi di procedere ricorsivamente propagando l'imposizione di vincoli ai nodi vicini. A partire da queste considerazioni abbiamo implementato una procedura che, prima di attraversare un nuovo ramo dell'albero di ricerca, impone e cerca di propagare i vincoli associati a quel branch, verificando infine la presenza di eventuali inconsistenze. Prima di passare al branch successivo o risalire l'albero di ricerca, viene effettuato un rollback per ristabilire i vincoli preesistenti.

2.5 Risultati computazionali

2.5.1 Subgradiente lagrangiano

In tabella 2.1 sono riportati i risultati ottenuti con le varie implementazioni del subgradiente lagrangiano. I risultati sono in percentuale rispetto all'ottimo, e i tempi sono espressi in secondi; sono evidenziati in grassetto i bound che corrispondono al costo del tour ottimo, segno che il subgradiente lagrangiano ha risolto l'istanza. Sono riportati, da sinistra a destra, i risultati di PHK, PVJ e KVJ. L'ultima colonna riporta il miglior lower bound ottenuto.

Si nota facilmente come PHK sia la soluzione che consente di ottenere i migliori lower bound sia Prim-Held-Karp, che tuttavia è anche la soluzione

	PHK		PVJ		KVJ		
Istanza	z	T[s]	z	T[s]	z	T[s]	z^*
a280	99.50	33.54	99.49	1.65	98.94	1.61	2566.00
ahk48	99.83	0.04	99.81	0.00	99.81	0.00	11 441.31
ali535	99.47	218.44	99.40	31.36	99.37	78.53	201 236.24
att48	99.77	1.30	99.69	0.01	99.70	0.00	9029.00
att532	95.58	193.52	95.57	25.09	95.44	66.58	27 419.14
berlin52	**100.00**	0.05	**100.00**	0.00	**100.00**	0.01	7542.00
bier127	99.28	1.61	99.27	0.10	99.27	0.16	95 331.00
ch130	99.44	1.93	99.40	0.12	99.27	0.10	6075.50
ch150	99.42	3.56	99.37	0.19	99.25	0.17	6490.12
d198	99.57	8.45	99.55	0.44	99.46	0.65	15 712.00
d493	99.50	153.44	99.49	16.53	99.34	40.99	34 828.49
d657	99.07	427.04	99.05	78.32	99.00	187.93	48 455.17
eil101	99.76	0.70	99.69	0.04	99.57	0.03	627.50
eil51	99.18	0.05	99.12	0.00	99.03	0.01	422.50
eil76	99.74	0.24	99.79	0.02	99.73	0.01	536.90
fl417	99.40	128.11	99.39	10.55	99.28	26.14	11 789.50
fri26	**100.00**	0.00	**100.00**	0.00	99.94	0.01	937.00
gil262	99.01	33.01	99.00	1.49	98.60	1.31	2354.50
gr137	98.95	1.91	98.93	0.11	98.93	0.19	69 120.25
gr17	**100.00**	0.00	95.70	0.00	95.37	0.00	2085.00
gr202	99.74	9.50	99.73	0.47	99.73	0.75	40 055.00
gr21	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	2707.00
gr229	99.05	14.61	99.03	0.74	99.03	1.51	133 317.02
gr24	**100.00**	0.00	99.96	0.00	99.76	0.00	1271.98
gr48	98.28	0.05	98.25	0.00	98.19	0.01	4959.00
gr666	99.37	349.28	99.36	72.04	99.35	200.54	292 493.34
gr96	98.84	0.48	98.84	0.03	98.83	0.04	54 570.50
hk48	99.83	0.04	99.81	0.00	99.81	0.00	11 441.31
kroA100	98.38	0.75	98.37	0.05	98.35	0.05	20 936.50
kroA150	99.15	3.53	99.14	0.19	99.11	0.26	26 299.00
kroA200	98.97	11.14	98.94	0.54	98.93	0.58	29 065.00
kroB100	98.61	0.73	98.61	0.05	98.60	0.06	21 834.00
kroB150	98.48	3.60	98.47	0.19	98.44	0.26	25 732.50
kroB200	99.08	11.01	99.07	0.54	99.05	0.55	29 165.00
kroC100	98.67	0.72	98.66	0.05	98.65	0.05	20 472.50
kroD100	99.28	0.73	99.27	0.05	99.26	0.05	21 141.50
kroE100	98.78	0.73	98.78	0.05	98.78	0.05	21 799.50
lin105	99.94	0.68	99.94	0.04	99.94	0.06	14 370.38
lin318	99.67	57.15	99.66	2.73	99.65	5.52	41 888.75
p654	96.87	318.87	99.83	66.61	99.46	137.72	34 582.78
pcb442	99.45	150.48	99.41	11.96	99.34	25.31	50 499.49
pr107	87.25	0.72	81.96	0.04	81.23	0.09	44 302.48
pr124	98.37	1.31	98.36	0.07	98.36	0.09	58 067.50
pr136	99.13	2.07	99.09	0.12	99.07	0.20	95 934.49
pr144	99.41	2.40	99.40	0.16	99.40	0.21	58 189.25
pr152	99.36	3.03	98.24	0.16	98.27	0.29	73 207.57
pr226	99.66	14.16	99.65	0.72	99.64	1.33	80 092.00
pr264	99.73	26.44	99.77	1.29	99.76	2.25	49 019.86
pr299	98.32	42.15	98.31	2.06	98.29	2.93	47 380.00
pr439	98.80	135.56	98.78	9.67	98.77	31.25	92 443.00
pr76	97.17	0.20	97.18	0.02	97.18	0.02	90 111.00
rat195	98.98	7.69	98.81	0.40	98.37	0.39	2299.25
rat575	99.28	262.04	99.27	40.17	98.68	75.13	6723.99
rat783	99.62	667.19	99.62	185.33	99.00	380.34	8772.74
rat783	99.62	667.19	99.62	185.33	99.00	380.34	8772.74
rat99	99.59	0.54	99.57	0.03	99.25	0.04	1206.00
rd100	99.87	0.72	99.81	0.04	99.76	0.04	7899.33
rd400	99.19	109.70	99.18	7.41	99.06	10.78	15 157.00
st70	99.41	0.17	99.37	0.01	99.06	0.02	671.00
swiss42	99.92	0.01	99.89	0.00	99.76	0.01	1272.00

	PHK		PVJ		KVJ		
Istanza	z	T[s]	z	T[s]	z	T[s]	z^*
ts225	91.28	13.96	91.27	0.70	91.26	1.04	115 605.00
tsp225	98.96	14.53	98.94	0.72	98.62	0.71	3878.25
u159	99.63	3.67	99.63	0.21	99.61	0.35	41 925.00
u574	99.48	268.70	99.47	42.42	99.42	92.58	36 714.00
u724	99.39	554.70	99.38	138.04	99.31	320.33	41 652.66
ulysses16	**100.00**	0.00	95.75	0.00	95.83	0.00	6859.00
ulysses22	**100.00**	0.00	98.24	0.00	98.28	0.00	7013.00

Tabella 2.2: Confronto dei diversi metodi per il subgradiente lagrangiano.

che impiega il maggior tempo a convergere. Sia PVJ che KVJ riportano lower bound leggermente inferiori (con PVJ appena inferiore come prestazioni rispetto a PHK, mentre KVJ spesso non arriva a pareggiare la qualità della soluzione ottenuta) che tempi decisamente ridotti (anche sotto questo aspetto la soluzione basata su Prim è migliore di quella basata su Kruskal).

Nei test effettuati in seguito si è scelto di adoperare PHK, in quanto un miglior bound consente un preprocessing più efficace in termini di archi eliminati.

2.5.2 Branch-and-bound

Riportiamo i risultati ottenuti per i test sul branch-and-bound. In tabella 2.3 sono riportati i test effettuati utilizzando nei nodi interni rispettivamente Prim-Volgenant-Jonker con numero di iterazioni pari a $n/4$ (BB-PVJ-1), Kruskal-Held-Karp con $n/4$ iterazioni (BB-KVJ-1), Prim-Volgenant-Jonker con n iterazioni (BB-PVJ-2) e Kruskal-Held-Karp con n iterazioni (BB-KVJ-2). In tutti i casi vengono effettuate la riduzione e la propagazione dei vincoli sia al nodo radice che a tutti i nodi interni.

Per ognuno di questi metodi sono riportati il massimo livello raggiunto nell'albero di ricerca, il numero di nodi attraversati e il tempo impiegato. Nelle prime colonne sono riportati nell'ordine il costo della soluzione ottima, l'upper bound di partenza (calcolato con RC+23opt seguito da un branch-and-bound con prefissaggio – si veda la sezione 4.1.3), il lower bound iniziale calcolato con PHK, e la percentuale di archi rimossi dal preprocessing al nodo radice.

Nella tabella 2.5, invece, sono riportati i test effettuati sul branch-and-bound impiegando nei nodi interni Kruskal-Volgenant-Jonker con riduzione solo al nodo radice, senza propagazione dei vincoli (BB-KVJ-3), Kruskal-Volgenant-Jonker con riduzione solo al nodo iniziale e propagazione ad ogni nodo (BB-KVJ-4), Kruskal-Volgenant-Jonker con riduzione e propagazione dei vincoli ad ogni nodo (BB-KVJ-5) e Kruskal-Volgenant-Jonker con riduzione e propagazione dei vincoli ad ogni nodo e regola di branching di Volgenant e Jonker (BB-KVJ-6).

Per alcune delle istanze più grandi sono riportati solo i valori ottenuti con BB-KVJ-6, ovvero la soluzione rivelatasi più efficace, nella speranza spesso vana di giungere alla loro risoluzione.

I test sono stati effettuati utilizzando un tempo limite di 3600 secondi, su una macchina con processore Intel Core i7 quad-core con hyperthreading a 2.66GHz e 8 GB di RAM. L'upper bound iniziale è ricavato a seguito dell'esecuzione di B& B con prefissaggio di vincoli, e nei casi considerati coincide con il valore della soluzione ottima. Le tabelle evidenziano quindi il tempo speso dal B& B per la certificazione dell'ottimo. La certificazione fallisce quando viene superato il tempo massimo. Si noti che nei tempi di esecuzione non è stato incluso il tempo speso dall'euristico per la ricerca

					BB-PVJ-1			BB-KVJ-1			BB-PVJ-2			BB-KVJ-2		
Istanza	z^*	UB	LB	AR[%]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]
att48	10628	10628	10 604	91.76	5	16	0.01	3	12	0.01	4	8	0.02	3	8	0.00
berlin52	7542	7542	7542	96.08	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
bier127	118282	118282	117 431	87.59	34	583267	3600.01	36	3777582	3600.01	27	23466	470.43	25	33809	92.54
burma14	3323	3323	3323	84.62	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
ch130	6110	6110	6076	93.56	22	6771	43.14	29	56854	45.80	26	5361	101.07	31	43425	99.47
ch150	6528	6528	6490	94.71	17	1605	16.74	23	10730	12.35	19	719	25.15	20	6582	21.09
dantzig42	699	699	697	90.48	3	5	0.00	4	7	0.00	3	5	0.01	3	6	0.00
eil101	629	629	628	96	5	41	0.22	18	1034	0.54	5	28	0.43	15	258	0.33
eil51	426	426	423	90.59	13	265	0.16	23	2560	0.36	13	263	0.34	14	216	0.09
eil76	538	538	537	95.09	15	62	0.14	14	919	0.32	15	62	0.35	6	25	0.03
fri26	937	937	937	92	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
gr137	69853	69853	69 104	89.54	18	3398	21.62	18	3367	3.15	15	1343	26.09	16	1882	4.92
gr17	2085	2085	2085	87.50	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
gr21	2707	2707	2707	90	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
gr24	1272	1272	1270	85.87	2	4	0.00	3	6	0.00	2	4	0.01	3	6	0.00
gr48	5046	5046	4959	82.18	12	495	0.23	16	1200	0.19	10	245	0.28	18	1059	0.34
gr96	55209	55209	54 571	88.18	16	752	1.97	14	1030	0.60	18	473	3.40	16	503	0.88
hk48	11461	11461	11 444	93.09	4	12	0.01	3	7	0.00	4	8	0.01	4	9	0.00
kroA100	21282	21282	20 937	87.09	21	4498	14.11	20	7276	4.33	20	4382	39.54	19	6429	9.72
kroA150	26524	26524	26 299	92.49	24	13369	133.40	27	31453	39.31	24	5858	184.97	28	20658	72.57
kroB100	22141	22141	21 834	86.22	20	2784	9.25	19	4113	2.50	19	2217	22.27	18	2089	3.51
kroB150	26130	26130	25 733	86.10	36	379961	3600.01	40	969462	1158.50	34	124041	3600.04	39	792119	2499.05
kroC100	20749	20749	20 473	88.34	15	534	1.61	14	1154	0.68	16	735	6.28	16	1326	1.76
kroD100	21294	21294	21 142	91.92	11	73	0.27	11	121	0.08	6	19	0.24	12	149	0.24
kroE100	22068	22068	21 800	88.57	23	9148	29.42	27	18513	10.24	21	3043	29.07	21	4793	6.96
lin105	14379	14379	14 370	96.54	2	2	0.06	2	4	0.01	2	2	0.06	2	4	0.01
pr107	44303	44303	43 434	67.47	52	756475	3600.01	50	1706228	3600.01	15	401	5.60	11	400	2.62
pr124	59030	59030	58 068	84.21	19	1156	5.71	18	1426	1.38	16	667	9.47	17	898	2.42
pr136	96772	96772	95 935	90.94	39	262308	1800.89	37	267236	284.37	32	72740	1556.00	36	94345	280.15
pr144	58537	58537	58 188	89.71	25	1520	12.35	14	235	0.35	14	281	7.31	13	108	0.51
pr226	80369	80369	80 092	94.06	44	29915	799.22	45	138602	318.40	32	16284	1524.05	31	48247	311.15
pr264	49135	49135	48 991	94.85	42	68186	3600.01	47	497384	3600.01	14	144	25.41	13	128	2.51
pr76	108159	108159	105 120	68.56	31	187548	246.89	30	248822	90.01	31	169070	586.97	29	156224	132.36
rat99	1211	1211	1206	95.32	9	37	0.15	25	966	0.46	12	46	0.42	10	57	0.08
rd100	7910	7910	7900	95.98	13	51	0.19	10	45	0.03	4	8	0.13	5	23	0.04

					BB-PVJ-1			BB-KVJ-1			BB-PVJ-2			BB-KVJ-2		
Istanza	z^*	UB	LB	AR[%]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]
st70	675	675	671	91.84	14	490	0.68	32	5337	1.50	5	17	0.09	10	86	0.08
swiss42	1273	1273	1272	91.99	3	8	0.00	3	10	0.00	3	8	0.01	3	10	0.00
u159	42080	42080	41 925	96.22	14	738	7.03	14	774	0.91	12	396	12.56	16	781	2.24
ulysses16	6859	6859	6859	86.67	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
ulysses22	7013	7013	7012	84.42	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00

Tabella 2.4: Risultati del branch-and-bound con PVJ e KVJ (1/2)

					BB-KVJ-3			BB-KVJ-4			BB-KVJ-5			BB-KVJ-6		
Istanza	z^*	UB	LB	AR[%]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]
a280														58	387893	3600.01
att48	10628	10628	10 604	91.76	5	11	0.01	4	13	0.01	3	8	0.00	3	8	0.00
berlin52	7542	7542	7542	96.08	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
bier127	118282	118282	117 431	87.59	31	55092	424.40	27	43340	305.07	25	33809	92.54	30	28846	82.18
burma14	3323	3323	3323	84.62	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
ch130	6110	6110	6076	93.56	31	52543	211.23	29	56556	215.54	31	43425	99.47	31	24303	56.88
ch150	6528	6528	6490	94.71	24	8368	42.71	22	8430	41.69	20	6582	21.09	24	4881	16.13
d198	15780	15863	15 712											44	407066	3600.01
dantzig42	699	699	697	90.48	3	7	0.00	3	6	0.00	3	6	0.00	3	6	0.00
eil101	629	629	628	96.00	19	228	0.36	19	413	0.48	15	258	0.33	11	173	0.25
eil51	426	426	423	90.59	17	408	0.17	15	263	0.11	14	216	0.09	17	183	0.07
eil76	538	538	537	95.09	4	16	0.02	5	19	0.03	6	25	0.03	7	28	0.03
fri26	937	937	937	92.00	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
gr137	69853	69853	69 104	89.54	18	2336	13.46	17	2743	14.67	16	1882	4.92	18	1958	5.10
gr17	2085	2085	2085	87.50	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
gr202	40160	41069	40 055											19	2763	19.15
gr21	2707	2707	2707	90.00	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
gr24	1272	1272	1270	85.87	3	7	0.00	3	7	0.00	3	6	0.00	3	6	0.01
gr48	5046	5046	4959	82.18	17	1204	0.58	16	1264	0.55	18	1059	0.34	14	556	0.19
gr96	55209	55209	54 571	88.18	16	693	1.99	16	671	1.88	16	503	0.88	15	404	0.68
hk48	11461	11461	11 444	93.09	3	6	0.00	4	10	0.00	4	9	0.00	4	9	0.00
kroA100	21282	21282	20 937	87.09	20	8542	25.20	20	8091	22.14	19	6429	9.72	18	3276	5.20
kroA150	26524	26524	26 299	92.49	29	21177	150.10	29	28110	186.48	28	20658	72.57	25	15121	53.86
kroA200	29368	29575	29 065											41	477430	3600.01
kroB100	22141	22141	21 834	86.22	16	2361	8.19	18	2661	8.95	18	2089	3.51	18	1271	2.30
kroB150	26130	26130	25 733	86.10	39	419667	3600.01	38	446871	3600.01	39	792119	2499.05	41	498885	1623.01
kroB200	29437	30035	29 165											34	90148	742.00
kroC100	20749	20749	20 473	88.34	17	1553	3.64	15	1760	3.90	16	1326	1.76	16	1216	1.67
kroD100	21294	21294	21 142	91.92	14	210	0.44	11	183	0.39	12	149	0.24	10	86	0.13
kroE100	22068	22068	21 800	88.57	26	9154	23.69	21	7024	16.65	21	4793	6.96	21	4402	6.55
lin105	14379	14379	14 370	96.54	2	4	0.01	2	4	0.00	2	4	0.01	2	4	0.01
pr107	44303	44303	43 434	67.47	49	81729	1135.39	31	56785	620.00	11	400	2.62	12	312	2.22
pr124	59030	59030	58 068	84.21	15	945	6.77	19	1176	7.87	17	898	2.42	18	890	2.39
pr136	96772	96772	95 935	90.94	36	152514	1166.97	32	129881	933.14	36	94345	280.15	33	103237	296.93
pr144	58537	58537	58 188	89.71	13	224	2.17	11	232	2.20	13	108	0.51	16	112	0.52

					BB-KVJ-3			BB-KVJ-4			BB-KVJ-5			BB-KVJ-6		
Istanza	z^*	UB	LB	AR[%]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]	#L	#N	T[s]
pr152	73682	74021	73 209											53	419315	3600.01
pr226	80369	80369	80 092	94.06	63	175451	3600.02	54	250515	3600.01	31	48247	311.15	31	53452	337.83
pr264	49135	49135	48 991	94.85	40	6552	208.34	22	1469	40.03	13	128	2.51	12	120	2.41
pr76	108159	108159	105 120	68.56	32	225972	634.88	29	206724	518.44	29	156224	132.36	31	225842	183.53
rat99	1211	1211	1206	95.32	14	90	0.14	10	67	0.10	10	57	0.08	10	57	0.08
rat195	2323	2371	2300											55	255304	1464.57
rd100	7910	7910	7900	95.98	5	29	0.05	7	29	0.05	5	23	0.04	5	24	0.04
st70	675	675	671	91.84	14	113	0.12	11	94	0.09	10	86	0.08	9	47	0.05
swiss42	1273	1273	1272	91.99	3	10	0.01	3	10	0.00	3	10	0.00	3	10	0.00
u159	42080	42080	41 925	96.22	15	401	1.88	15	613	2.64	16	781	2.24	18	677	2.07
ulysses16	6859	6859	6859	86.67	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00
ulysses22	7013	7013	7012	84.42	1	1	0.00	1	1	0.00	1	1	0.00	1	1	0.00

Tabella 2.6: Risultati del branch-and-bound con KVJ (2/2)

dell'upper bound, che in molti casi non supera comunque qualche decina di secondi.

Considerando la prima tabella, possiamo notare come la scelta di una soluzione Kruskal-based porti a dei risultati nettamente migliori rispetto a quelli ottenuti con la versione del B & B basata su Prim. Si confronti a questo proposito le colonne BB-PVJ-1 e BB-KVJ-1. La riduzione iniziale e, in parte, la riduzione ad ogni nodo unita alla propagazione dei vincoli rendono KVJ sensibilmente più veloce rispetto a PVJ. Un'ulteriore miglioramento si ha aumentando il numero di iterazioni compiute dall'algoritmo del subgradiente nei nodi interni dell'albero di ricerca, sostituendo il valore $n/4$ proposto da Volgenant e Jonker con n. Per questo si confrontino BB-PVJ-2 e BB-KVJ-2 rispettivamente con BB-PVJ-1 e BB-KVJ-1. Sono stati effettuati dei test utilizzando valori maggiori, rispettivamente proporzionali a $n\log(n)$ e a n^2, ottenendo però un generale peggioramento delle prestazioni.

Nella seconda tabella abbiamo messo in evidenza il miglioramento dovuto all'introduzione di tecniche quali la riduzione ad ogni nodo e la propagazione dei vincoli. Sono riportati infatti i risultati ottenuti utilizzando come algoritmo di base un B & B che utilizza KVJ, con n iterazioni(la versione che si è rivelata più promettente in base ai risultati della prima tabella), con riduzione al solo nodo radice e privato del meccanismo di propagazione dei vincoli.

Si può osservare come l'aggiunta dapprima della propagazione, e in seguito della riduzione ad ogni nodo, comporti un sostanziale miglioramento delle prestazioni, confermando l'idea che sia in generale preferibile tentare di interrompere il prima possibile la discesa nell'albero di ricerca, riducendo il numero di potenziali strade con tecniche di riduzione e propagazione, anche al costo di appesantire leggermente l'esecuzione ad ogni nodo.

Infine, osservando l'ultima colonna, un lieve miglioramento è stato registrato implementando le regole di selezione del vertice su cui effettuare il branching, e di selezione dei lati su cui imporre i vincoli, come suggerito in [16].

2.6 Commenti

Il rilassamento basato su 1-albero è una tecnica molto semplice per determinare un lower bound sul costo della soluzione ottima di un'istanza del TSP. Tuttavia, questo bound è generalmente piuttosto scarso, data la semplicità della tecnica.

Il subgradiente lagrangiano consente di determinare invece dei lower bound decisamente migliori, spesso (tranne istanze molto particolari come `ts225`) superiori al 99% della soluzione ottima. Per istanze semplici, addirittura il subgradiente è in grado di determinare la soluzione ottima, o un

bound il cui scostamento dell'ottimo è inferiore a 1. La qualità del risultato del subgradiente lagrangiano, associata ad un buon upper bound, consente di scartare a priori un'elevata quantità di archi in fase di preprocessing, rivelandosi quindi cruciale per una veloce risoluzione delle istanze. In base ai test effettuati, possiamo affermare che la qualità del lower bound è forse più importante di quella dell'upper bound.

Abbiamo quindi implementato una strategia branch-and-bound basata sul subgradiente lagrangiano, che consente di attaccare istanze fino a oltre 200 nodi. Le prestazioni del branch-and-bound sono fortemente dipendenti dalle prestazioni del subgradiente lagrangiano, anche se, come si è visto, si possono ottenere ulteriori margini di miglioramento tramite tecniche di filtraggio. La capacità del branch-and-bound basato su KVJ di beneficiare a pieno dei vantaggi della riduzione hanno suggerito la combinazione del branch-and-bound con altre tecniche come euristico, come descriveremo in sezione 4.1.3.

Capitolo 3

Risolutori basati sul modello MIP

La seconda via studiata per risolvere in maniera esatta il TSP è la formulazione del problema come problema di programmazione lineare mista intera, in cui alcune variabili sono vincolate all'interezza, mentre su altre tale vincolo è rilassato e possono assumere valori continui. Il modello viene poi risolto con il solver commerciale CPLEX, disponibile gratuitamente per uso accademico e di studio. Esistono vari MIP solver commerciali e open source, che si basano sostanzialmente su un branch-and-cut eseguito sul rilassamento continuo del problema, in cui ad ogni nodo un pool di tagli ed euristici viene applicato sulla soluzione valutata.

3.1 Modello MIP del TSP

Il modello MIP del TSP è riportato nell'introduzione della presente relazione. Il modello proposto è computazionalmente intrattabile; la causa principale di questa situazione risiede nel numero dei vincoli di aciclicità. Possiamo quindi rilassare il problema eliminando i SECs, e risolvere il problema così modificato. In genere, una soluzione al problema rilassato conterrà dei sottocicli; in tal caso possiamo individuarli, vietarli mediante appositi vincoli aggiunti come cutting planes e risolvere il nuovo problema. L'idea alla base di tale procedimento è che solo una piccola parte dei SECs sono effettivamente necessari per la corretta risoluzione del modello, mentre la maggior parte di loro non verrà mai violata da soluzioni di sufficiente qualità, pertanto la loro inclusione risulterebbe solo in un appesantimento del modello MIP.

3.1.1 Preprocessing

Il subgradiente lagrangiano fornisce un lower bound rispetto alla soluzione ottima, mentre varie tecniche euristiche (cap. 4) permettono di calcolare una soluzione ammissibile, quindi un upper bound al valore dell'ottimo. Possiamo sfruttare queste informazioni per "aiutare" la risoluzione del modello, imponendo al risolutore di escludere dal calcolo le variabili corrispondenti agli archi che possiamo determinare a priori non saranno presenti nella soluzione ottima. Questo procedimento si può effettuare nella maniera descritta in sezione 2.4.3.

Un altro intervento preliminare che possiamo compiere è fornire al solver una soluzione intera ammissibile da cui partire per un "raffinamento" verso la soluzione ottima. Anche in questo caso, la soluzione iniziale si può calcolare con un algoritmo euristico.

3.2 Risoluzione iterativa

Il primo procedimento di risoluzione analizzato consiste nell'applicare in maniera diretta la procedura descritta nella sezione 3.1. L'algoritmo itera i seguenti passi:

1. risoluzione del modello rilassato per eliminazione dei vincoli SEC;
2. ricerca di sottocicli nella soluzione ottenuta;
   (a) se presenti, aggiunta dei relativi vincoli al modello e ritorno al passo 1;
   (b) altrimenti, è stato individuato il ciclo hamiltoniano di costo minimo, e l'algoritmo termina.

Questa procedura implementa un metodo duale, che nel cammino verso l'ottimo globale individua soluzioni di costo migliore dell'ottimo, ma non ammissibili in quanto contenenti sottocicli. La prima soluzione composta di un solo ciclo sarà quindi la soluzione ottima. Tale procedimento comporta che, ad ogni iterazione, al passo 1 venga risolto da zero un modello corrispondente al modello dell'iterazione precedente, con l'aggiunta dei vincoli di eliminazione di sottociclo violati dalla soluzione ottima dell'iterazione precedente. Di conseguenza, iterazione dopo iterazione il modello sarà sempre più computazionalmente pesante.

Per velocizzare le operazioni, i solver forniscono funzionalità di *presolving*, che, prima del nodo radice del branch-and-cut interno, analizzano il modello e tentano di semplificarlo, identificando e rimuovendo variabili e vincoli ridondanti o inefficaci.

La ricerca di sottocicli viene effettuata individuando le componenti connesse nella soluzione, e inserendo un vincolo 2.4 nel modello per ciascuna di esse.

3.3 Risoluzione con callback "alla Miliotis"

Un secondo metodo di risoluzione, la cui implementazione in CPLEX è resa possibile dal meccanismo delle callback, è quello proposto da Miliotis [12], che valuta l'incumbent nel corso della risoluzione, invece che al termine della procedura come fa il metodo precedente. In particolare, non appena ad un nodo viene individuata una soluzione intera, su di essa viene immediatamente valutata la presenza o meno di sottocicli, in maniera identica a quanto fatto con il metodo iterativo. Se presenti, essi vengono aggiunti come tagli, altrimenti l'esecuzione può terminare, avendo trovato l'ottimo globale. Tale procedura segue un metodo primale, ovvero che attraversa una serie di soluzioni ammissibili subottime fino a raggiungere la soluzione ottima.

Il meccanismo delle callback è un paradigma di implementazione che prevede l'esecuzione di un metodo fornito dallo sviluppatore al verificarsi di un particolare evento, per gestire la situazione nella maniera più appropriata. Nel nostro caso, CPLEX fornisce callbacks per eventi diversi che possono occorrere durante la risoluzione di un modello, o per accedere ad alcune informazioni sullo stato della risoluzione (*informative callbacks*). Per motivi di segretezza commerciale alcune funzionalità proprietarie di CPLEX vengono tuttavia disabilitate con l'uso delle callback. Le callback che permettono di analizzare l'incumbent sono dette in CPLEX *lazy constraints*, perché invocate solo quando è presente una soluzione, al contrario delle callback di tipo *user cut* che vengono valutate ad ogni nodo dell'albero di branching.

3.4 Risoluzione con callback sulla soluzione frazionaria

Abbiamo menzionato la possibilità di invocare delle callback ad ogni nodo dell'albero decisionale sulla soluzione frazionaria corrente, valutando l'eventuale violazione dei vincoli di eliminazione dei sottocicli con le callback *user cut*.

In generale, una soluzione frazionaria avrà conterrà nodi su cui incidono un numero di archi maggiore di 2, ma pesati in maniera tale da rispettare comunque il vincolo di grado. Il problema da risolvere in una soluzione così composta è un problema di flusso massimo su archi di capacità 1 (essendo le variabili degli archi binarie), in cui è necessario imporre che per ogni insieme di nodi S vi siano almeno due archi che collegano S all'insieme complementare $V \setminus S$. L'implementazione di tale procedimento è stata invece effettuata basandoci sul problema duale del max-flow, ovvero la determinazione del taglio minimo nel grafo, usando le funzioni per il calcolo del min cut implementate nel software Concorde. Tali metodi richiedono la connessione del grafo. Se tale condizione è verificata, il metodo `CCmincut` ritorna l'insieme di nodi S costituenti un taglio minimo nel grafo così partizionato in S e

$V \setminus S$; in questo caso, si impone il vincolo

$$\sum_{e \in \delta(S)} x_e \geq 2, \tag{3.1}$$

per rendere connesso il sottografo S con il resto dei nodi. Se invece il grafo con archi frazionari non è connesso, si aggiungono vincoli mirati a spezzare le componenti connesse e rendere il grafo connesso.

Poiché le user cut callbacks vengono invocate ad ogni nodo dell'albero decisionale, esse possono rallentare enormemente l'esecuzione, rendendo impraticabile la risoluzione di problemi di medio-grandi dimensioni. Una possibile strategia è quella di applicare la ricerca dei vincoli violati solo nei nodi a bassa profondità, o su una frazione dei nodi scelta con qualche criterio (ad es: in una certa percentuale di nodi, scelti a caso).

3.5 Risultati computazionali

I test riportati sono stati effettuati su una macchina con processore Intel Core i7 quad-core con hyperthreading a 2,40GHz e 4 GB di RAM. Nelle tabella 3.1 mostriamo i risultati ottenuti con i metodi esatti basati su CPLEX, imponendo un tempo limite di un'ora per ogni soluzione. I valori mancanti indicano che non è stata individuata alcuna soluzione ammissibile nei tempi dati, o che l'esecuzione è stata abortita anticipatamente (soprattutto per eccessivo uso di memoria, nel caso delle user cut callback). I tempi talvolta eccedono i 3600 secondi, a causa dell'implementazione interna a CPLEX del controllo sul time limit. Le user cut callbacks vengono individuate sul 5% dei nodi dell'albero di branching scelti in maniera casuale.

Si può facilmente notare come nella maggior parte dei casi le soluzioni con callback siano decisamente più performanti rispetto alla soluzione iterativa, con alcune significative eccezioni. Vi sono istanze di taglia elevata che non sono state risolte nei tempi dati; in un caso non è stato nemmeno possibile trovare una soluzione ammissibile, sintomo di come sia consigliabile fornire una soluzione di partenza al solver. In altri casi l'implementazione basata su callbacks ha raggiunto la soluzione ottima, non riuscendo a certificarla nei tempi dati, situazione che con la versione iterativa non si può verificare.

La risoluzione con callback user cut sulla soluzione frazionaria delle istanze più grandi è stata abortita quando l'occupazione di memoria diventava eccessiva.

Mostriamo inoltre due immagini a titolo di esempio su come procede l'aggiornamento dell'incumbent nella versione iterativa (figura 3.1) e con le callback (figura 3.2).

Nella versione iterativa il costo della soluzione incumbent è sempre un lower bound della soluzione ottima, procedendo attraverso soluzioni di costo minore dell'ottimo ma inammissibili, fino a raggiungere la miglior soluzione

			Iterativo		Lazy constraints cbs.		User cut cbs.	
Istanza	UB	LB	z^*	T[s]	z^*	T[s]	z^*	T[s]
a280	2627	2566	2579	30.28	2579	2.82	2579	2.14
ali535	208383	201237	201467	5783.06	202368	3927.78		
att48	10628	10603	10628	0.06	10628	0.03	10628	0.03
att532	28743	27420	27684	3662.56	27706	3190.86		
berlin52	7542	7542	7542	0.00	7542	0.01	7542	0.00
bier127	119003	117431	118282	1.89	118282	0.44	118282	0.63
ch130	6178	6076	6110	0.91	6110	0.27	6110	0.45
ch150	6548	6491	6528	3.85	6528	0.90	6528	1.13
d198	15863	15712	15780	136.32	15780	6.19	15780	2.82
d493	35987	34829	34957	3609.99	35004	3872.03		
d657	50330	48456	48887	4117.63		3600.75		
eil101	634	628	629	0.31	629	0.11	629	0.20
eil51	427	423	426	0.12	426	0.03	426	0.03
eil76	543	537	538	0.05	538	0.02	538	0.01
fl417	11961	11790	11861	1644.89	11869	3657.51		
fri26	937	937	937	0.01	937	0.00	937	0.00
gil262	2438	2355	2378	88.63	2378	118.71	2378	90.31
gr137	70496	69121	69853	5.12	69853	0.60	69853	0.31
gr17	2085	2085	2085	0.01	2085	0.00	2805	0.00
gr202	41069	40055	40160	21.58	40160	2.22	40160	1.78
gr21	2707	2707	2707	0.00	2707	0.00	2707	0.00
gr229	137652	133318	134602	175.39	134602	57.85	134602	32.97
gr24	1272	1272	1272	0.01	1272	0.01	1272	0.00
gr48	5046	4959	5046	0.47	5046	0.09	5046	0.10
gr666	304907	292494	294358	3444.76	294358	3852.26		
gr96	55589	54571	55210	2.27	55210	0.33	55210	0.28
hk48	11461	11442	11461	0.01	11461	0.01	11461	0.00
kroA100	21282	20937	21282	1.44	21282	0.27	21282	0.22
kroA150	26815	26299	26524	16.33	26524	3.05	26524	3.57
kroA200	29575	29065	29368	75.04	29368	105.74	29368	133.07
kroB100	22278	21834	22141	3.28	22141	0.52	22141	0.57
kroB150	26445	25733	26130	74.43	26130	41.63	26130	45.22
kroB200	30035	29165	29437	33.57	29437	12.90	29437	3.89
kroC100	20769	20473	20749	1.08	20749	0.27	20749	0.27
kroD100	21404	21142	21294	0.98	21294	0.20	21294	0.30

			Iterativo		Lazy constraints cbs.		User cut cbs.	
Istanza	UB	LB	z^*	T[s]	z^*	T[s]	z^*	T[s]
kroE100	22100	21800	22068	2.43	22068	0.73	22068	0.62
lin105	14379	14371	14379	0.42	14379	0.25	14379	0.40
lin318	43200	41889	42029	213.63	42029	31.10	42029	84.01
p654	34905	33567				1261.05		
pcb442	53051	50500	50778	1145.56	50778	44.30	50778	37.54
pr107	44676	43681	44303	0.13	44303	0.02	44303	0.02
pr124	59087	58068	59030	16.63	59030	15.36	59030	0.50
pr136	97644	95935	96772	4.65	96772	0.31	96772	0.50
pr144	58571	58158	58537	11.86	58537	2.66	58537	0.89
pr152	74021	73209	73682	1.70	73682	1.91	73682	0.94
pr226	80590	80092	80369	2062.50	80369	78.51	80369	3.07
pr264	49636	49010			49135	65.05	49135	1.06
pr299	48816	47380	48191	628.81	48191	4013.15	48231	
pr439	108425	105929	106978	3683.16	107250	2011.72		
pr76	108274	105120	108159	12.21	108159	12.32	108159	15.35
rat195	2371	2300	2323	114.32	2323	26.84	2323	9.10
rat575	7105	6724	6773	3224.39	6773	3021.12		
rat783	9186	8773	8806	3333.15	8806			
rat99	1213	1206	1211	0.54	1211	0.06	1211	0.06
rd100	7913	7900	7910	0.46	7910	0.35	7910	0.18
rd400	15666	15157	15281	841.03	15282		15302	
st70	675	671	675	0.09	675	0.06	675	0.07
swiss42	1273	1272	1273	0.01	1273	0.00	1273	0.00
ts225	126809	115605						
tsp225	4047	3879	3916	165.70	3916	36.96	3916	35.26
u159	42624	41925	42080	2.16	42080	1.15	42080	0.41
u574	38334	36714	36905	1230.89	36907	3741.80		
u724	43860	41653	41821	3687.35	42003	3762.89		
ulysses16	6859	6859	6859	0.00	6859	0.01	6859	0.01
ulysses22	7013	7013	7013	0.02	7013	0.00	7013	0.00

Tabella 3.2: Risultati della risoluzione con CPLEX iterativo, con lazy constraint callbacks e user cut callbacks.

ammissibile, che sarà la soluzione ottima; nel metodo che fa uso di callback, invece, la risoluzione attraversa soluzioni ammissibili ma subottime, fino a giungere alla soluzione ottima, che dovrà essere certificata.

Vediamo chiaramente come in entrambe le versioni il solver all'inizio aggiorni la migliore soluzione molto velocemente, per poi rallentare vistosamente quando si avvicina all'ottimo globale, fatto poco sorprendente essendo una manifestazione dello stato di problema NP-completo del TSP. Il MIP solver deve infatti attraversare, al caso peggiore, l'intero albero di branching.

In entrambi i casi osserviamo un drastico miglioramento nelle prime iterazioni, che in seguito rallenta vistosamente. Per quanto riguarda `att532` l'ottimo viene trovato (all'incirca) all'iterazione 56000, mentre l'esecuzione termina con la certificazione dopo oltre 120000 iterazioni. L'ottimo di `rat575` invece viene trovato all'ultima iterazione prima che CPLEX venga terminato per aver ecceduto il time limit, senza quindi poter essere certificato; anche in questo caso, comunque, il trend dell'aggiornamento ricalca quello di `att532`, così come, in genere, di tutte le altre istanze.

3.6 Commenti

La risoluzione con solver MIP permette di attaccare problemi che altrimenti non sarebbero risolvibili con un branch-and-bound classico. La risoluzione con metodo iterativo scandisce lo spazio analizzando le soluzioni in ordine crescente di costo, di modo che la prima soluzione intera senza sottocicli è la soluzione ottima. Soluzioni intere incontrate nella risoluzione precedentemente all'ottimo includono sottocicli, e vanno perciò vietate mediante cutting planes. La risoluzione mediante callback invece procede in maniera primale, individuando soluzioni ammissibili di costo sempre minore, fino a raggiungere la soluzione ottima. Questo fornisce alla risoluzione con callback un comportamento *anytime*, avendo sempre una soluzione incumbent sin dalla prima soluzione feasible trovata. Nel tempo limite di un'ora assegnato ad ogni istanza, non è sempre stato possibile individuare la soluzione ottima; tuttavia, il metodo con le callbacks ha permesso di individuare soluzioni di costo di poco superiore all'ottimo, fatto non possibile con il metodo iterativo. Come abbiamo visto dai grafici in figura 3.2, questo non è comunque una garanzia dell'essere nei pressi della soluzione ottima, ovvero non è possibile stimare il tempo o il numero di iterazioni rimanenti prima di individuare il tour ottimo.

Dal punto di vista delle prestazioni, la risoluzione con callback è più veloce, al costo di una maggiore occupazione di memoria. In particolare, l'uso delle callback sulla soluzione frazionaria aumenta di molto l'occupazione di RAM, essendo invocate ad ogni nodo dell'albero di branching.

Come risolutore di default, in base ai risultati ottenuti con i test precedentemente riportati, la soluzione consigliabile è quella basata sulle callback

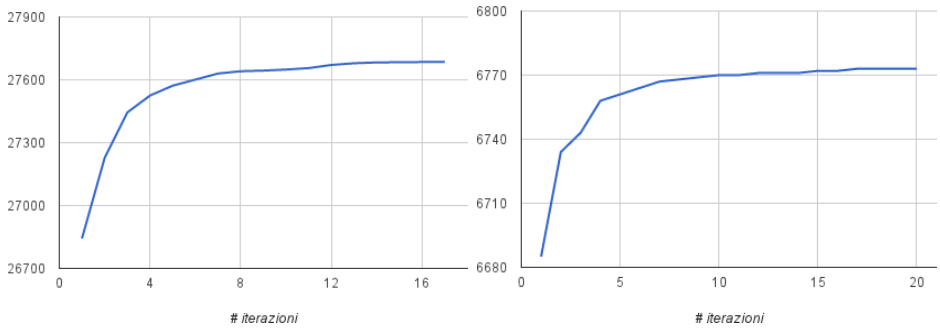

Figura 3.1: Aggiornamento dell'incumbent di `att532` e `rat575` nella risoluzione iterativa.

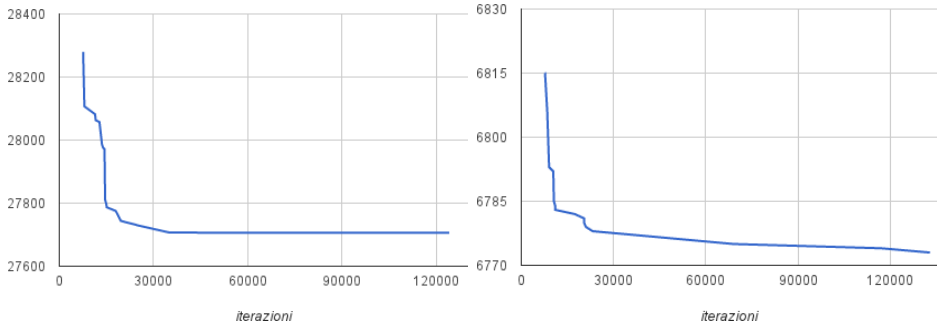

Figura 3.2: Aggiornamento dell'incumbent di `att532` e `rat575` nella risoluzione con callback.

di tipo *lazy constraints*, essendo quella che rende il miglior compromesso tra prestazioni e requisiti computazionali.

Segnaliamo che in certi casi il solver ha terminato l'esecuzione ad un tempo superiore al tempo massimo assegnato: questo è dovuto al fatto che il controllo sul time limit viene effettuato quando il solver è in uno stato consistente, cosa che talvolta lo porta a "sforare" significativamente i limiti assegnati. Altra caratteristica dei solver MIP, dovuta alla loro natura di branch-and-cut, è il loro comportamento erratico che rende impredicibile l'esecuzione; le scelte di branching effettuate all'interno del solver dipendono da molti fattori, spesso al di fuori della possibilità di intervento dell'utente. Oltre a questo, soluzioni come Dynamic Search per l'attraversamento dell'albero sono tenute nascoste all'utente, cosa che limita ulteriormente la possibilità di comprensione dell'esecuzione [1].

[1] Ad esempio, all'interno di una callback non è possibile ottenere un timestamp, perciò non si può valutare il tempo trascorso ed eventualmente abortire l'esecuzione. Anche il nodo in cui l'incumbent viene aggiornato è tenuto nascosto, e viene mostrato solo il suo valore arrotondato.

Capitolo 4

Euristici

Analizziamo infine alcune tecniche euristiche usate per calcolare una soluzione ammissibile, anche se in generale subottima, da impiegare come punto di partenza per il subgradiente lagrangiano, il branch-and-bound o una risoluzione basata su MIP, ma anche per calcolare una soluzione a sé stante. In questa sezione esponiamo e confrontiamo euristici basati sia sulla formulazione generale del problema, sia sul modello MIP.

4.1 Euristici basati sulla formulazione del problema

4.1.1 Nearest Neighbour

Il primo euristico che analizziamo è anche quello forse più intuitivo, l'algoritmo Nearest Neighbour (NN). Partendo da un nodo qualsiasi, l'algoritmo esegue una scelta greedy scegliendo l'arco di costo minimo tra quelli che collegano il nodo a nodi non già precedentemente visitati, fino a che tutti i nodi non sono stati visitati, al che l'algoritmo ritorna al nodo di partenza con l'unico arco possibile.

L'idea è molto semplice e "naturale", ma la scelta localmente ottima può, nel corso delle iterazioni, spingere l'algoritmo verso zone in cui ci sono diversi nodi molto vicini, costringendo poi la selezione di archi di costo elevato per muoversi verso i rimanenti nodi del grafo. Ciò risulta evidente osservando il risultato dell'esecuzione dell'algoritmo su un grafo, quando si notano archi che attraversano lunghi tratti per collegare due nodi distanti tra loro. Una possibile idea per attenuare in parte il problema è quella di eseguire più volte la ricerca, partendo da diversi nodi del grafo.

4.1.2 k-opt

Un'idea molto naturale che nasce dall'osservazione del risultato dell'algoritmo NN su istanze del TSP, o perlomeno su istanze del TSP che rispettano

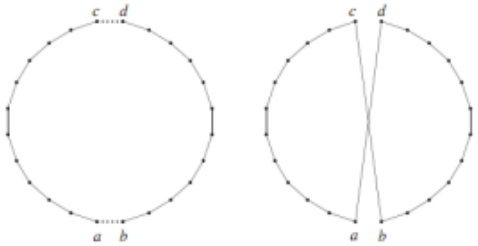

Figura 4.1: Scambio di archi in 2-opt.

la disuguaglianza triangolare, è quella di scambiare due archi che si incrociano, secondo l'operazione in figura 4.1. Tale operazione è chiamata 2-opt (Croes [5]). Il risultato è quello di collegare in maniera localmente ottima due coppie di nodi.

Questa procedura può essere estesa a più coppie di nodi. Ad esempio, lo scambio di archi tra 3 coppie di nodi è detta 3-opt, e, in generale, su k coppie di nodi si definisce un'operazione k-opt. Queste operazioni sono applicabili a qualsiasi tour hamiltoniano, per migliorare la soluzione corrente, ottenuta con un algoritmo euristico. Nel nostro caso abbiamo applicato 2-opt e 3-opt a cicli ottenuti con NN, e a cicli generati in maniera casuale. Nel primo caso l'idea è quella di migliorare una soluzione che si suppone già sufficientemente buona. Nel secondo caso invece di ottiene un circuito in genere pessimo dal punto di vista del costo, e molto intricato se osservato, ma molto veloce da calcolare e che può essere migliorato di molto; questo permette di generare moltissimi cicli e testare il loro "potenziale" applicando gli step di k-ottimalità in un tempo relativamente contenuto. Quest'ultimo procedimento è chiamato di seguito RC, per random cycle.

Le figure 4.2–4.4 mostrano le differenze tra NN, NN+2-opt, e NN+23opt.

Dal punto di vista teorico le operazioni k-opt costituiscono una ricerca locale in un intorno della soluzione corrente. Non sono quindi garantiti il raggiungimento della soluzione ottima, né la sua certificazione come tale. Al crescere di k la ricerca si estende a intorni di raggio sempre maggiore, a scapito tuttavia del tempo di calcolo e della complessità dell'algoritmo, che già per $k = 3$ diventa molto oneroso. Queste procedure sono comunque alla base degli euristici Lin-Kernighan (Lin and Kernighan [11]) e Lin-Kernighan-Helsgaun (Helsgaun [9]), ad oggi i più efficaci euristici per il TSP in letteratura.

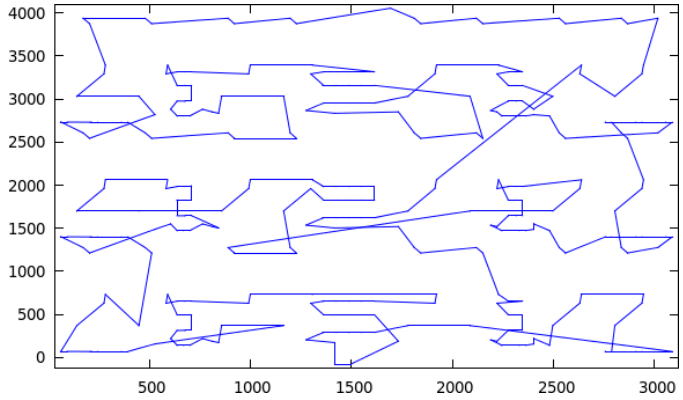

Figura 4.2: Risultato di NN per `lin318`.

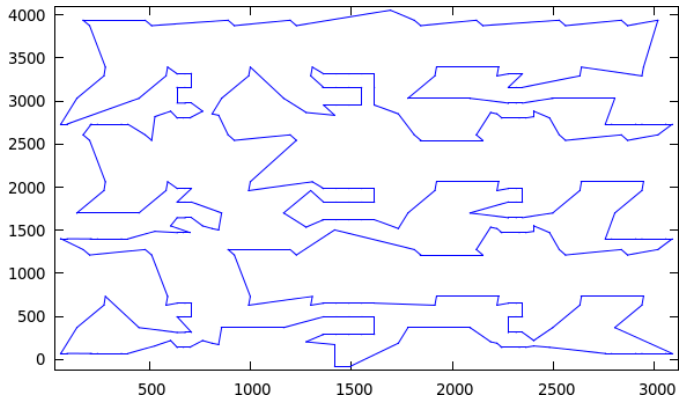

Figura 4.3: Risultato di NN+2opt per `lin318`.

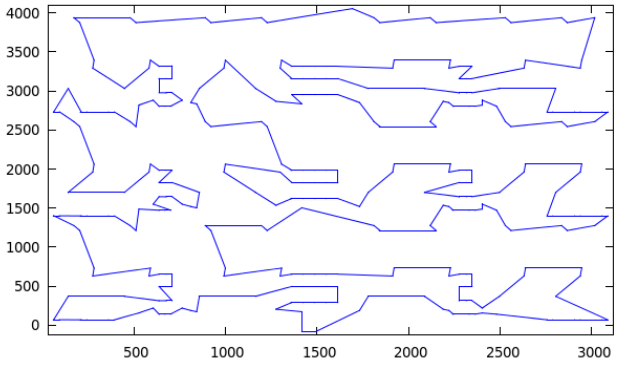

Figura 4.4: Risultato di NN+23opt per `lin318`.

4.1.3 Branch-and-bound con prefissaggio

Abbiamo inoltre provato un approccio ispirato a RINS (si veda la sezione 4.2.3), basato sulle soluzioni calcolate dall'euristico di partenza. L'idea è valutare le soluzioni calcolate ad ogni iterazione dell'euristico (in particolare RC+2opt, che genera molti cicli potenzialmente diversi, mentre NN+2opt restituirà un numero limitato di cicli distinti) e confidare nel fatto che gli archi selezionati in tutte le iterazioni sono parte della soluzione ottima; allo stesso modo, gli archi sempre scartati da tutte le iterazioni dell'euristico sono considerati esclusi anche dalla soluzione ottima. Vengono così fissate a zero le variabili corrispondenti agli archi scartati, e a 1 le variabili relative agli archi sempre presenti nelle varie soluzioni. Viene quindi lanciato il branch-and-bound, che agirà sulle variabili non fissate.

Ovviemente, non c'è nessuna garanzia che questo metodo porti alla soluzione ottima (basti pensare al caso limite in cui viene eseguita una sola iterazione dell'euristico: tutte le variabili verranno fissate al valore che assumono nella soluzione euristica, il branch-and-bound non ha gradi di libertà, e la soluzione ritornata sarà quella calcolata dall'euristico), ma possiamo sperare che se molte iterazioni concordano su alcune variabili, il loro valore sarà molto probabilmente lo stesso della soluzione ottima.

B & B con prefissaggio debole Nel tentativo di accelerare la ricerca di un migliore upper bound, abbiamo valutato l'idea di rilassare le condizioni necessarie per l'imposizione o la rimozione dei lati utilizzate nel B & B con prefissaggio. In particolare, considerato l'insieme di m cicli prodotto dall'euristico RC-2opt e scelto un fattore $0 < f \leq 1.00$, vengono fissate a 0 le variabili associati ai lati che non appartengo alla soluzione in almeno $m \times f$ cicli, mentre vengono fissate a 1 le variabili associate ai lati che appartengono alla soluzione in almeno $m \times f$ cicli. Si può osservare che, a differenza di prima, l'insieme di lati forzati e vietati così calcolato non è più, in generale, un'intersezione rispettivamente di lati forzati e vietati appartenenti ad una famiglia di cicli. Non è più quindi garantita la consistenza di tali vincoli. La probabilità di incorrere in inconsistenze o escludere buoni upper-bound aumenta al diminuire di f; per contro, aumenta la velocità di esecuzione del B & B. L'idea è quindi quella di tentare la ricerca di un nuovo upper bound applicando il B & B dopo aver fissato i vincoli ottenuti a partire da un fattore f piuttosto basso, nel nostro caso $f = 0.80$; successivamente, si procede per tentativi aumentando di volta in volta f di una certa quantità.

4.2 Euristici basati su modello MIP

L'altra tipologia di euristici che abbiamo analizzato e testato appartiene ad un filone di ricerca sviluppatosi negli ultimi anni, in seguito al diffondersi di software commerciali e non per la risoluzione di modelli MIP, e alla loro

sempre maggiore potenza. Sono conosciuti come *matheuristics*, crasi tra "euristici" e "programmazione matematica".

Anche in questo caso si tratta di tecniche di ricerca locale, che tuttavia sfruttano i solver MIP a scatola chiusa per esplorare in maniera più efficiente un intorno dello spazio delle soluzioni.

4.2.1 Hard Fixing

Il primo euristico MIP-based che abbiamo trattato è detto Hard Fixing, in quanto prevede il fissaggio di alcune variabili, lasciando che il solver risolva il sottoproblema così determinato. La scelta di quali variabili fissare è decisa dall'utente (eventualmente in maniera casuale).

La ricerca parte da una soluzione ammissibile iniziale precedentemente calcolata, fornita al solver. Al modello originale (intero o rilassato da risolvere mediante cutting plane) viene aggiunto un vincolo

$$\Delta(x, \tilde{x}) \leq k \tag{4.1}$$

che specifica una massima distanza di Hamming tra la soluzione ammissibile iniziale $\tilde{x}$ e le soluzioni x che il solver può generare. Dal punto di vista geometrico questo equivale a imporre un massimo raggio dell'intorno della soluzione in cui cercare soluzioni miglioranti. Dal punto di vista dell'albero di branching, invece, l'effetto è di un salto in un sottoalbero del branch-and-cut, motivo per cui tale tecnica è detta anche *diving*.

Il raggio dell'intorno k è il parametro che determina il trade-off tra qualità della soluzione e tempo impiegato: un valore basso di k restringe l'intorno, permettendo di trovare velocemente una soluzione migliorante, se presente, ma limitando le possibilità di miglioramento. Al contrario, un valore molto alto rende il vincolo inutile, facendo tendere la ricerca ad una ricerca globale.

4.2.2 Proximity Search

Proximity Search (Fischetti and Monaci [7]) è una tecnica euristica più recente che ricerca una soluzione migliore in un intorno dell'incumbent dato da un parametro di prossimità θ. Il metodo impiegato da Proximity Search consiste nel sostituire alla funzione obiettivo originale

$$\min c^T x \tag{4.2}$$

una nuova funzione obiettivo

$$\min \Delta(x, \tilde{x}) \tag{4.3}$$

e aggiungendo un nuovo vincolo

$$c^T x \leq c^T \tilde{x} - \theta \tag{4.4}$$

dove $\tilde{x}$ è la soluzione incumbent corrente, e $\Delta(x, \tilde{x})$ è la distanza di Hamming tra l'incumbent e una soluzione generica x. Il vincolo 4.4 è chiamato *vincolo di cutoff* che specifica di quanto le soluzioni ammissibili possono discostari dalla precedente, in termini di costo. L'idea alla base di questo procedimento è di cercare soluzioni miglioranti in un intorno ristretto, iterando questa ricerca partendo ogni volta dal nuovo incumbent, fino a che non viene raggiunta una qualche condizione di terminazione.

Nell'articolo originale vengono proposte alcune varianti a Proximity Search, con alcune idee per migliorare le prestazioni. La versione da noi implementata prevede il ricentramento della funzione obiettivo a partire da una soluzione euristica precedentemente calcolata.

4.2.3 RINS+Polishing

L'ultimo metodo di questa categoria è dato da una combinazione di due euristici presenti in CPLEX, rispettivamente RINS (Danna, Rothberg, and Le Pape [6]) e Polishing (Rothberg [15]). RINS risolve il rilassamento continuo del problema, individa le variabili che sono già intere nella soluzione frazionaria ottima e concordano con la soluzione euristica iniziale, le fissa nel modello intero e risolve il problema così ristretto, più semplice del problema originale avendo limitato la ricerca ad un numero inferiore di variabili, ma senza la garanzia di poter raggiungere la soluzione ottima. Polishing invece è un post-processing che esegue un genetico a partire dalla soluzione incumbent; in CPLEX viene eseguito come ultima operazione, lanciata quando si verifica una predeterminata condizione (ad esempio un certo numero di nodi valutati, o un certo tempo trascorso).

4.3 Risultati computazionali

Dato che lo scopo di queste soluzioni è la ricerca "veloce" di una soluzione "buona", i test sono stati effettuati con un tempo più limitato di quello concesso ai metodi per la risoluzione all'ottimo del problema. I test sono stati effettuati su una macchina con processore Intel Core i7 quad-core con hyperthreading a 2,40GHz e 4 GB di RAM.

4.3.1 NN, RC e k-opt

Abbiamo implementato e testato gli euristici 2-opt e 3-opt, applicandoli a cicli generati sia con algoritmo Nearest Neighbour (partendo da ogni nodo del grafo) che generando casualmente l'ordine dei nodi (metodo chiamato d'ora in avanti RC). Il numero delle iterazioni di RC è stato calcolato in maniera tale da cercare di limitare il tempo di esecuzione, dato che la 2-ottimalità di un circuito generato casualmente di, ad esempio, 100 nodi richiede molte più iterazioni della 2-ottimalità di un grafo di 30 nodi. Tuttavia, poiché

	NN		NN+2-opt		NN+23opt		RC+2-opt		RC+23opt			
Istanza	z'[%]	T[s]	z'[%]	T[s]	z'[%]	T[s]	z'[%]	T[s]	z'[%]	T[s]	z	z^*
a280	13.42	0.07	1.18	5.41	0.00	6.57	0.99	349.21	0.15	350.73	2623	2579
ali535	15.69	0.5	2.14	148.11	0.00	175.02	1.56	2174.07	0.13	2217.43	208383	202310
att48	13.02	0.0	1.18	0.02	0.24	0.02	**0.00**	6.63	**0.00**	6.63	10628	10628
att532	17.25	0.7	1.98	170.93	0.00	194.74	1.32	2103.6	0.94	2117.32	28474	27686
berlin52	8.47	0.0	**0.00**	0.01	**0.00**	0.01	**0.00**	5.25	**0.00**	5.25	7542	7542
bier127	12.56	0.01	1.25	0.4	0.57	0.45	0.00	58.89	0.00	58.93	119003	118282
ch130	16.16	0.01	2.00	0.66	1.60	0.76	0.67	69.62	0.00	69.73	6137	6110
ch150	8.53	0.02	1.25	0.87	0.75	0.98	0.72	92.75	0.00	92.91	6554	6528
d198	10.75	0.03	0.51	2.66	0.00	3.24	0.78	187.61	0.78	187.73	15909	15780
d493	13.53	0.37	1.49	93.02	0.00	120.43	2.90	1484.83	1.66	1502.71	35401	35002
d657	19.56	0.82	1.14	372.95	0.13	445.91	1.73	3101.57	0.00	3159.41	50330	48912
eil101	15.66	0.01	0.16	0.16	0.16	0.18	0.00	26.68	0.00	26.71	645	629
eil51	12.88	0.0	0.23	0.02	0.00	0.02	0.00	7.79	0.00	7.79	427	426
eil76	11.97	0.0	1.66	0.08	0.92	0.1	0.00	19.75	0.00	19.77	543	538
fl417	16.10	0.32	2.45	59.48	1.29	65.86	0.54	1325.16	0.00	1338.92	11961	11861
fri26	2.99	0.0	**0.00**	0.0	**0.00**	0.0	**0.00**	0.97	**0.00**	0.97	937	937
gil262	15.93	0.07	1.48	7.63	0.70	8.86	1.23	333.08	0.00	335.72	2435	2378
gr137	19.99	0.02	0.47	0.8	0.31	0.86	0.67	76.64	0.00	76.75	70400	69853
gr17	4.46	0.0	**0.00**	0.0	**0.00**	0.0	**0.00**	0.48	**0.00**	0.48	2085	2085
gr202	14.40	0.04	1.20	2.31	0.66	2.76	1.25	164.52	0.00	164.8	40885	40160
gr21	10.93	0.0	**0.00**	0.0	**0.00**	0.0	**0.00**	0.8	**0.00**	0.8	2707	2707
gr229	15.41	0.05	1.08	6.14	0.17	7.28	1.27	256.6	0.00	257.81	136466	134602
gr24	22.09	0.0	**0.00**	0.0	**0.00**	0.0	**0.00**	1.05	**0.00**	1.05	1272	1272
gr48	15.74	0.0	1.09	0.02	1.09	0.02	**0.00**	6.0	**0.00**	6.0	5046	5046
gr666	15.49	0.8	0.45	339.88	0.00	382.32	2.61	3282.66	0.95	3357.81	303958	294358
gr96	16.84	0.01	1.06	0.14	0.10	0.16	0.00	23.23	0.00	23.25	55633	55209
hk48	5.90	0.0	1.44	0.01	1.08	0.01	**0.00**	7.64	**0.00**	7.64	11461	11461
kroA100	16.05	0.0	1.91	0.36	0.88	0.42	**0.00**	41.0	**0.00**	41.02	21282	21282
kroA150	17.39	0.02	1.63	1.29	0.32	1.46	0.60	106.91	0.00	107.07	26815	26524
kroA200	16.39	0.04	0.57	3.3	0.00	3.85	0.44	205.28	0.09	205.45	29679	29368
kroB100	16.91	0.0	1.27	0.16	1.27	0.17	0.29	42.17	**0.00**	42.19	22141	22141
kroB150	19.63	0.02	0.78	1.63	0.43	1.8	0.00	104.87	0.00	104.93	26424	26130
kroB200	19.22	0.04	3.17	3.91	2.59	4.35	0.66	209.02	0.00	209.35	29683	29437
kroC100	13.92	0.01	1.33	0.17	0.40	0.2	0.00	43.81	0.00	43.83	20769	20749
kroD100	16.11	0.0	1.90	0.32	0.78	0.39	0.31	50.49	0.00	50.54	21404	21294

	NN		NN+2-opt		NN+23opt		RC+2-opt		RC+23opt			
Istanza	z'[%]	T[s]	z'[%]	T[s]	z'[%]	T[s]	z'[%]	T[s]	z'[%]	T[s]	z	z^*
kroE100	12.14	0.0	2.25	0.32	1.82	0.35	0.30	39.26	0.00	39.29	22100	22068
lin105	17.78	0.01	1.83	0.31	0.61	0.36	0.56	40.19	**0.00**	40.21	14379	14379
lin318	15.70	0.13	3.22	21.26	2.17	23.67	1.44	522.53	0.00	525.84	42525	42029
p654	23.44	0.68	0.32	392.56	0.00	409.6	0.47	3641.18	0.13	3674.83	34858	34643
pcb442	13.45	0.26	1.09	40.08	0.00	48.81	2.57	983.12	2.10	990.06	51960	50778
pr107	5.37	0.0	**0.00**	0.17	**0.00**	0.18	0.50	37.25	0.50	37.27	44303	44303
pr124	13.51	0.0	0.14	0.23	0.00	0.29	0.00	58.86	0.00	58.9	59076	59030
pr136	18.19	0.01	3.85	0.84	3.22	1.08	0.42	68.84	0.00	68.95	96920	96772
pr144	4.15	0.01	0.23	0.3	**0.00**	0.48	0.11	75.42	**0.00**	75.52	58537	58537
pr152	7.78	0.02	0.84	0.8	0.44	0.91	0.00	82.6	0.00	82.66	73822	73682
pr226	14.84	0.04	0.93	2.95	0.43	3.41	0.04	206.03	0.00	206.37	80590	80369
pr264	10.32	0.07	3.05	4.92	2.17	6.28	0.52	308.98	0.00	310.0	49395	49135
pr299	19.39	0.08	1.87	11.78	0.82	13.33	0.99	478.07	0.00	479.44	48816	48191
pr439	17.34	0.24	2.61	49.24	0.65	67.79	0.97	1122.66	0.00	1137.23	108425	107217
pr76	20.78	0.0	0.62	0.07	0.60	0.09	0.03	13.95	0.00	13.97	108395	108159
rat195	10.54	0.03	0.85	1.72	0.38	1.93	0.51	165.58	0.00	166.06	2363	2323
rat575	13.30	0.53	0.72	159.83	0.00	182.16	1.49	2399.29	0.71	2424.01	7055	6773
rat783	15.47	1.61	1.28	649.3	0.00	750.62	2.22	5431.2	0.64	5544.48	9128	8806
rat99	17.98	0.0	0.82	0.17	0.57	0.2	0.16	31.67	0.00	31.69	1218	1211
rd100	19.08	0.0	2.27	0.2	2.00	0.22	0.08	27.97	0.00	27.99	7913	7910
rd400	17.89	0.22	1.28	44.92	0.00	52.53	1.98	948.75	0.20	964.19	15634	15281
st70	17.75	0.0	1.48	0.04	1.48	0.04	0.00	10.98	0.00	10.99	676	675
swiss42	12.88	0.0	0.08	0.01	0.08	0.01	**0.00**	2.99	**0.00**	2.99	1273	1273
ts225	10.79	0.03	1.16	1.37	0.00	2.1	0.72	176.59	0.00	177.29	126809	126643
tsp225	14.14	0.05	1.10	3.96	0.00	4.5	1.07	223.56	1.07	223.75	4011	3916
u159	14.63	0.02	3.19	0.7	2.25	0.78	0.20	92.84	0.00	92.93	42389	42080
u574	19.28	0.52	1.38	180.69	0.00	217.07	1.65	2084.0	0.62	2111.26	38096	36905
u724	17.94	1.13	1.75	488.58	0.00	580.54	2.75	3661.33	1.83	3747.88	43073	41910
ulysses16	15.80	0.0	**0.00**	0.0	**0.00**	0.0	**0.00**	0.31	**0.00**	0.31	6859	6859
ulysses22	16.63	0.0	**0.00**	0.0	**0.00**	0.0	**0.00**	0.7	**0.00**	0.7	7023	7023

Tabella 4.2: Risultati degli euristici NN, NN+2opt, NN+23opt, RC+2opt, RC+23opt.

RC richiede un numero relativamente elevato di iterazioni per poter sperare di ottenere qualche soluzione accettabile, è stato implementato in parallelo, con un diverso seme per ogni thread.

Sia per NN che RC, ad ogni ciclo calcolato viene applicato 2-opt; al termine, sulla migliore soluzione viene applicato 3-opt. Si è scelto di applicare 3-opt solo sulla migliore soluzione a causa della sua pesantezza; la successiva applicazione di 2-opt permette di limare ulteriormente il bound. Ovviamente, non c'è nessuna garanzia che questa scelta porti al miglior risultato, ovvero l'applicare lo step 3-opt solo alla miglior soluzione potrebbe non essere la scelta ottimale dal punto di vista della qualità della soluzione ottenuta. Anche in questo caso si è comunque scelto un compromesso tra qualità della soluzione e tempo impiegato per ottenerla.

La tabella 4.1 va interpretata come segue: per ogni algoritmo testato sono riportati la differenza (in percentuale) rispetto al miglior risultato ottenuto tra gli euristici, e il tempo impiegato per ottenerla. Le ultime due colonne mostrano il miglior risultato ottenuto tra gli euristici, e il costo reale della soluzione ottima. I risultati in grassetto indicano che l'upper bound trovato dall'algoritmo è la soluzione ottima.

NN+2opt permette di trovare la soluzione ottima per cicli più piccoli, mentre RC+2opt riesce ad individuare il tour ottimo anche per istanze fino a circa 100 nodi, e ad ottenere buoni bound anche per istanze di dimensioni maggiori. La causa principale per cui le prestazioni di RC2opt peggiorano al crescere delle istanze è da cercarsi nel fatto che, cercando di bilanciare la qualità del risultato con i tempi necessari ad ottenerlo, al crescere delle dimensioni delle istanze si effettuano sempre meno iterazioni, con la conseguenza che le possibilità di trovare una buona soluzione si riducono sempre di più. Molto probabilmente, il mantenere un elevato numero di iterazioni a prescindere dalla dimensione dell'istanza contribuirebbe al raggiungimento di soluzioni migliori, a scapito tuttavia del tempo di esecuzione, che crescerebbe in maniera enorme.

Chiaramente, 3-opt permette di migliorare anche di molto la soluzione ritornata. Date le prestazioni riportate nella tabella, come euristico di partenza per le prove basate su CPLEX è stato usato RC+23opt; per il branch-and-bound, invece, è stato usato anche il B&B con prefissaggio, essendo le istanze provate con quel metodo solo quelle di taglia inferiore ai 300 nodi e potendo quindi sfruttare a pieno anche questo euristico.

4.3.2 B&B con prefissaggio

Riportiamo ora i risultati dei test del branch-and-bound con prefissaggio delle variabili in seguito agli euristici. L'euristico usato, come detto più sopra, è RC+23opt. Come il branch-and-bound non limitato, anche questa versione euristica è stata testata sulle istanze di taglia inferiore ai 300 nodi, con un tempo limite di 2000 secondi. Segnaliamo inoltre che questa versione

Istanza	UB	LB	z^*	IG[%]	FG[%]	T[s]	AR[%]	z^*	FG[%]	T[s]
a280	2627	2566.00	2627	2.3773	2.3773	2000.0		2579	0.00	49.27
att48	10628	10 604.00	10628	0.2263	0.0	0.01	7.5354	10628	0.00	0.01
berlin52	7542	7542.00	7542	0.0	0.0	0.00	5.2790	7542	0.00	0.00
bier127	119003	118 068.00	118282	0.7919	0.0	0.68	4.1619	118282	0.00	2.17
ch130	6178	6075.50	6110	1.6871	0.0	45.46	4.2098	6110	0.00	18.31
ch150	6657	6490.75	6528	2.5613	0.0	104.30	3.4093	6528	0.00	32.61
d198	15863	15 717.00	15781	0.9289	0.0	343.39	2.8867	15780	0.00	1374.91
eil101	634	627.50	629	1.0363	0.0	2.33	5.8613	629	0.00	24.1
eil51	427	422.50	426	1.0654	0.0	0.17	9.1764	426	0.00	1.07
eil76	546	537.00	538	1.6760	0.0	0.02	7.7543	538	0.00	0.03
fri26	937	937.00	937	0.0001	0.0	0.00	11.0769	937	0.00	0.00
gil262	2419	2355.50	2415	2.6959	2.5261	2000.0		2410	2.3355	2000.0
gr137	70317	69 120.25	69853	1.7314	0.0	18.86	3.5530	69853	0.00	1.05
gr17	2085	2085.00	2085	0.0	0.0	0.00	12.5	2085	0.00	0.00
gr202	41069	40 055.00	40160	2.5315	0.0	116.25	2.8028	40160	0.00	121.67
gr21	2707	2707.00	2707	0.0	0.0	0.00	0.0	2707	0.00	0.00
gr229	137445	133 317.02	136329	3.0963	2.2592	2000.0		134829	1.1341	2000.0
gr24	1272	1271.98	1272	0.0014	0.0	0.00	12.6811	1272	0.00	0.00
gr48	5046	4959.00	5046	1.7544	0.0	0.10	10.1950	5046	0.00	0.00
gr96	55260	54 570.50	55210	1.2635	0.0	1.80	5.3728	55210	0.00	2.18
hk48	11461	11 441.07	11461	0.1742	0.0	0.00	6.8262	11461	0.00	0.00
kroA100	21282	20 936.50	21282	1.6502	0.0	1.19	4.3434	21282	0.00	0.00
kroA150	26842	26 299.00	26524	2.0647	0.0	23.24	3.4541	26524	0.00	2.87
kroA200	29575	29 065.00	29522	1.7547	1.5723	2000.0		29368	1.0425	2000.0
kroB100	22278	21 834.00	22141	2.0335	0.0	1.44	5.0707	22141	0.00	2.15
kroB150	26338	25 732.50	26130	2.3531	0.0	622.89	3.4899	26130	0.00	16.47
kroB200	30002	29 176.50	29635	2.8293	1.5715	2000.0		29437	0.00	1641.73
kroC100	20769	20 472.50	20749	1.4483	0.0	0.54	4.5858	20749	0.00	0.00
kroD100	21395	21 141.50	21294	1.1991	0.0	1.42	5.2525	21294	0.00	0.50
kroE100	22260	21 799.50	22068	2.1124	0.0	6.14	5.3535	22068	0.00	12.34
lin105	14379	14 370.50	14379	0.0592	0.0	0.01	2.8754	14379	0.00	2.31
pr124	59087	58 286.67	59030	1.3731	0.0	0.18	2.7930	59030	0.00	0.00
pr136	97644	95 934.49	96772	1.7820	0.0	302.71	4.3790	96772	0.00	134.01
pr144	58571	58 216.25	58537	0.6094	0.0	0.10	2.1173	58537	0.00	1.91
pr152	74021	73 234.50	73682	1.0739	0.0	2.48	2.5880	73682	0.00	1.83
pr226	80590	80 204.00	80369	0.4813	0.0	2.45	1.5496	80369	0.00	14.13

Istanza	UB	LB	z^*	IG[%]	FG[%]	T[s]	AR[%]	z^*	FG[%]	T[s]
pr264	49395	49 020.18	49135	0.7647	0.0	14.33	1.990	49135	0.00	17.29
pr299	48816	47 380.00	48816	3.0308	3.0308	2000.0		48816	3.0308	2000.0
pr76	108395	105 120.00	108159	3.1155	0.0	11.76	6.8070	108159	0.00	4.04
rat195	2386	2299.25	2323	3.7730	1.0330	2000.0		2323	0.00	1202.47
rat99	1218	1206.00	1211	0.9951	0.0	0.40	5.1535	1211	0.00	2.18
rd100	7913	7899.33	7910	0.1730	0.0	0.11	3.7777	7910	0.00	1.02
st70	676	671.00	675	0.7452	0.0	0.33	6.4596	675	0.00	0.3
swiss42	1273	1272.00	1273	0.0786	0.0	0.00	7.3170	1273	0.00	0.00
ts225	126809	115 605.00	126809	9.6916	9.6916	2000.0		126809	9.6916	2000.0
tsp225	4047	3878.25	4047	4.3512	4.3512	2000.0		3999	3.1135	976.62
u159	42543	41 925.00	42168	1.4741	0.0	57.40	2.8898	42080	0.00	18.08
ulysses16	6859	6859.00	6859	0.0	0.0	0.00	13.3333	6859	0.00	0.00
ulysses22	7013	7012.87	7013	0.0019	0.0	0.00	14.7186	7013	0.00	0.00

Tabella 4.4: Risultati del branch-and-bound con prefissaggio di variabili.

euristica del branch-and-bound ha risolto (trovando la soluzione ottima) l'istanza `lin318` in poco più di 40 minuti.

Nella tabella 4.3 sono riportati, per ciascuna istanza, i due bound e il valore finale ottenuto, il gap iniziale percentuale tra i due bound, il gap percentuale finale (0.0 indica che l'istanza è stata risolta all'ottimo), il tempo impiegato e la percentuale di archi su cui il B&B ha lavorato, cioè la percentuale di variabili non fissate dal preprocessing.

Nelle istanze più grandi il branch-and-bound, anche se limitato a poche centinaia di variabili, non è riuscito a trovare l'ottimo (locale) nel tempo dato. In alcune di esse, non è stato nemmeno rilevato alcun miglioramento del bound iniziale. Tuttavia, nelle rimanenti istanze testate, l'algoritmo ha quasi sempre trovato l'ottimo globale, ad eccezione di poche istanze, in cui sono state trovate soluzioni di costo di poco superiore all'ottimo), in un tempo considerevolmente ridotto rispetto al branch-and-bound presentato nel capitolo 2. Sono state inoltre affrontate e risolte istanze che altrimenti sarebbero state proibitive, come `pr264`. Nella maggior parte dei casi, le istanze che terminano entro il limite dato sono state completate rimanendo anche entro il tempo limite di 500 secondi assegnato agli euristici basati su CPLEX.

Chiaramente le prestazioni del RC+2opt a monte del B&B influenzano fortemente l'esito di quest'ultimo: aumentare il numero di iterazioni aumenta di conseguenza le probabilità di ottenere un bound di partenza migliore e una soluzione di costo minore, ma aumenta allo stesso tempo anche la possibilità di trovare un circuito che non si "allinea" a quelli precedentemente trovati, riducendo quindi il numero di variabili che si possono fissare. Al contrario, con meno iterazioni in genere ci saranno tendenzialmente più variabili fissate, ma la soluzione finale sarà cercata in uno spazio delle soluzioni più limitato, con minori garanzie sulla sua qualità. Oltre a ciò, è immediato notare come le istanze che non terminano sono anche quelle in cui il gap iniziale tra i bound è più ampio, e di conseguenza ci sono molte più variabili libere.

Proprio per questo motivo, vale a dire la possibilità di eliminare più archi, la versione "rilassata" del B& B con prefissaggio consente di risolvere istanze che la versione che richiede il 100% di consenso sugli archi non riesce a sfruttare, come ad esempio `kroA200`, `kroB200` e `a280`, e in generale è più veloce.

4.3.3 Euristici basati su solver MIP

In tabella 4.5 sono riportati i risultati dei test effettuati usando gli euristici basato su CPLEX. Valori mancanti indicano che l'esecuzione è stata abortita per aver ecceduto il time limit senza terminare, e non è stato possibile ottenere alcuna informazione sul suo stato al momento della terminazione.

			HF-iter		HF-cb		PS-iter		PS-cb		RINS+Polishing-cb	
Istanza	UB	LB	z^*	T[s]	z^*	T[s]	z^*	T[s]	z^*	T[s]	z^*	T[s]
a280	2627	2566	2579	6.08	2579	0.44	2620	5.24	2622	0.72	2588	1238.01
ali535	209431	201237	208716	9.48	209431	0.99	209307	512.58	209119	382.91	204907	1096.32
att48	10628	10603	10628	0.05	10628	0.03	10628	0.43	10628	193.77	10628	1.63
att532	28743	27420	28068	86.35	28041	17.73	28732	561.05	28736	181.84	27735	1067.32
berlin52	7542	7542	7542	0.01	7542	0.01	7542	0.28	7542	0.00	7542	0.00
bier127	119903	117431	118282	0.26	118282	0.20	118719	0.48	118898	0.68	118282	7.90
ch130	6176	6076	6110	0.55	6110	0.19	6121	5.01	6133	1.04	6110	950.49
ch150	6548	6491	6528	2.92	6528	0.42	6581	1.13	6533	1.86	6528	1076.62
d198	15948	15712	15780	6.72	15780	3.48	15848	21.02	15848	2.64	15780	1326.36
d493	35987	34829	35562	178.74	35506	18.73	35981	508.18	35984	116.94	35358	1118.15
d657	50330	48456			49931	530.61	50311	549.90	50286	52.38	49007	1035.64
eil101	634	628	629	0.49	629	0.28	632	2.25	632	0.60	629	37.07
eil51	427	423	426	0.07	426	0.03	427	0.23	426	0.03	426	8.47
eil76	543	537	538	0.08	538	0.02	543	0.19	544	0.24	538	0.19
fl417	11961	11790	10643	2729.19	11868	577.95	11953	448.70	11953	16.52	11863	1276.10
fri26	937	937	937	0.00	937	0.00	937	0.00	937	0.00	937	0.00
gil262	2438	2355	2378	12.10	2378	13.63	2434	52.11	2435	25.43	2378	1283.29
gr137	70317	69121	69853	1.89	69853	0.40	70157	2.20	70117	0.56	69853	1157.52
gr17	2085	2085	2085	0.00	2085	0.00	2085	0.00	2085	0.00	2085	0.00
gr202	41100	40055	40160	5.42	40160	1.04	41052	57.89	40994	2.86	40160	1306.27
gr21	2707	2707	2707	0.01	2707	0.01	2707	0.00	2707	0.00	2707	0.00
gr229	138040	133318	134602	20.14	134602	3.23	137075	30.59	136072	4.19	134602	1294.02
gr24	1272	1272	1272	0.00	1272	0.02	1272	0.02	1272	0.00	1272	0.02
gr48	5046	4959	5046	0.11	5046	0.03	5046	2.45	5046	0.43	5046	42.60
gr666	304907	292494	298431	8.79	298434	1.85	304796	506.99	304858	303.37	297102	971.19
gr96	55589	54571	55210	1.46	55210	0.18	55210	1.67	55513	0.44	55210	355.47
hk48	11461	11442	11461	0.02	11461	0.01	11461	0.10	11461	0.01	11461	0.00
kroA100	21319	20937	21282	0.54	21282	0.13	21282	3.26	21282	3.54	21282	1179.49
kroA150	26774	26299	26524	13.06	26524	1.09	26661	1.89	26775	1.42	26524	1067.88
kroA200	29690	29065	29368	32.32	29368	10.85	29574	20.05	29575	1.85	29368	1327.63
kroB100	22141	21834	22141	1.70	22141	0.57	22158	2.43	22235	1.52	22141	1144.44
kroB150	26344	25733	26130	14.95	26130	4.58	26436	4.54	26374	0.69	26132	1085.49
kroB200	30020	29165	29448	4.05	29447	1.16	30038	20.64	29655	1.31	29438	1166.82
kroC100	20769	20473	20749	0.50	20749	0.07	20749	5.79	20749	0.81	20749	427.23
kroD100	21404	21142	21294	1.04	21294	0.20	21381	6.76	21346	0.91	21294	151.95

			HF-iter		HF-cb		PS-iter		PS-cb		RINS+Polishing-cb	
Istanza	UB	LB	z^*	T[s]	z^*	T[s]	z^*	T[s]	z^*	T[s]	z^*	T[s]
kroE100	22100	21800	22068	0.75	22068	0.27	22178	5.27	22073	1.80	22068	1130.23
lin105	14379	14371	14379	0.03	14379	0.01	14379	1.11	14402	0.55	14379	5.78
lin318	43200	41889	42029	23.20	42029	2.87	43167	157.04	43153	53.80	42050	1409.98
p654	34905	33567	42029	23.20	34753	11.88	34758	508.57	34870	349.85	34738	1272.78
pcb442	53051	50500	51507	441.87	51642	5.41	53040	436.58	53032	75.41	50778	1176.46
pr107	44526	43573	44303	0.08	44303	0.02	44482	8.58	44522	14.37	44303	0.03
pr124	59087	58068	59030	0.78	59030	0.04	59030	30.30	59159	3.39	59076	1097.66
pr136	97644	95935	96772	1.64	96772	0.19	97456	3.82	97594	0.30	96772	1090.81
pr144	58571	58158	58537	0.26	58537	0.09	58554	11.19	58554	3.72	58537	1132.69
pr152	74021	73209	73682	0.68	73682	0.35	74017	10.14	74017	7.85	73682	1099.28
pr226	80590	80092	80369	1.13	80369	5.69	80508	32.43	80508	3.86	80369	1138.82
pr264	49395	49005	49135	66.92	49135	1.85	49363	16.73	49151	89.87	49135	1197.71
pr299	48816	47380	48191	104.43	48191	21.46	48787	67.19	48786	4.92	48220	1291.68
pr439	108425	105929	107323	147.91	107323	130.94	108415	364.82	108325	104.63	107250	1285.75
pr76	108395	105100	108159	1.93	108159	0.33	108381	1.40	108381	0.66	108274	1241.69
rat195	2371	2300	2323	22.41	2323	1.42	2383	9.45	2383	6.03	2325	1079.34
rat575	7105	6724	6904	157.95	6904	147.68	7095	576.81	7100	93.07	6802	1004.64
rat783	9186	8773	9011	179.22	9011	219.04	9175	555.25	9175	583.55	8827	582.56
rat99	1218	1206	1211	0.21	1211	0.10	1215	0.12	1213	0.05	1211	0.29
rd100	7913	7900	7910	0.45	7910	0.12	7910	0.63	7910	0.30	7910	0.34
rd400	15666	15157	15507	15.65	15500	1.86	15648	73.76	15648	77.30	15281	1147.82
st70	675	671	675	0.12	675	0.04	675	0.01	675	0.16	675	4.21
swiss42	1273	1272	1273	0.01	1273	0.00	1273	0.01	1273	0.01	1273	0.00
ts225	126809	115605	126726	2.18	126796	0.29	126962	159.80	126726	120.90	126643	1162.75
tsp225	4024	3879	3916	13.91	3916	5.09	4004	69.75	3983	3.42	3916	1265.52
u159	42624	41925	42080	0.94	42080	0.13	42465	2.42	42162	0.44	42080	892.53
u574	38334	36714	37567	34.59	37567	22.57	38332	540.13	38321	299.00	36939	1088.53
u724	43860	41653	43612	27.35	43539	1.80	43801	588.76	43854	625.94	43145	1137.15
ulysses16	6859	6859	6859	0.01	6859	0.01	6859	0.00	6859	0.00	6859	0.00
ulysses22	7013	7013	7013	0.01	7013	0.01	7013	0.03	7013	0.00	7013	0.00

Tabella 4.6: Risultati di Hard Fixing, Proximity Search, RINS.

Abbiamo implementato e testato Hard Fixing sfruttando le informazioni sugli archi ricavate dall'euristico, scegliendo inizialmente come prime variabili da fissare quelle su cui tutte le iterazioni dell'euristico iniziale concordano; poi, tra le rimanenti variabili, in maniera casuale abbiamo selezionato le ultime variabili da imporre. A queste variabili è stato assegnato il valore identificato dalla miglior soluzione dell'euristico iniziale.

Il raggio dell'intorno, dopo alcuni test su istanze medio-grandi del testbed usato, è stato posto a 1000; valori inferiori come 10, 20, 100, 500, portavano il solver a terminare quasi immediatamente senza alcun miglioramento dell'incumbent. Per istanze di taglia "piccola", ovviamente, questo vincolo è poco o per nulla significativo; istanze di tagli limitata, comunque, vengono facilmente risolte all'ottimo (in alcuni casi anche dagli euristici di partenza), motivo per cui la scelta di un euristico basato su MIP ha poco senso. Vengono comunque riportati per completezza i risultati ottenuti su tutte le istanze di prova usate. Come per gli altri euristici, è stato assegnato un tempo limite per ciascuna istanza di 500 secondi.

L'implementazione che fa uso di callback ha risolto all'ottimo locale in tempi molto brevi (pochi secondi) quasi tutte le istanze. Quella basata su risoluzione iterativa, invece, in parecchi casi impiegato tempi superiori, segno evidente di come il dover risolvere da capo il modello sia un compito eccessivamente pesante rispetto alla possibilità di muoversi all'interno dello spazio delle soluzioni ammissibili, in particolar modo in un contesto euristico.

La tecnica Proximity Search è stata implementata e testata con un tempo limite di 500 secondi; la ricerca partiva dalla soluzione calcolata con RC+23opt e veniva terminata al raggiungimento del time limit o della prima soluzione migliorante. Si nota come la versione basata su callbacks è generalmente molto più veloce di quella basata su CPLEX in modalità iterativa. In alcuni casi la soluzione individuata è la soluzione ottima; in molti altri casi, la veloce individuazione della prima soluzione migliorante suggerisce un rapido avvicinamento ad una buona soluzione, cosa auspicabile soprattutto per istanze di grandi dimensioni, o difficili (come `ts225`).

Le prestazioni riportate mostrano comunque come una risoluzione limitata ad un numero ristretto di variabili sia molto performante anche rispetto ad un procedimento che valuta il modello intero.

Abbiamo infine testato sulle medesime istanze gli euristici RINS e Polishing presenti in CPLEX. Anche in questo caso il tempo limite assegnato è stato di 500 secondi, in cui, a partire da una soluzione iniziale precedentemente calcolata con RC+23opt, è stato eseguito RINS al 50% dei nodi del branch-and-cut fino alla prima soluzione, momento in cui il b&c è stato fermato e si è lanciato il Polishing per il tempo rimanente.

Su istanze di dimensioni limitata, RINS+Polishing ha ottenuto buone soluzioni, spesso raggiungendo l'ottimo, senza tuttavia poterlo certificare (essendo il Polishing un algoritmo genetico). In particolare, questo euristico è stato anche l'unico in grado di trovare la soluzione ottima dell'istanza

`ts225`. Anche sulle istanze più grandi del testbed, nel tempo dato è stato possibile individuare soluzioni di buona qualità, molto vicine all'ottimo.

4.4 Commenti

Gli euristici basati su k-opt permettono di ottenere buoni risultati su istanze anche di taglie relativamente elevate. Tuttavia, partire da una soluzione calcolata con NN permette di ottenere un numero limitato di istanze diverse, mentre RC+2opt/23opt, che ha a suo favore un più elevato potenziale, diventa computazionalmente sempre più pesante al crescere delle istanze. Possiamo tuttavia affermare che, tra le varianti implementate, RC2opt è l'euristico che, con maggior tempo di calcolo a disposizione, è in grado di fornire i migliori risultati; inoltre, è parallelizzabile in maniera molto naturale. Tale affermazione è supportata dalla percentuale decrescente di archi eliminati, che da molto elevata per istanze di taglia limitata, scende progressivamente mano a mano che aumenta il numero di nodi nel grafo: cio è ovviamente dovuto a bound peggiori.

Il branch-and-bound con prefissaggio delle variabili si è rivelato molto efficace su istanze di taglia moderata (< 300 nodi), spesso trovando l'ottimo globale, ma non ci sono indizi che suggeriscano una sua reale applicabilità a istanze più grandi di 300 nodi. In altre parole, la sua qualità è quella di accelerare la risoluzione di istanze che con ogni probabilità sono alla portata di un normale branch-and-bound (come quello presentato nel capitolo 2), se questo fosse eseguito per un tempo maggiore (qualche ora). Le istanze più difficili, come `ts225`, rimangono al di fuori della sua portata. Il vantaggio ottenuto sul tempo viene tuttavia pagato in termini di garanzia dell'ottimalità della soluzione individuata.

Gli euristici basati su risolutori MIP sono quindi, tra le soluzioni analizzate, quelle a cui rivolgersi per attaccare istanze di dimensioni più elevate. Hard Fixing converge all'ottimo locale molto velocemente, anche se in maniera poco predicibile e poco dipendente dall'effettiva dimensione dell'istanza; per di più, la qualità della soluzione ritornata è legata alla scelta delle variabili imposte, scelta che richiede della conoscenza pregressa del dominio per poter essere effettuata con cognizione di causa. Proximity Search ha mostrato in molti casi un rapido aggiornamento iniziale, ma sulle istanze più grandi ha impiegato molto tempo ad aggiornare l'incumbent. La coppia di euristici interni a CPLEX RINS+Polishing, fornisce risultati molto buoni nei tempi dati, cosa tuttavia poco sorprendente data la qualità del solver. L'operazione di crossing over dell'algoritmo genetico su cui si basa Polishing permette di esplorare maggiormente lo spazio di ricerca, mentre le altre tecniche analizzate, così come il Branch-and-bound con prefissaggio, sono limitate ad un intorno della soluzione iniziale fornita (sia pure "mobile", nel caso di Proximity Search). Inoltre, il fatto di essere implementazioni

interne a CPLEX consente loro di sfruttare a pieno le potenzialità e il tuning del solver, cosa preclusa alle nostre implementazioni, sia per l'uso limitato del solver consentito alle implementazioni "esterne" (come ad esempio la disabilitazione di dynamic search quando vengono usate le callback), sia per la nostra minore conoscenza del solver.

La principale difficoltà nell'implementare euristici usando un MIP solver, perlomeno basandosi su CPLEX o altre soluzioni commerciali, sta nel fatto che i risolutori commerciali nascondono il più possibile all'utente il proprio funzionamento interno per proteggere i propri segreti industriali, e di conseguenza diventa più difficile capire se l'algoritmo implementato è corretto, totalmente errato, o migliorabile, o individuare la causa degli errori che si possono verificare.

Capitolo 5

Conclusioni

Abbiamo presentato le nostre implementazioni degli homework assegnati durante il corso, assieme ai risultati computazionali, riguardandi diverse strategie per la risoluzione del Problema del Commesso Viaggiatore. Abbiamo quindi riportato i risultati ottenuti testando le soluzioni implementate su alcune istanze della libreria TSPLIB, un benchmark standard per questo problema.

Il branch-and-bound costruito sul rilassamento lagrangiano basato su 1-albero si è rivelato efficace su istanze fino a circa 200-250 nodi; tuttavia, vi sono istanze di taglia inferiore a questo limite che non sono state risolte nel limite di tempo assegnato a ciascuna istanza. Per risolvere all'ottimo problemi di taglia superiore è stato necessario usare solver MIP, che "ignorano" l'interpretazione reale del modello (ovvero il percorso di un agente attraverso tutti i nodi del grafo) e risolvono il modello di programmazione lineare mista-intera. Anche in questo caso è tuttavia necessario rilassare il problema eliminando i vincoli di eliminazione dei sottocicli, e inserirli come cutting planes quando necessario. Con questo approccio sono state attaccate con successo istanze fino a oltre 700 nodi. Infine abbiamo presentato alcuni approcci euristici di vario genere, sia motivati dall'interpretazione "geometrica" del problema (NN e k-opt) sia basati su metodi di risoluzione ottima (branch-and-bound e solver MIP) per l'individuazione di buoni cicli hamiltoniani, anche subottimi, da impiegare anche come subroutine nei metodi precedenti per ottenere dei bound sul costo e delle soluzioni di partenza.

Pur ricordando che non necessariamente un'istanza "grande" è più difficile di una di taglia più limitata, la chiave per approcciare istanze medio-grandi è ridurre drasticamente lo spazio di ricerca. Le prestazioni degli euristici e del subgradiente lagrangiano sono fondamentali in questo; nel caso questi forniscano dei bound laschi, lo spazio di ricerca non viene ridotto in maniera significativa e, dato lo stato di problema NP-completo, l'istanza rimane intrattabile. Ad esempio, la difficoltà nella risoluzione di `ts225`, che permane con la maggior parte dei metodi di risoluzione implementati, è dovuta

alle pessime prestazioni del lagrangiano, che non riesce a calcolare un lower bound accettabile (l'upper bound invece è molto buono). In questo modo, non è possibile eliminare un numero di archi sufficiente a rendere l'istanza trattabile. È significativo che l'unico metodo tra quelli proposti in grado di trovare (senza certificare) la soluzione ottima sia il Polishing, che può esplorare in maniera più erratica lo spazio delle soluzioni, e che al contempo ha fornito risultati peggiori rispetto agli altri metodi su altre istanze.

Questi metodi, tuttavia, pur spinti al massimo grado possibile di ottimizzazione e tuning, sono sufficienti a risolvere istanze di difficoltà limitata. Per approcciare, ad esempio, istanze dell'ordine di migliaia di nodi è necessario adottare altre tecniche, motivo per cui la letteratura sul TSP è estremamente ricca.

Bibliografia

[1] David Applegate, Robert Bixby, Vasek Chvatal, and William Cook. Concorde tsp solver, 2006.

[2] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study.* Princeton University Press, 2011.

[3] Pascal Benchimol, Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem-Jan van Hoeve. Improving the held and karp approach with constraint programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 40–44. Springer, 2010.

[4] Pascal Benchimol, Willem-Jan Van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3):205–233, 2012.

[5] GA Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[6] Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 102(1):71–90, 2005.

[7] Matteo Fischetti and Michele Monaci. Proximity search for 0-1 mixed-integer convex programming. Technical report, Technical Report, DEI, University of Padova (in preparation), 2012.

[8] Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[9] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[10] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[11] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[12] P Miliotis. Using cutting planes to solve the symmetric travelling salesman problem. *Mathematical programming*, 15(1):177–188, 1978.

[13] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review*, 33(1):60–100, 1991.

[14] Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[15] Edward Rothberg. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, 19(4): 534–541, 2007.

[16] Ton Volgenant and Roy Jonker. A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation. *European Journal of Operational Research*, 9(1):83–89, 1982.