Università di Pisa

Dipartimento di Matematica
Corso di Laurea in Matematica

Tesi di Laurea Magistrale

Valutazione sperimentale di metodi risolutivi per problemi "master" nella generazione di colonne

RELATORE
Prof. Antonio Frangioni

CANDIDATO
Giuliano Vallese

ANNO ACCADEMICO 2014/2015

Indice

Capitolo 1

Scopo della tesi

1.1 Introduzione

Definiamo *problema di ottimizzazione* la ricerca del massimo (o del minimo) di una funzione soggetta a vincoli; formalmente

$$\min(\max)\{f(x), x \in X\},$$

con f funzione definita su X a valori reali. Tramite un'opportuna modellizzazione molte delle decisioni che ci si trova ad affrontare, nella vita privata così come nelle realtà aziendali, rientrano in questa definizione.

A causa della loro diffusione ed importanza, nel corso degli anni sono stati sviluppati innumerevoli algoritmi per risolvere problemi di ottimizzazione. La necessità di questa varietà nasce da due fattori: la struttura stessa del problema, e le necessità dell'utente.

Come ci si può immaginare, è molto vantaggioso avere metodi diversi che sfruttino le varie proprietà della funzione f e dei vincoli che definiscono X: se ad esempio la struttura è lineare (sia la funzione che i vincoli), parleremo di metodi per la risoluzione di Programmazione Lineare (PL). All'interno di questa classe poi è noto come il vincolo di interezza sulle variabili (PLI) complichi il problema, richiedendo quindi tecniche più complesse per una risoluzione efficiente.

Nel caso in cui il problema da trattare sia *grande* (numero elevato di variabili, di vincoli, ...) e/o *difficile* (f non differenziabile, vincolo di interezza sulle variabili, ...) è possibile che ottenere una soluzione sia impossibile, o comunque richieda più tempo di quanto l'utente abbia a disposizione. Per questo esistono metodi che, in varia misura, sacrificano la precisione del risultato producendo un'approssimazione della soluzione per risparmiare in termini di tempo.

Esistono casi in cui la precisione è un fattore preponderante; se pensiamo ad esempio ad una missione spaziale, dove un errore infinitesimo potrebbe portare una sonda ad un impatto rovinoso o a perdersi nello spazio, è evidente come il tempo impiegato sia un fattore "secondario", e quindi l'utente sarà disposto a spendere molto più tempo per avere una soluzione anche solo leggermente più precisa. Tuttavia nella maggior parte dei casi reali non c'è bisogno di estremi, e si utilizzano quindi algoritmi che abbiano un buon trade-off tra velocità ed accuratezza.

Può capitare in applicazioni concrete che il numero di variabili utilizzate nel modello sia talmente elevato da renderne complicata l'enumerazione, ancora prima del calcolo della soluzione. Una delle tecniche utilizzate quando si affrontano problemi molto grandi è la cosiddetta *generazione di colonne* (o *Column Generation*), nella quale solo un numero ridotto del totale di variabili (e quindi di colonne del problema) viene utilizzato per creare una versione parziale (il *Master Problem*) del problema di partenza. Una sua soluzione viene poi utilizzata per determinare nuove variabili (colonne) il cui ingresso nel sottoinsieme potrebbe portare ad un miglioramento della soluzione stessa, generando così un metodo iterativo.

In questo lavoro di tesi abbiamo analizzato il comportamento di alcuni dei più diffusi solutori di problemi di ottimizzazione sui Master Problem generati dall'algoritmo per il *Crew* (o *Bus Driver*) *Scheduling* sviluppato da un'azienda italiana, la M.A.I.O.R. S.r.l. (MAIOR).

Per quanto il codice della MAIOR sia in continuo sviluppo, il solver di tipo *Bundle* che si occupa di questo aspetto dell'algoritmo risale al 2005. Tenendo conto dei continui sviluppi della Ricerca Operativa, è naturale chiedersi se questo sia tuttora valido o se sia possibile ottenere risultati migliori: analizzeremo quindi un certo numero di solutori, cercando di rendere ciascuno di loro (Bundle incluso) il più competitivo

possibile.

Segue una presentazione dei metodi selezionati. Al termine di questa vedremo la modellizzazione del problema affrontato dall'azienda, la realizzazione dell'algoritmo adibito al calcolo e dove si colloca la soluzione di problemi di ottimizzazione al suo interno. Nella descrizione eviteremo di dilungarci in dettagli troppo tecnici o riguardanti aspetti che non sono affini alle tematiche trattate.

Per ora ci limitiamo ad evidenziarne le caratteristiche principali dei problemi in esame: sono problemi di PL in cui la funzione obiettivo è non differenziabile e *decomponibile* come somma di funzioni più semplici.

Come già accennato, nel calcolo di una soluzione spesso è presente un trade-off tra la precisione della stessa ed il tempo impiegato a trovarla; i solver sono presentati in ordine di accuratezza decrescente, partendo quindi dai solutori lineari "esatti" e procedendo verso algoritmi sempre più rapidi ma inesatti.

Quindi anche se il solutore attualmente utilizzato è di tipo Bundle, questa classe di metodi verrà introdotta solo dopo quelli di tipo *Simplesso* e *Barrier*.

1.2 Simplesso

Ideato da George Dantzig nel 1947, il Simplesso è stato uno dei primi algoritmi sviluppati per la soluzione di problemi di PL. Sappiamo che sotto opportune ipotesi la soluzione ottima di un problema (se finita) è un vertice del poliedro che delimita la regione ammissibile. L'algoritmo del simplesso considera l'insieme dei vertici del problema primale e, ad ogni passo, cerca un vertice tra quelli adiacenti che migliori il valore della funzione obiettivo. Senza entrare troppo nel dettaglio, introduciamo alcune definizioni.

Se il problema è individuato dalla matrice $A \in \mathbb{R}^{m \times n}$, indichiamo con il termine *base* un qualsiasi sottoinsieme di indici B tale che la sottomatrice A_B sia di rango n; si può dimostrare che se un punto $\bar{x}$ è un vertice, allora esiste almeno una base tale che $\bar{x} = A_B^{-1} b_B$ (base associata). Viceversa, data una base possiamo associare ad essa un vertice utilizzando la stessa formula.

Indicando con $I(\bar{x})$ l'insieme di indici di A che $\bar{x}$ soddisfa con vincoli di uguaglianza, si ha che $|I(\bar{x})| \geqslant n$; un qualsiasi sottoinsieme di cardinalità n sarà una base associata a $\bar{x}$. Le situazioni possibili sono due: se $|I(\bar{x})| = n$ il vertice viene detto *non degenere*, mentre sarà *degenere* se $|I(\bar{x})| > n$ (e quindi esistono più basi ad esso associate).

Ponendo di avere in partenza un vertice $\bar{x}$ ed una base ad esso associata, l'algoritmo esegue iterativamente i seguenti passi:

1. Verifica l'ottimalità del vertice attuale (tramite il duale del problema). Se è ottimo termina, altrimenti individua una *direzione di crescita* ξ. Se il vertice è non degenere, ξ è *ammissibile* (ossia ci permette di migliorare la soluzione senza uscire dal poliedro). Se il vertice è degenere, la direzione potrebbe essere *non ammissibile*, nel qual caso non si sposta dal vincolo.

2. Calcola il massimo spostamento possibile nella direzione ξ. Questo può essere nullo (caso degenere), infinito (il problema è illimitato) o finito positivo, nel qual caso ha trovato un vertice adiacente che migliora la soluzione.

3. Viene aggiornata la base facendo entrare un indice k ed uscire un vincolo h. Questo corrisponde ad un cambio di vertice nel caso in cui si abbia una direzione di crescita ammissibile finita e non nulla, e ad un cambio di base (ma non di vertice) nel caso in cui la direzione sia non ammissibile.

Alcune osservazioni:

- Per semplicità abbiamo supposto di avere in input un vertice ed una base ad esso associato; in generale è necessaria una fase di inizializzazione in cui viene determinata una tale coppia.

- Per far sì che nel caso degenere le varie basi vengano visitate al più una volta sono necessarie delle regole dette di *anticiclo* che garantiscano di non entrare in un ciclo che blocchi l'algoritmo.

- Ad ogni passo è necessario calcolare l'inversa della matrice di base A_B e poi con essa calcolare prodotti scalari. Questi passaggi in generale hanno un costo di

$O(n^3)$ e $O(n^2)$ rispettivamente, tuttavia questo può essere ridotto considerando che spesso le matrici sono molto sparse e che quando cambia la base, B' ha un solo indice diverso rispetto a B e quindi A'^{-1}_B può essere calcolata a partire da A^{-1}_B.

- Dato un problema in forma standard, possiamo applicare il simplesso ad esso od al suo duale, determinando così due algoritmi: il **Simplesso Primale** ed il **Simplesso Duale**.

Vista l'importanza del Simplesso, una sua descrizione più dettagliata può essere trovata in qualsiasi libro di testo che tratti di ricerca operativa; rimandiamo ad esempio a pag.64 degli appunti disponibili al link [1].

1.3 Barrier

Con *Barrier* od *Interior Point* si indica una classe di problemi che, al contrario dei metodi di tipo simplesso, si muovono *all'interno* della zona ammissibile (anziché lungo i bordi), ed utilizzano una *barriera* per tenere sotto controllo questi spostamenti.

In particolare descriviamo un algoritmo di tipo *primal-dual path following*, il quale modifica iterativamente soluzione primale e duale finché le condizioni di ottimalità non sono "sufficientemente" soddisfatte. L'idea fondamentale dell'algoritmo è quella di applicare il metodo di Newton ad un particolare insieme di equazioni non lineari, per arrivare alla soluzione del problema di ottimizzazione. Queste equazioni si ricavano da opportune manipolazioni del problema di partenza, volte ad eliminare i vincoli di disuguaglianza. Vediamo brevemente i passaggi.

1. Senza perdita di generalità, possiamo considerare il problema ed il suo duale in questa forma (dove z rappresentano le variabili di scarto):

(P)	$\max\ cx$	(D)	$\min\ yb$
	$Ax = b$		$yA - z = c$
	$x \geqslant 0$		$z \geqslant 0$

2. Le uniche disuguaglianze presenti nei due problemi sono le condizioni di non negatività delle variabili. Partiti da un punto interno alla regione ammissibile, si vuole costruire una barriera che impedisca alle variabili di raggiungere il bordo: in questo modo non ci sarà più bisogno dei vincoli sul segno. Prendendo in considerazione $x_j = 0$ ad esempio, aggiungendo il termine $\log(x_j)$ alla funzione obiettivo del primale il suo valore decresce senza limiti quando x_j si avvicina a 0, impedendo quindi alla funzione di avvicinarsi a tale valore. Tuttavia in questo modo non ci avvicineremo mai abbastanza alla soluzione se almeno una delle variabili ha valore 0, cosa che accade spesso. Aggiungiamo allora un parametro μ che bilanci il contributo della vera funzione obiettivo con quello del termine barriera, ottenendo

(P)	max	$cx + \mu \sum_{j=1}^{n} \log(x_j)$	(D)	min	$yb - \mu \sum_{j=1}^{n} \log(z_j)$
		$Ax = b$			$yA - z = c$

   Effettuata questa modifica, i due nuovi problemi hanno funzioni obiettivo non lineari (e regione ammissibile strettamente minore del problema di partenza) e possono essere risolti ricorrendo ai moltiplicatori di Lagrange (per $\mu > 0$ fissato). Al decrescere di μ, le soluzioni tendono a quelle del problema originale.

3. Possiamo quindi usare il metodo di Newton per risolvere le equazioni non lineari che compongono le condizioni di ottimalità per un μ fissato. In questo modo si trovano delle direzioni ammissibili per le variabili, e dopo aver calcolato il massimo spostamento possibile viene restituito un nuovo punto dal quale poter ripartire. Piuttosto che ripetere questo passaggio e trovare l'ottimo per il valore di μ dato, è più efficiente diminuire il valore di μ al passo successivo. Il nuovo valore può essere calcolato a partire dal gap primale-duale.

4. Quando il gap è sufficientemente piccolo, l'algoritmo termina.

Una descrizione più approfondita di questo particolare metodo interior point può essere trovata al link [2].

1.4 Clp e Cplex

Per poter utilizzare i tre solutori appena descritti (i due tipi di Simplesso ed il Barrier), sono stati considerati due software diversi: il *Clp* ed il *Cplex*. Il primo è rilasciato come codice open source sotto licenza pubblica dalla COIN-OR (COmputational INfrastructure for Operations Research, [8]), mentre il secondo è commercializzato dalla IBM ([9]).

In realtà oltre a questi tre (a cui spesso ci riferiremo usando la notazione (P), (D) e (B) rispettivamente) comuni ad entrambi i codici, studieremo anche altre due opzioni potenzialmente utili offerte dal Cplex: il *Net* (N) e l'*Automatic* (A).

Net: Il Net è un adattamento dell'algoritmo del Simplesso Primale, applicato in modo da sfruttare la struttura particolare di problemi di flusso su reti. Alcune proprietà di questa classe si traducono in modifiche che permettono di velocizzare questo solutore in modo sostanziale.

L'entità del miglioramento ottenuto sostituendo l'utilizzo di un metodo di tipo Simplesso con il Net, seguito poi da Primale o Duale, dipende dalla struttura del problema: se questo è interamente di tipo flusso su reti, il Net è estremamente più efficiente. Nel caso in cui invece questa proprietà sia presente solo in una parte del problema, l'algoritmo cerca in una fase iniziale di individuarla e calcolare una prima soluzione ignorando il resto; questa viene poi utilizzata come punto di partenza da un algoritmo di tipo Simplesso. Quanto questo secondo approccio sia efficiente dipende dalla dimensione relativa del sottoproblema, ed è difficile da determinare a priori.

Rientrando i nostri problemi nel secondo caso, è possibile che si riveli competitivo; per questo è stato incluso nell'analisi.

Automatic: È il setting base del Cplex. Non si tratta di un vero e proprio algoritmo, ma di un meta-algoritmo che sceglie quale degli altri utilizzare. Se lanciato in parallelo, fa girare il Simplesso Duale su di un thread, il Primale su un secondo ed il Barrier (anch'esso in parallelo) su tutti gli altri a disposizione. Tuttavia non siamo

interessati a questa possibilità, avendo deciso di lanciare tutti gli algoritmi in modo sequenziale.

In questo modo la sua utilità viene di gran lunga ridotta, in quanto la versione sequenziale utilizza praticamente sempre il Simplesso Duale. Nel caso in cui si abbia già una base di partenza, sceglie se utilizzare il Primale od il Duale a seconda dell'ammissibilità.

1.5 Bundle

Con il termine Bundle ci si riferisce ad un'ampia classe di metodi ritenuti tra i più efficienti per l'ottimizzazione convessa non differenziabile. Nonostante la loro validità, spesso si preferisce ricorrere a metodi più datati e potenzialmente più lenti, come ad esempio il Subgradiente. Questa scelta è dettata principalmente da due fattori: la scarsa disponibilità di codici affidabili del Bundle, dovuta all'elevata difficoltà implementativa di questi metodi rispetto alla semplicità del Subgradiente, e la dipendenza del Bundle da un gran numero di parametri, che ne rendono arduo l'adattamento a problemi specifici.

Consideriamo il generico problema

$$\inf\{f(x), x \in X\},$$

con $f : \mathbb{R}^n \to \mathbb{R}$ una funzione a valori finiti convessa e non differenziabile, $X \subseteq \mathbb{R}^n$ chiuso e convesso.

È un'assunzione standard per la maggior parte degli algoritmi di ottimizzazione non differenziabile che la funzione in esame f sia nota solo tramite un *oracolo* (o *Black Box*) che, dato un x appartenente allo spazio delle soluzioni ammissibili, restituisce il valore $f(x)$ ed un elemento $z \in \partial f(x_i)$ del subdifferenziale di f in x.

L'idea principale dei metodi di tipo Bundle è quella di campionare lo spazio ammissibile in un insieme finito di punti $\bar{X} \subset X$, collezionando per ogni $x_i \in \bar{X}$ il valore della funzione $f(x_i)$ e lo specifico $z_i \in \partial f(x_i)$ restituito dall'oracolo. Il corrispondente *bundle* (insieme, gruppo) $\mathcal{B} = \{(x_i, f(x_i), z_i) : x_i \in \bar{X}\}$ viene utilizzato

per costruire un modello $f_{\mathcal{B}}$ di f; tipicamente viene impiegato il modello dei *piani di taglio*:

$$f_{\mathcal{B}}(x) = \max\{f(x_i) + z_i(x - x_i) : (x_i, f(x_i), z_i) \in \mathcal{B}\} \leqslant f(x)$$

Questo viene quindi utilizzato per trovare il prossimo punto da prendere in considerazione. La scelta più semplice sarebbe quella di considerare il minimo di $f_{\mathcal{B}}$:

$$\inf_x\{f_{\mathcal{B}}(x)\} = \inf_{v,x}\{v : v \geqslant f(x_i) + z_i(x - x_i) : (x_i, f(x_i), z_i) \in \mathcal{B}\}$$

Tuttavia è noto che questo sia un approccio inefficiente, in quanto la sequenza di punti generata tende ad oscillare incontrollabilmente anche quando ci si avvicina al minimo di f. Per ridurre questa oscillazione, si sceglie un *punto corrente* $\bar{x}$, di solito il punto migliore trovato finora, e si introduce un *termine di stabilizzazione* D_t (una funzione convessa chiusa) che impedisca al prossimo punto di allontanarsi troppo dal precedente. In questo termine, $t > 0$ è il *parametro di prossimità* che regola la "forza" di D_t, definendo implicitamente una regione di confidenza contenente la direzione d. Ponendo ad esempio $D_t = 1/2t||d||_2^2$ si ottiene il nuovo problema stabilizzato

$$\inf_d \{\ f_{\mathcal{B}}(\bar{x} + d) + \frac{1}{2t}||d||_2^2\ \}. \tag{1.1}$$

Indichiamo con $\tilde{d}$ la soluzione ottima e con $x_+ = \bar{x} + \tilde{d}$ il nuovo punto ottenuto. Se il modello è abbastanza preciso, $\tilde{d}$ sarà una direzione di discesa e quindi $f(x_+) < f(\bar{x})$. Se questa differenza viene reputata apprezzabile, si considera come nuovo punto corrente x_+ (*Serious Step*); altrimenti si mantiene $\bar{x}$ e si aggiunge la terna $(x_+, f(x_+), z_+)$ a $\mathcal{B}$ per migliorare il modello $f_{\mathcal{B}}$ (*Null Step*).

Durante questo processo t può essere tenuto fisso, ma è noto come aggiornarlo passo per passo affinché rifletta quanto "buono" è ritenuto il modello $f_{\mathcal{B}}$ di f sia fondamentale per avere un metodo efficiente.

Per risolvere 1.1 si potrebbe usare un qualsiasi solutore standard di problemi quadratici, tuttavia all'interno del Bundle è necessario risolvere un gran numero di questi problemi che differiscono tra loro "di poco", ad esempio per l'aggiunta di una terna a $\mathcal{B}$ o per una variazione del parametro t. Quindi sfruttare un solutore

specifico che tenga in memoria informazioni riguardo la chiamata precedente porta solitamente ad un algoritmo più rapido. Ci riferiremo a questa componente del Bundle come *MPSolver*.

In [10, 11, 12, 13] sono presentati diversi approcci al Bundle, sia dal punto di vista teorico che pratico, oltre a particolari sull'implementazione e la discussione di alcune varianti.

1.6 Bundle Disaggregato

Essendo stati sviluppati da diversi anni, esiste una vasta letteratura riguardante adattamenti di algoritmi di tipo Sugradiente a particolari strutture dei problemi in esame; molte di queste idee possono essere applicate anche ai metodi di tipo Bundle date le affinità concettuali. In particolare, siamo interessati a sfruttare il fatto che la nostra funzione di riferimento f sia esprimibile come somma, e che quindi un generico problema può essere riscritto come

$$\inf\{f(x) = \sum_{k \in \mathcal{K}} f^k(x), x \in X\}. \tag{1.2}$$

Chiaramente per poter applicare un metodo Bundle a questa riscrittura è necessario disporre di oracoli diversi, uno per ogni componente $k \in \mathcal{K}$, che restituiscano i valori $f^k(x_i)$ e $z_i^k \in \partial f^k(x_i)$ in modo tale che $f(x_i) = \sum_{k \in \mathcal{K}} f^k(x_i)$ e $z_i = \sum_{k \in \mathcal{K}} z_i^k$. La prima idea che viene in mente è quella di tenere $|\mathcal{K}|$ *bundle disaggregati* $\mathcal{B}^k$ per costruire k diversi modelli $f_{\mathcal{B}}^k$, uno per ogni componente, in modo tale che $f_{\mathcal{B}}$ sia un modello migliore (per lo stesso insieme di punti $\bar{X}$) di quello costruito nel caso del Bundle standard.

Diversi studi (ad esempio [3]) mostrano come in generale questo *modello disaggregato* possa aumentare la velocità di convergenza, tanto da compensare l'incremento del costo dato dall'aumento di dimensione dei corrispondenti problemi master. Purtroppo nel nostro caso questa via non è praticabile, visto che il numero di variabili dei nostri problemi è spesso molto alto. Quello che abbiamo provato è un passo inter-

medio: *aggregare* le varie f^k in un numero minore di funzioni somma, decomponendo quindi f come:

$$f(x) = \sum_{j \in \mathcal{J}} f^j(x), \quad f^j(x) = \sum_{k \in \mathcal{K}^j} f^k(x), \quad \bigsqcup_{j \in \mathcal{J}} \mathcal{K}^j = \mathcal{K}$$

In questo modo speriamo di poter sfruttare in parte la velocità del caso interamente disaggregato, mantenendo però le dimensioni dei master problem in un range che ci permetta di gestirli senza troppe difficoltà. C'è da considerare che in questo modo viene introdotto un ulteriore elemento al già complesso Bundle: la scelta dell'aggregazione, ossia come partizionare $\mathcal{K}$. Il modo più semplice è quello di considerare le f^j in ordine, e quindi

$$\mathcal{K}_1 = \{1, 2, \ldots, m\}, \quad \mathcal{K}_2 = \{m+1, m+2, \ldots, 2m\}, \quad \ldots$$

dove $m = |\mathcal{K}|/|\mathcal{J}|$. Alternativamente, le f^j possono essere distribuite nei vari gruppi di modo casuale; è anche possibile esistano delle proprietà che "accomunino" alcune delle funzioni e definiscano implicitamente una divisione più efficiente, e quindi corrispondente ad un algoritmo più rapido o preciso.

Una volta fissato il metodo di aggregazione delle componenti, sarà anche decisivo determinare il *livello* di disaggregazione, ovvero in quante componenti dividere $\mathcal{K}$; questo è evidentemente un valore critico, e difficile da individuare a priori.

1.7 Subgradiente

Il metodo del *Subgradiente* è una generalizzazione del ben noto metodo del *Gradiente* alla risoluzione di problemi di ottimizzazione non differenziabili. Dato un generico problema come descritto in precedenza, analogamente al caso del Bundle abbiamo bisogno di un oracolo che restituisca i valori di $f(x)$ e $z \in \partial f(x)$ per ogni x appartenente ad X.

Il metodo del Subgradiente utilizza la semplice ricorrenza

$$\hat{x}_{i+1} = x_i - \nu_i z_i, \qquad x_{i+1} = P_X(\hat{x}_{i+1}),$$

dove P indica la proiezione ortogonale su X e $\nu_i \in \mathbb{R}^+$ è l'*ampiezza del passo* o *stepsize*. Per quanto riguarda la scelta di questo valore, basta l'imposizione di poche regole per avere una sequenza $\{f_i = f(x_i)\}$ che risolva asintoticamente 1.2, ovvero tale che $\liminf_{i\to\infty} f_i = f_\infty = f_*$.

Purtroppo l'ordine di convergenza di questi metodi non è affatto buona: sono necessarie $O(1/\epsilon^2)$ iterazioni per risolvere un problema con un errore assoluto ϵ. Questo comporta che non è possibile sperare di ottenere soluzioni con un buon grado di precisione; tuttavia l'indipendenza del numero di passi dalle dimensioni del problema rende questi metodi particolarmente indicati nel caso si vogliano risolvere metodi molto grandi e con un grado di precisione relativamente basso.

Di nuovo analogamente al caso del Subgradiente, un problema fondamentale di questi metodi è l'oscillazione dei punti che vengono generati; si introducono quindi delle tecniche dette di *deflessione* il cui compito è minimizzare questo aspetto. Ad esempio si può modificare la regola di ricorrenza in

$$\hat{x}_{i+1} = x_i - \nu_i d_i,$$

dove la direzione d_i è una qualche combinazione lineare tra z_i ed il subgradiente del passo precedente z_{i-1}.

I risultati che utilizzeremo per quanto riguarda questo solutore (e la sua versione Incrementale) sono tratti da un lavoro di A. Frangioni, E. Gorgone, B. Gendron in fase di pubblicazione [14].

1.8 Subgradiente Incrementale

Il metodo del *Subgradiente Incrementale* è in un certo senso la versione disaggregata del Subgradiente, ed è quindi applicabile se il problema è della forma 1.2 ed abbiamo un oracolo per ogni componente.

L'idea è quella di utilizzare in alcuni passaggi solo una componente f^k invece dell'intera funzione f. Questo vuol dire che nel calcolo della direzione sostituiamo il subgradiente "intero" z_i con la sua componente z_i^k per un qualche $k \in \mathcal{K}$.

Sperabilmente, una sequenza di iterazioni *incrementali* lungo singole componenti del subgradiente è altrettanto efficace di una sequenza di iterazioni *intera*, mentre il costo della valutazione della funzione è ridotto di $|\mathcal{K}|$. Tuttavia per assicurare la convergenza è necessario utilizzare regolarmente l'intera funzione f.

Inoltre è possibile che lo spostamento lungo una singola direzione z_i^k allontani il punto corrente dall'ottimo, per questo nelle iterazioni incrementali viene ridotta la lunghezza del passo rispetto alle iterazioni intere. A causa di possibilità, unita al fatto che il costo di calcolare i vari z_i^k non viene ridotto in un'iterazione incrementale, non è detto che il Subgradiente Incrementale sia sempre applicabile.

In letteratura questo metodo è stato applicato con successo a vari problemi di grandi dimensioni, inclusi alcuni provenienti da rilassamenti lagrangiani (come nel nostro caso), perciò è possibile dia buoni risultati.

Capitolo 2

Ambito della tesi

2.1 La MAIOR

La M.A.I.O.R. S.r.l. è un'azienda di Lucca che si occupa dal 1989, anno della sua fondazione, di ottimizzazione matematica per la gestione delle risorse nel settore del trasporto pubblico nelle sue varie forme (ferroviario, aereo, su gomma).

Fondata da membri del gruppo di Ricerca Operativa del Dipartimento di Informatica dell'Università di Pisa, ha sempre mantenuto un rapporto attivo con l'ateneo, come dimostra anche questo lavoro di tesi. Il lavoro di ricercatori e studenti ha aiutato l'azienda a rendere gli strumenti utilizzati sempre più efficaci nel risolvere al meglio le specifiche esigenze dei vari problemi proposti.

Nello specifico, la MAIOR si occupa della formazione di turni del personale (Crew Scheduling o Bus Driver, rispettivamente CS o BD) e di veicoli (*Vehicle Scheduling*, VS) negli ambiti di trasporto aereo, ferroviario e collettivo su gomma. Lo scopo nei vari casi è essenzialmente quello di coprire tutte le attività programmate dal servizio con dei turni (sequenze di attività) cercando di minimizzare i costi per l'azienda utente. Per quanto il modello sia simile, ciascuno di questi settori ha difficoltà proprie: una grande varietà di normative aziendali che vincolano l'ammissibilità di un turno nel caso del trasporto su gomma; il variare dei vincoli su base giornaliera nella programmazione mensile nel trasporto aereo, che aumenta la difficoltà del modello;

una maggiore rigidità strutturale e la necessità di dover includere doppie coperture, ossia unire due veicoli per coprire attività in orari di punta, in ambito ferroviario.

La descrizione che segue è tratta dalla relazione interna della MAIOR [4] e dall'articolo [5].

2.2 Modellizzazione del problema

In letteratura, i problemi di CS e VS vengono solitamente formalizzati utilizzando due differenti modelli di Programmazione Lineare Intera (PLI): *flusso di costo minimo su grafo* e *Set Partitioning*.

In entrambi i casi, è utile introdurre il cosiddetto *grafo di compatibilità* $G = (N, A)$. Qui N rappresenta l'insieme dei nodi ed è della forma $N = T \cup \{O, D\}$, dove T contiene un elemento per ogni attività da coprire mentre O e D rappresentano due nodi deposito, non corrispondenti ad alcuna attività reale quanto a due depositi fittizi in cui si pone che i turni abbiano rispettivamente inizio e fine.

A rappresenta invece l'insieme degli archi: dati $(i, j) \in T \times T$, $(i, j) \in A$ se e solo se le due attività i e j possono essere eseguite consecutivamente in un turno. Quindi $(O, j) \in A$ se e solo se un turno può iniziare in j e $(i, D) \in A$ se e solo se un turno può terminare in i.

In questo modo, un turno può essere visto come un cammino che congiunge O a D su tale grafo.

Introducendo per ogni arco $(i, j) \in A$ una variabile booleana $x_{i,j}$ che assume valore 1 se le attività i e j vengono svolte in sequenza e 0 altrimenti, ed assumendo che il costo di un turno sia esprimibile come somma dei costi $c_{i,j}$ delle singole attività che lo compongono, otteniamo il modello di flusso di costo minimo su grafo:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}$$

$$\sum_{j:(i,j)\in A} x_{ij} = 1 \qquad i \in T \qquad (2.1)$$

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = 0 \qquad i \in T \qquad (2.2)$$

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in A$$

dove il vincolo (2.1) impone la copertura di tutte le attività mentre il (2.2) modella la conservazione di flusso per ciascun nodo. Per tale modello sono stati sviluppati numerosi algoritmi, alcuni dei quali risultano particolarmente efficienti poiché si possono utilizzare i numerosi risultati noti relativi ai problemi di flusso su grafi.

Se indichiamo invece con P l'insieme dei cammini ammissibili su G e definiamo per ogni turno (cammino) $p \in P$, di costo c_p, una variabile booleana x_p che ha valore 1 se il turno p appartiene alla soluzione e 0 altrimenti, otteniamo il modello di Set Partitioning:

$$\min \sum_{p\in P} c_p x_p$$

$$\sum_{p\in P} a_{ip} x_p = 1 \qquad i \in N \qquad \text{(SP)}$$

$$x_p \in \{0,1\} \qquad \forall p \in P$$

dove i coefficienti booleani a_{ip} hanno valore 1 se se l'attività i appartiene al turno p, 0 altrimenti.

Per semplicità in questi modelli abbiamo omesso i cosiddetti *vincoli globali*, che pure svolgono un ruolo fondamentale. Con questo termine intendiamo dei vincoli aggiuntivi che variano a seconda dei casi presi in considerazione, ad esempio un

vincolo che limiti il numero massimo di turni in soluzione. Quando tali vincoli sono presenti, la MAIOR ne studia una descrizione lineare e li inserisce nel modello.

Inoltre eventuali vincoli dovuti a normative vigenti, come la durata massima della guida continuativa per un autista, sono impliciti, in quanto regolano l'ammissibilità dei cammini e quindi non sono indicati.

Di solito nei problemi di VS questi vincoli di normativa possono essere formalizzati in un modello di flusso su grafo; tuttavia ciò non accade per quelli di CS (spesso la possibilità di svolgere due attività in sequenza è condizionata dalle attività precedenti): si preferisce quindi utilizzare il più generale modello del Set Partitioning, dove qualsiasi regola di ammissibilità per un cammino può essere inclusa nel vincolo $p \in P$.

Purtroppo nei casi concreti il numero di cammini (e quindi turni) possibili è enorme, rendendone quindi impraticabile l'enumerazione completa. Per ovviare a questo problema, si ricorre alla Column Generation (CG).

2.3 Column Generation

Innanzitutto consideriamo il rilassato (P), ed il suo duale (D), del problema (SP):

$$\min \left\{ \sum_{p \in P} c_p x_p : \sum_{p \in P} a_{ip} x_p = 1 \quad i \in N,\ x_p \in [0,1] \quad p \in P \right\} \tag{P}$$

$$\max \left\{ \sum_{i \in N} \pi_i : c_p - \sum_{i \in N} \pi_i a_{ip} \geqslant 0,\ p \in P \right\} \tag{D}$$

Come abbiamo già accennato, il grande numero di variabili in (P) (rispettivamente vincoli in (D)) non permette nemmeno di scrivere esplicitamente i problemi. L'idea della CG è quella di generare un sottoinsieme $B \subset P$ di dimensioni ridotte, in modo tale da poter essere maneggiato con maggiore facilità. A partire da questo si considera il problema master (P_B), ossia la restrizione di (P) in cui sono presenti solo le colonne di B ($x_p = 0$ per $p \in P \backslash B$). Si ha che il suo duale (D_B) è un rilassamento di (D).

In questo modo possiamo risolvere (P_B) ed ottenere una soluzione ottima x^* e, allo stesso tempo, una soluzione ottima π^* di (D_B). Osserviamo che mentre x^* (completata con zeri) è sempre una soluzione ammissibile per (P), in generale π^* non lo è per (D). Tuttavia è facile verificare se esistono colonne (ossia cammini) in $P \backslash B$ che corrispondono a vincoli di (D) violati da π^*.

Definito infatti il *costo ridotto* di un cammino p come:

$$\bar{c}_p = c_p - \sum_{i \in N} \pi_i^* a_{ip},$$

si ha che questo valore è negativo se e solo se il vincolo duale corrispondente a p è violato.

Considerato un cammino π^* di costo minimo su G, se il costo ridotto è non negativo allora π^* è soluzione ammissibile (quindi ottima) anche per (D); altrimenti è possibile determinare una colonna la cui aggiunta all'insieme B può portare ad un miglioramento della soluzione di (P_B).

Iterando questo passaggio, si arriva infine ad una soluzione ottima per (P); questa sarà in generale non intera, ma come vedremo in seguito può essere usata come punto di partenza per trovare una soluzione ammissibile per (SP).

Questo procedimento presenta al momento dell'applicazione una difficoltà non indifferente: il cosiddetto *problema di pricing*, ossia la ricerca del cammino a costo ridotto minimo, è in generale un problema NP-completo. Questo è dovuto alla presenza dei vincoli dati dalle normative, che impongono ai turni il soddisfacimento di regole molto complesse (per questo la soluzione del problema di pricing è spesso uno dei passaggi più costosi dell'intero algoritmo). Solitamente questo viene risolto in modo euristico; tuttavia evitiamo di dilungarci in una descrizione esplicita del metodo utilizzato, in quanto il nostro interesse non risiede in questa componente quanto piuttosto nel solver.

2.4 Solver

In quanto problema di Programmazione Lineare (PL), il Master Problem potrebbe essere risolto direttamente tramite vari solutori disponibili (sia open source che commerciali). Questo approccio però è sconsigliabile a causa dalle dimensioni estremamente grandi che il problema può assumere nei casi presi in esame dalla MAIOR. La strategia utilizzata consiste quindi nel non approcciare (P_B), e considerare invece il *rilasciamento lagrangiano* dei vincoli di copertura (compresi i globali, se presenti):

$$\phi(\lambda) = \left\{ \sum_{p \in B} c_p x_p + \sum_{i=1}^{n} \lambda_i \left(1 - \sum_{p \in B} a_{ip} x_p \right) : x_p \in [0,1] \quad p \in B \right\} \tag{2.3}$$

ed il suo corrispondente *duale lagrangiano*:

$$\max \{ \phi(\lambda) : \lambda \in \mathbb{R}^n \} \tag{DL}$$

È noto che, nel caso lineare, il duale Lagrangiano (DL) è equivalente al duale lineare (D). La funzione $\phi(\lambda)$ è convessa ma in generale non è differenziabile; per questo nella risoluzione non possono essere applicati quegli algoritmi di ottimizzazione non lineare che richiedono una certa regolarità della funzione, tipicamente derivate prime (o seconde) continue. Per questo vengono utilizzati gli algoritmi di tipo Bundle descritti nel capitolo precedente.

Per evitare il rischio della divergenza, in una fase preliminare dell'algoritmo viene generato un insieme di turni che assicurino una copertura totale di tutti gli elementi. Ci si riferisce a questi turni con il nome di *tripper* e sono, nell'implementazione attuale, colonne che coprono una sola attività.

2.5 Fixing delle colonne

Una volta ottenuta una coppia di soluzioni ottime (x^*, π^*) per (P) e (D), l'algoritmo risolutivo MAIOR costruisce una soluzione intera in modo incrementale: si alternano fasi di *fixing* (fissaggio, si scelgono alcuni turni $p \in P$ e si fissa $x_p = 1$) e di risoluzione

del rilasciamento continuo rispetto al sottospazio delle variabili indotto dal fixing, fino a quando tutte le attività sono coperte.

Chiaramente la scelta di quali turni fissare è cruciale per l'efficacia dell'approccio.

Precedentemente la fase di fixing era guidata esclusivamente dalla soluzione duale π^*, ovvero i turni appartenenti a B al termine della CG venivano ordinati per costo ridotto crescente, ed i primi turni della lista entravano a far parte della soluzione discreta fino ad aver coperto una determinata percentuale di attività. Da alcuni anni invece, il criterio di scelta tiene conto anche della soluzione primale.

Il criterio dei costi ridotti viene ancora utilizzato per definire un sottoinsieme $\bar{B} \subset B$ di turni promettenti; all'interno di $\bar{B}$ però, la scelta viene compiuta utilizzando i valori della soluzione primale x^* , in quanto si considera che un turno con x_p^* "grande" (vicino a uno) ha più probabilità di far parte di una buona soluzione rispetto ad uno con x_p^* "piccolo" (vicino a zero).

Per quanto riguarda la modalità con cui si ottiene x^*, questa può essere prodotta direttamente dall'algoritmo Bundle che risolve (DL); tale soluzione però tende ad essere fortemente frazionaria.

Una strategia alternativa, che si dimostra spesso preferibile, è di utilizzare un solutore per la PL basato su tecniche di tipo Simplesso sul sottoproblema corrispondente all'insieme $\bar{B}$, che solitamente contiene comunque le colonne ottime per (P); dato che il problema ha dimensioni ridotte, e viene comunque risolto solamente una volta per ogni fixing (invece che ad ogni passo della CG), il costo computazionale di questa strategia è accettabile.

La soluzione ottima primale di base prodotta in questo modo è tipicamente meno frazionaria ed è spesso una migliore guida per la fase di fixing. Si noti che un simile effetto non si ottiene risolvendo il problema attraverso metodi di tipo Barrier, che producono soluzioni interne allo spazio e quindi fortemente frazionarie, in modo non dissimile da quelle prodotte dai metodi Bundle.

2.6 Sfixing e ripartenza

L'approccio descritto è euristico, quindi la soluzione determinata può, in generale, non essere ottima. Per questo l'algoritmo MAIOR produce più soluzioni intere ripetendo più volte l'intero processo; continuando ad esplorare la regione ammissibile discreta del problema, è possibile produrre soluzioni migliori.

Al termine del calcolo di una soluzione intera segue quindi una fase di *sfixing*, in cui una parte dei turni presenti nella soluzione intera viene rimosso da questa. La scelta di quali turni tenere e quali rimuovere per poter arrivare ad un miglioramento della soluzione è guidato dalle variabili duali "accumulate" durante il processo; nuovamente, non essendo interessati a questo particolare passaggio i dettagli sono omessi. In realtà si è osservato che la scelta delle colonne da sfissare basata sui costi ridotti non sia significativamente più efficace di una scelta casuale.

Terminato lo sfixing, l'algoritmo riparte dalla soluzione parziale e ne costruisce una nuova utilizzando un diverso insieme di colonne.

Capitolo 3

Salvataggio e caricamento istanze

Come mostrato nella sezione 2.3, per produrre una singola soluzione intera l'algoritmo MAIOR genera e risolve un certo numero (molto variabile, come si vedrà a fine capitolo) di Master Problem; usiamo il termine *istanza* per indicare ciascuno di essi.

L'analisi dei vari solutori su una singola istanza deve necessariamente essere preceduta da due fasi: una di *salvataggio* in cui i dati che la caratterizzano vengono collezionati, ed una di *caricamento* in cui questi vengono recuperati e passati ai vari solver.

Inoltre siamo interessati ad analizzare l'algoritmo di tipo Bundle già presente nel codice della MAIOR; per questo poniamo due obiettivi preliminari al lavoro di tesi vero e proprio:

- Modificare il codice preesistente aggiungendo la possibilità di salvare i dati delle istanze, funzione in origine non contemplata dalla MAIOR
- Scrivere un programma che legga i dati di una istanza e replichi in tutto e per tutto il comportamento del Bundle come eseguito all'interno dell'algoritmo. Spesso ci riferiremo a questo come *nuovo codice*.

Terminati questi passaggi, basterà sostituire il Bundle con uno degli altri solver per analizzarne l'andamento sulle varie istanze generate. L'analisi del codice è stato reso più facili dalle descrizioni presenti in [6], [7].

3.1 Implementazione MAIOR

Dovremo quindi analizzare il codice MAIOR su due livelli:

- la *struttura del codice* per sapere *dove* è necessario intervenire per aggiungere le funzioni e reperire le informazioni necessarie
- le *strutture dati* per capire *cosa* dobbiamo salvare. É evidente ad esempio che andranno salvate le righe e le colonne che individuano l'istanza, ma queste potrebbero essere memorizzate in modo non banale

Nel seguito verrà innanzitutto mostrata una panoramica della struttura generale del codice e delle classi principali per dare un'idea di come sono state implementate le varie sezioni descritte nel capitolo precedente e quando / come vengono generate / immagazzinate le informazioni che andranno salvate.

Quindi l'attenzione si sposterà verso la struttura delle classi di maggiore interesse, quelle contenenti i dati relativi ad istanze e solver.

Questa trattazione sarà necessariamente parziale, in quanto un'analisi approfondita della struttura del codice non rientra tra gli obiettivi di questo lavoro, ed inoltre non siamo interessati a gran parte di esso (generazione di colonne, gestione soluzioni intere, ...); per una descrizione dettagliata di tutte le componenti si rimanda alla documentazione interna della MAIOR.

Osserviamo che nel codice sono già presenti delle interfacce per Clp e Cplex; queste sono state implementate in un primo momento per effettuare un confronto tra Simplesso e Bundle, ed inoltre il Clp viene utilizzato per eseguire la strategia alternativa discussa nella sezione 2.5.

Tuttavia, per motivi che saranno chiariti nel prossimo capitolo, piuttosto che riutilizzare queste componenti nella successiva fase di analisi dei metodi relativi si è preferito ignorarle e riscriverne di nuove.

3.1.1 Struttura del codice

ConstrainedSetPartitioning Rappresenta la classe di più alto livello, quella che coordina il lavoro di tutte le altre per ottenere una o più soluzioni al problema.

In breve:

1. Istanzia le altre classi coinvolte nel calcolo
2. Calcola il lower bound tramite una chiamata al *MainModule*
3. Calcola un certo numero di soluzioni intere, per ciascuna:
   (a) Se la soluzione attuale non è la prima, chiama il *RigeneraPezzi* per effettuare lo sfissaggio di alcune colonne della precedente
   (b) Fino a quando ci sono task ancora scoperti, alterna chiamate al MainModule (per il ciclo di stabilizzazione generazione-solver) al fissaggio di alcune colonne tramite *IncrTurniInSol*
   (c) Notifica la soluzione intera trovata

Il numero di soluzioni intere calcolate di default è 40, tuttavia questa quantità è elevata per i nostri scopi. Infatti il numero di istanze generato per ogni singolo caso sarebbe troppo grande per poter essere analizzato efficacemente, senza di fatto apportare alcunché di significativo.

Salvo casi particolari, considereremo quindi le istanze appartenenti al calcolo delle prime 5 soluzioni intere.

MainModule La tecnica di Column Generation descritta nella sezione 2.3 richiede chiamate alternate al generatore di colonne (*GeneratoreColonne*) ed al solver del master problem (*MasterProblemSolver*). Questa classe si occupa proprio di coordinare queste chiamate, nel seguente modo:

1. Istanzia le classi MasterProblemSolver e GeneratoreColonne
2. Nel caso in cui sia necessario calcolare il lower bound, genera un primo insieme di colonne con costo ridotto qualsiasi, in quanto non c'è ancora una soluzione duale con cui calcolare i costi ridotti
3. Fino a quando non è soddisfatta una delle condizioni di arresto:

(a) Se il numero di colonne disponibili è maggiore di una certa soglia, elimina alcuni turni tra i peggiori.

(b) Se necessario effettua il *backtrack*, ovvero rimuove dalla soluzione discreta delle colonne tra quelle inserite nel fissaggio precedente.

(c) Chiama il GeneraColonne per generare delle colonne a costo ridotto negativo

(d) Chiama il MasterProblemSolver per calcolare la soluzione del problema attuale

(e) Se il ciclo è stabile per più di due iterazioni, termina

Le condizioni di arresto possono essere varie: oltre alla stabilità del ciclo, abbiamo ad esempio il raggiungimento del numero massimo di passi stabilito o l'interruzione da parte dell'utente.

MasterProblemSolver Permette di utilizzare un solver per risolvere il sottoproblema attuale. Oltre che per interfacciarsi con il Bundle (tramite la classe *BundleMaior2005* che descriveremo meglio in seguito), può essere impiegata per calcolare la soluzione tramite Clp e Cplex.

GeneratoreColonne Il compito principale di questa classe è la generazione di colonne a costo ridotto negativo, secondo la soluzione duale corrente, che vanno ad aggiungersi a quelle già presenti nel *GestoreColonne*.

Fixer (IncrTurniInSol/RigeneraPezzi) Un oggetto di tipo Fixer ha il compito di modificare l'insieme delle colonne facenti parte della soluzione. In particolare può aggiungerne o rimuoverne, a seconda che stiamo parlando di fixing (IncrTurniInSol) o sfixing (RigeneraPezzi), come visto nelle sezioni 2.5 e 2.6. In particolare:

- IncrTurniInSol ha le funzionalità di *fixer di primo livello*, ovvero selezionare dall'insieme delle colonne attive un sottoinsieme da aggiungere a quelle presenti nella soluzione corrente, e si occupa del backtracking

- RigeneraPezzi ha le funzionalità di *fixer di secondo livello*, ossia costruire una soluzione parziale di ripartenza dall'ultima soluzione intera trovata rimuovendo alcune colonne

3.1.2 Classi di interesse: Istanze

Le classi che stiamo per approfondire sono divise in due gruppi, a seconda che le informazioni contenute siano relative alle istanze o al solver.

Un problema è identificato dalle sue righe e dalle sue colonne; in particolare cominciamo da queste ultime in quanto contengono gran parte della sua struttura.

Colonna La classe base è la *Colonna*. Un elemento di questo tipo ha tre campi fondamentali:

- Il costo
- La copertura dei task
- I coefficienti dei vincoli globali

La situazione di task e vincoli globali è molto diversa. Per quanto riguarda i primi infatti dobbiamo solo memorizzare se la colonna copre un task o meno. Dato che una singola colonna in genere ne copre un numero ridotto rispetto al totale, è molto più efficiente memorizzare l'indice di questi piuttosto che un lungo vettore booleano composto principalmente di zeri. La struttura dei vincoli globali invece (se presenti), cambia di problema in problema ed è più generica. Quindi abbiamo bisogno di tener conto dei singoli coefficienti, che appartengono a $\mathbb{Q}$.

Ci aspettiamo però che queste non siano tutte le informazioni riguardanti le colonne: ad esempio esisterà una qualche forma di ordinamento per mantenerle in memoria ed avranno quindi un indice che le identifichi.

GestoreColonne Di questo e molto altro si occupa il *GestoreColonne*. Ogni volta che una colonna viene generata, questa viene aggiunta ad un *buffer* e le viene associato un indice che la identifichi in modo univoco. Ad ogni calcolo di una soluzione

intera corrisponde un gran numero di generazioni di colonne e, considerando che di soluzioni ne vengono calcolate diverse, questo porta a problemi di memoria ma soprattutto di convergenza del solver. Per ridurre l'insieme B delle colonne utilizzabili, è utile la seguente considerazione: le soluzioni dei Master Problem delle prime iterazioni possono essere molto distanti da quelle delle iterazioni successive. Questo comporta dei costi ridotti diversi nel tempo, e quindi le colonne generate in una fase iniziale potrebbero rivelarsi successivamente inutili. Definiamo allora tre attributi di una colonna; diremo che questa è

- *Attiva* se fa parte dell'insieme B, cioè è utilizzata nel Master Problem
- *Fissata* se è stata selezionata come facente parte della soluzione discreta dal Set Partitioning
- *Bloccata* se non può essere cancellata dalla memoria

Nella descrizione dell'approccio, avevamo visto che il problema viene risolto in modo incrementale alternando fasi di fixing e di generazione; le colonne fissate sono appunto quelle che fanno già parte della soluzione corrente.

Inoltre, questa classe si occupa anche della rimozione di alcune colonne dall'insieme: ogni volta che una colonna viene fissata dal *Fixer*, tutte le colonne che hanno almeno una riga coperta in comune (il *conflict set*) devono essere rimosse. Infatti la natura del problema richiede che una riga sia coperta da una ed una sola colonna della soluzione, quindi queste devono essere necessariamente escluse.

Quando viene effettuata la rimozione del conflict set le colonne interessate vengono cancellate e di esse non rimane traccia in memoria, a meno che non siano state precedentemente bloccate.

Come abbiamo visto, un oggetto di tipo colonna non ha un campo che ne memorizzi lo stato nel buffer: è compito del GestoreColonne tenere traccia di questo attributo. Nel buffer le colonne attive e fissate sono tenute mescolate, ma questa classe fornisce funzioni per separare ed esplorare i due insiemi.

GestoreCostiRidotti L'ultimo dato riguardante le colonne è il loro costo ridotto. Il *GestoreCostiRidotti* si occupa di calcolarlo a partire dalle variabili duali ogni

volta sia necessario e di mantenere in memoria l'attuale valore. Questo è facilmente accessibile utilizzando la funzione apposita.

Tripper Per quanto siano colonne, i tripper sono tenuti in memoria nel GeneratoreColonne piuttosto che nel GestoreColonne. Questi vengono generati una volta sola nella fase iniziale del codice e restano costanti nel corso dell'algoritmo. Osserviamo che nella costruzione del Master Problem non vengono utilizzati tutti i tripper, ma solo quelli relativi ai task scoperti; ci si riferisce a questi come tripper *attivi*.

Se necessari vengono utilizzati anche i tripper dei vincoli globali, tuttavia questi non vengono tenuti esplicitamente in memoria ed il loro costo non necessita di essere calcolato in quanto questo viene posto ad un valore fisso predeterminato.

Per quanto riguarda le righe invece, la loro descrizione è molto più semplice. Questo perché le colonne provvedono già a gran parte della struttura del problema, le uniche informazioni che mancano per descrivere una riga sono lower ed upper bound. Quindi non abbiamo una struttura dati di tipo *Riga*, tutte le informazioni sono immagazzinate nel *GestoreRighe*.

GestoreRighe É la classe che si occupa della gestione delle righe, in essa sono quindi memorizzate le informazioni riguardanti i vincoli del problema. Sarebbe una classe "semplice" se non fosse per la presenza dei vincoli globali che ne complicano la struttura; infatti

- lower ed upper bound delle righe relative ai task sono sempre 1 (esprimono i vincoli di uguaglianza), mentre come abbiamo già visto nel caso dei vincoli globali non c'è una forma generale
- per tenere in memoria la copertura dei task basta un vettore di bool (1 coperto, 0 scoperto) mentre per i vincoli globali è necessario tenere in conto il valore dei coefficienti relativi alle colonne già fissate, memorizzati nell'*Rhs*. Tramite questo è possibile ottenere rapidamente il valore di lower ed upper bound aggiornati.

Ogniqualvolta una colonna viene fissata, il bool corrispondente ai task appena coperti viene modificato per tenerne conto. In questo modo è possibile sapere istantaneamente se una riga è coperta o meno, e quindi ad esempio quali tripper sono attivi e quali no.

3.1.3 Classi di interesse: Solver

Come già detto il Bundle (e le classi ad esso collegate, come quella che implementa l'oracolo *PhiOracle*) è di fatto indipendente dal codice MAIOR. A fare da interfaccia tra questo solver ed il resto del codice è la classe *BundleMaior2005*.

BundleMaior2005 Classe derivata del MasterProblemSolver, si occupa di lanciare il Bundle dopo aver impostato tutti i parametri necessari al funzionamento dello stesso. La maggior parte di questi sono fissi, nel senso che non sono dipendenti dal problema o dall'istanza ma sono posti sempre pari a dei valori di default in fase di inizializzazione. Vedere il significato di ciascuno di questi sarebbe di scarso interesse in questo momento, tuttavia vi torneremo in seguito quando ci concentreremo su questo solutore. Infatti i vari valori sono rimasti invariati negli ultimi 10 anni mentre i problemi da affrontare sono cambiati, quindi una nuova operazione di tuning potrebbe portare a dei miglioramenti nelle prestazioni.

I parametri che variano sono invece:

- Il numero massimo di iterazioni eseguite dal Bundle prima di fermarsi. Questo valore decide l'accuratezza del solver e può assumere tre valori in base alla precisione necessaria: minima (150), media (250) o massima (2000).
- La soluzione duale dell'istanza precedente (è un vettore di zeri nel caso in cui sia la prima generata).

Una volta scritte le funzioni adibite al salvataggio delle informazioni contenute nelle classi appena descritte, resta da decidere dove inserirle all'interno del codice. L'inizio della funzione di BundleMaior2005 che si occupa del calcolo (la *BundleMaior2005::Calcola*) è un candidato perfetto: salvando la situazione immediatamente

precedente alla chiamata del Bundle, siamo sicuri di poter ripetere esattamente l'esecuzione dello stesso. Due eccezioni, i Tripper ed i *VincoliIncasinati*, saranno discussi insieme ad altre osservazioni nella prossima sezione.

3.2 Osservazioni

Per quanto riguarda la gestione delle strutture dati, abbiamo deciso di mantenere nel nuovo codice il più possibile dell'implementazione sviluppata dalla MAIOR piuttosto che costruire un nuovo modello.

Questo vuol dire che abbiamo già le strutture dati e le funzioni adatte alla loro gestione, comodamente divise in classi. Tuttavia quelle utili sono in alcuni casi fortemente legate ad altre che invece non saranno presenti: oltre alla selezione dunque, è necessario un lavoro aggiuntivo di separazione.

Per restare fedeli al codice originario, abbiamo tenuto conto nelle fasi di salvataggio e caricamento anche di parametri non strettamente necessari riguardanti l'inizializzazione e la struttura delle classi. Ad esempio il GestoreColonne ha un parametro *MaxNum* il cui valore viene utilizzato per inizializzare la dimensione del buffer; questo viene poi eventualmente ampliato nel caso in cui vengano generate più di MaxNum colonne. Questo parametro non sarebbe necessario, in quanto sappiamo esattamente il numero di colonne con cui dobbiamo lavorare.

Per quanto importanti, questo passaggi non sono molto interessanti da vedere nel dettaglio. Nel seguito vengono riportate delle osservazioni riguardanti alcuni aspetti visti in precedenza.

- *Tripper*: Vista la natura dei tripper, al momento del salvataggio viene solo stampato il loro costo; una funzione apposita ha il compito di ricostruirli. Inoltre venendo generati all'inizio del codice e rimanendo costanti, la funzione che si occupa del salvataggio viene invocata una sola volta all'inizio di *CSP3LoopMaior::Calcola* (la funzione principale della classe ConstrainedSetPartitioning) e non ripetuta per ogni istanza dello stesso caso.

- *GeneratoreColonne*: Non avendo bisogno del generatore, questo non è presente nel nuovo codice. La lista dei tripper è stata quindi spostata ed aggiunta come elemento del GestoreRighe.

- *GestoreColonne*: Ogni volta che viene eliminata una colonna, nel buffer resta un "buco" che viene riempito solo qualora sia esaurita la dimensione del buffer (prima di incrementarne la dimensione). Non essendo necessario il mantenimento di questa struttura, al momento del caricamento le colonne vengono "compresse" all'inizio del buffer; questo implica che l'indice non viene salvato in quanto cambia dopo il caricamento. Viene solo mantenuto l'ordine ed ovviamente lo status; nello specifico basta un bool per tenere in memoria l'eventuale fissaggio in quanto non è rilevante se una colonna sia bloccata o meno (non ne vengono cancellate). Una volta caricata, una colonna viene attivata e, se necessario, fissata tramite funzioni apposite. Queste si occupano anche di modificare di conseguenza i parametri che tengono nota delle cardinalità dei due insiemi (*fNumColonneAttive* e *fNumColonneFissate*).

- *Inizializzazione classi*: Bisogna porre particolare attenzione a come vengono inizializzati alcuni valori. Come abbiamo appena visto, *fNumColonneAttive* va posto a 0; sarà poi compito della funzione che attiva le colonne modificarlo. Se venisse posto pari al suo valore nel momento del salvataggio, assumerebbe un valore errato (doppio) al termine del caricamento e genererebbe successivamente errori.

- *BundleMaior2005*: Volendo mantenere la struttura del codice MAIOR, è utile tenere questa classe come interfaccia con il Bundle. I parametri non variabili precedentemente menzionati vengono impostati da due funzioni che ritroviamo anche nel nuovo codice, quindi non è necessario il loro salvataggio e caricamento. Per il corretto funzionamento di questa classe è necessario però salvare una particolare modellizzazione dei vincoli globali; questa modellizzazione è mantenuta nella classe VincoliIncasinati, il cui comportamento in fase di salvataggio è analogo al caso dei Tripper (ovvero questo avviene una sola volta)

Fase		N° Istanze
Lower Bound		h
SolIntera_1	Fix_1	$h_{1,1}$
	Fix_2	$h_{1,2}$
	...	...
	Fix_n_1	h_{1,n_1}
SolIntera_2	Fix_1	$h_{2,1}$
	Fix_2	$h_{2,2}$
	...	...
	Fix_n_2	h_{2,n_2}

Tabella 3.1: Distribuzione delle istanze in un generico problema

3.3 Istanze

Alla struttura del codice aggiungiamo un'analisi della distribuzione delle istanze all'interno del calcolo delle soluzioni intere ed alcune delle loro caratteristiche principali.

3.3.1 Distribuzione

Come sappiamo, il calcolo di una soluzione intera è costituito da una serie di fissaggi seguiti ciascuno da un certo numero di generazioni di colonne. Ad ogni generazione corrisponde un problema che viene passato al solver, ed abbiamo indicato con il termine istanza ciascuno di questi.

Possiamo quindi dire che un'istanza *appartiene* ad un fissaggio, intendendo che la generazione corrispondente fa parte del ciclo di stabilizzazione successivo ad esso. Quando effettuiamo il salvataggio di tutte le istanze di un caso, è importante mantenere questa divisione: la posizione relativa ci fornirà informazioni riguardo la conformazione.

Possiamo rappresentare il salvataggio delle istanze di un caso come nella tabella 3.1, e possiamo riferirci ad una sua istanza come *la i-esima del j-esimo fissaggio della k-esima soluzione intera* piuttosto che *la n-esima generata*.

3.3.2 Caratteristiche

Vediamo adesso quali sono le caratteristiche principali di un'istanza e cosa possiamo dire a riguardo a seconda della posizione all'interno del calcolo.

Dimensione La dimensione di un'istanza è data da due quantità: il numero di righe m ed il numero di colonne n. É noto come questi due parametri influiscono in modo diverso sulla complessità dei vari algoritmi, ad esempio nel caso del simplesso applicato ad una matrice di dimensioni $m \times n$ avevamo individuato passaggi di complessità $O(n^3)$ e $O(n^2)$, indipendenti quindi dal numero di righe. Tuttavia a volte utilizzeremo la dimensione intesa come $n \cdot m$; per quanto sia una quantità imprecisa può essere utile per fare una prima divisione tra istanze "grandi" ed istanze "piccole".

Densità Oltre alla dimensione, può far comodo sapere la quantità di elementi diversi da zero nella matrice. La densità è una quantità molto variabile, che nei casi da noi presi in esame varia in un range che da 0.001 a 1. E non è direttamente correlata alla dimensione, nel senso che istanze con un numero simile di righe e colonne ma appartenenti a casi diversi possono avere densità molto diverse.

Il valore limite della densità, 1, viene raggiunto in alcune istanze particolari ("terminali", nel senso che se presenti sono l'ultima istanza del calcolo della soluzione intera corrispondente) con una sola riga scoperta ed una sola colonna, il tripper ad essa associata, il cui unico elemento è diverso da zero.

Per quanto riguarda la localizzazione, una prima divisione è tra calcolo del lower bound e calcolo di una soluzione intera. Infatti questi vengono gestiti in modo diverso e le istanze corrispondenti hanno strutture distinte.

Lower Bound Nel calcolo del lower bound non si eseguono fissaggi, ma solo generazioni di colonne. Questo vuol dire che la soluzione non viene calcolata in modo incrementale, e le istanze che ne fanno parte hanno quindi un gran numero di righe scoperte (tutte) ed un numero elevato (spesso crescente) di colonne attive. Queste

tendono perciò ad essere le più grandi e problematiche, nel senso che a seconda del caso possono richiedere un gran quantitativo di tempo.

Soluzione Intera Se invece ci troviamo nel calcolo di una soluzione intera, mano a mano che andiamo avanti con i fissaggi ci sono sempre meno righe scoperte, e quindi i problemi tendono a diventare più piccoli e di più semplice risoluzione. All'interno di uno stesso fissaggio invece, istanze successive hanno un maggior numero di colonne (a causa delle generazioni).

Possiamo anche dire che in generale, ma non sempre, il calcolo della prima soluzione intera tende ad essere più problematico. Dalla seconda in poi abbiamo già delle colonne fissate (quelle lasciate dalla fase di sfixing) e quindi il problema che dobbiamo risolvere in modo incrementale è più piccolo in partenza.

3.4 Casi test

La MAIOR ha fornito per questo studio un certo numero di casi test da analizzare, tutti appartenenti al caso di trasporto su gomma. Sono divisi in tre categorie, denominate *BDSA* (Bus Driver Scheduling Algorithm), *VBDSA* (Vehicle Bus Driving Scheduling Algorithm) e *VSA* (Vehicle Scheduling Algorithm). Riportiamo nelle tabelle 3.2, 3.3 e 3.4 cardinalità e distribuzione delle istanze che vengono affrontate nel calcolo del lower bound e delle prime cinque (salvo casi particolari) soluzioni intere nei vari test. Qui non è mostrata esplicitamente la divisione nei vari fissaggi; per un caso particolare si rimanda alla tabella 4.4 del capitolo successivo.

Inoltre per motivi di privacy i nomi delle città interessate sono stati rimossi e sostituiti con nomi fittizi.

Il numero di istanze in un singolo gruppo (lower bound o soluzione intera) fornisce un'idea abbastanza accurata della difficoltà del calcolo affrontata dall'algoritmo: una gran quantità di istanze vuol dire un gran numero di fissaggi/generazioni e quindi individua spesso (ma non sempre) un caso computazionalmente complicato.

Problema	LB	SI_1	SI_2	SI_3	SI_4	SI_5
BDSA_1	70	322	84	186	16	295
BDSA_2*	70	1173	207	121	121	147
BDSA_a	10	40	28	3	23	43
BDSA_b1*	33	136	16	49	38	219
BDSA_b2	18	47	42	74	40	62
BDSA_b3	17	21	8	22	5	11
BDSA_c1	27	34	11	9	6	5
BDSA_c2	28	31	6	6	6	14
BDSA_cc*	70	535	/	/	/	/

Tabella 3.2: Distribuzione delle istanze nei BDSA

Problema	LB	SI_1	SI_2	SI_3	SI_4	SI_5
VBDSA_3	33	49	51	100	29	55
VBDSA_a1	23	30	42	6	6	8
VBDSA_a2	41	129	135	155	130	115
VBDSA_c1	24	51	65	45	46	10
VBDSA_c2	21	98	63	59	70	72
VBDSA_f	27	36	4	12	5	9
VBDSA_g	30	69	11	16	9	8
VBDSA_cc*	117	278	/	/	/	/

Tabella 3.3: Distribuzione delle istanze nei VBDSA

Problema	LB	SI_1	SI_2	SI_3	SI_4	SI_5
VSA_cc1*	70	393	458	/	/	/
VSA_cc2*	70	172	124	105	83	131

Tabella 3.4: Distribuzione delle istanze nei VSA

3.4.1 Casi critici

I casi contrassegnati da un asterisco corrispondono ai più ardui tra quelli forniti. Gli elementi di criticità possono essere diversi, in generale sono riconducibili ad un numero elevato di task o di vincoli globali. Per quanto riguarda i casi di tipo VSA, ne abbiamo a disposizione solo due e sono entrambi segnati come critici a causa del fatto che in generale l'algoritmo non è ritenuto molto valido su questa tipologia di problemi.

Come verrà mostrato meglio in seguito, l'analisi di questi casi critici è resa difficile dal fatto che le istanze che li compongono, oltre ad essere molto numerose, sono solitamente molto grandi, e di conseguenza la loro risoluzione richiede una quantità di tempo elevata. Questo potrebbe tradursi nella necessità di selezionare un sottoinsieme su cui restringersi, i cui criteri di scelta saranno presentati se e quando ne sorgerà il bisogno. Per questo motivo in alcuni casi ci siamo anche limitati a considerare un numero minore di soluzioni intere.

Capitolo 4

Simplesso e Barrier

Prima di iniziare l'analisi dei vari solutori, è fondamentale fare la seguente osservazione: *un miglioramento nel solutore potrebbe non risultare in un miglioramento complessivo*. Ovvero, cambiando il solutore la soluzione intera trovata in seguito ai vari cicli di fixing/generazione/solver potrebbe essere peggiore, od il calcolo potrebbe impiegare più tempo rispetto a prima, nonostante il solver restituisca soluzioni più precise. Questa incertezza è dovuta alla presenza del generatore di colonne; tuttavia non ci cureremo dell'impatto globale quanto piuttosto di valutare la performance di diversi solutori su singoli istanze.

Chiaramente, nel caso in cui dovessimo trovare un metodo preferibile al Bundle attuale, sarebbe interessarne verificarne l'applicabilità all'interno dell'algoritmo nella sua interezza.

Con le varie istanze debitamente salvate e la possibilità di caricarle (e quindi risolvere ogni singolo problema di PL che viene generato), siamo pronti ad affrontare uno studio sistematico degli algoritmi descritti nel Capitolo 1. In questa analisi saremo aiutati in modo decisivo dall'avere a disposizione un super computer situato a La Spezia (controllato in remoto) per poter fare i calcoli.

Seguendo lo stesso ordine della presentazione, iniziamo da Simplesso (Primale e Duale) e Barrier.

Quando parliamo di analizzare Simplesso e Barrier, dobbiamo tenere conto del fatto che le implementazioni dello stesso algoritmo date da Clp e Cplex possono

essere diverse, e quindi avere prestazioni molto differenti.

Inoltre come abbiamo già menzionato il Cplex è, al contrario del Clp, a pagamento. Quindi ci aspettiamo una performance migliore da parte dei solutori del primo rispetto a quelli del secondo; da questo segue che non è significativo, ad esempio, confrontare tra di loro le due versioni (potenzialmente molto diverse) del Barrier.

Il motivo per cui stiamo considerando questi due diversi programmi di ottimizzazione, anche se sappiamo già che uno sarà migliore dell'altro, risiede nel fatto che l'utilizzo di codice con licenza a pagamento (come il Cplex) all'interno di un software ne aumenta ovviamente il prezzo di mercato; se non si traduce in un miglioramento significativo delle prestazioni, questa operazione potrebbe risultare sconveniente, portando ad un software dal prezzo non competitivo.

Piuttosto considereremo i solutori separati nelle due categorie e li valuteremo contestualmente; poniamo come obiettivo quello di selezionarne due che "rappresentino" Clp e Cplex. In particolare, cercheremo il rappresentante nella forma di un *meta-algoritmo* che alterni l'uso di vari solver (dello stesso software) su istanze diverse per migliorare le prestazioni totali. Una volta fatto questo, analizzeremo l'entità dello scarto; starà poi all'utente valutare l'utilità o meno dell'investire nel Cplex.

Premettiamo ai dati sperimentali una breve descrizione della classe utilizzata per comunicare con i due programmi di ottimizzazione.

4.1 Implementazione Osi

L'*OSI* (Open Solver Interface) è un'interfaccia libera messa a disposizione dalla COIN-OR, gli stessi sviluppatori del CLP. Tuttavia questa permette di utilizzare anche altri programmi, sia open source che commerciali, a patto di avere i file necessari (librerie, headers). Chiaramente il Cplex ha una sua interfaccia, ed anche ben sviluppata, ma ci sono diversi vantaggi nell'utilizzare la stessa: possiamo ad esempio costruire il modello una sola volta e passarlo ad entrambi i programmi, invece di ripetere due volte la stessa operazione utilizzando i diversi strumenti delle rispettive interfacce.

È principalmente per questo motivo che abbiamo preferito riscrivere questa sezione piuttosto che usare le due interfacce già presenti nel codice MAIOR. Per apprezzare alcune delle funzionalità dell'OSI, ne mostriamo le funzioni principali.

CoinModel Innanzitutto, è necessario disporre di un modello del problema da risolvere che sia comprensibile ai solutori; questo può essere caricato da un file o costruito a partire da righe e colonne. Visto che le informazioni necessarie sono immagazzinate in apposite classi (GestoreRighe e GestoreColonne), questa seconda via è chiaramente preferibile. Basta allora creare un oggetto vuoto di tipo *CoinModel*, scorrere righe e colonne memorizzate ed aggiungerle una ad una tramite le apposite funzioni (*addRow*, *addCol*). In questa fase possono essere specificati anche altri dettagli, come la natura del problema (massimo o minimo).

OsiSolverInterface Una volta completato il modello, questo può essere caricato direttamente nell'interfaccia specifica del programma, nel nostro caso l'*OsiClpSolverInterface* e l'*OsiCpxSolverInterface*. Queste permettono anche di impostare altre specifiche, come la quantità di informazioni che vogliamo i solver restituiscano in un eventuale log; per quanto riguarda le funzioni messe a disposizione dalle due classi, possiamo avere

- funzioni in comune come *setCoinModel* e *initialSolve*, rispettivamente per caricare il CoinModel nell'interfaccia e per avviare il calcolo dell'istanza
- diverse formulazioni per la stessa funzione, ad esempio la selezione del solutore da utilizzare nel Clp è affidata a *setSolveType*, mentre nel caso del Cplex è integrata nella generica *CPXsetintparam* che può essere utilizzata per impostare un gran numero di parametri.

4.2 Test

Le soluzioni restituite dai vari solutori sulla stessa istanza sono "uguali", nel senso che le differenze sono talmente minime da non essere rilevanti (solitamente inferiori

a 1e-10). In teoria può accadere che in casi fortemente degeneri queste siano distinte, tuttavia abbiamo verificato ciò non accadesse nei casi in esame e possiamo quindi assumere che la soluzione prodotta sia la stessa; in questo modo dobbiamo analizzare solo i tempi: un metodo sarà definito *migliore* semplicemente se è quello che impiega meno.

Possiamo già fare delle considerazioni di carattere teorico:

- È difficile predire l'andamento degli algoritmi di tipo Simplesso, nel senso che pur avendo complessità esponenziale al caso pessimo (in cui si debbano esplorare tutti i vertici, di cardinalità esponenziale rispetto alla dimensione del problema), è raro che questo si verifichi.

- Gli algoritmi di tipo Barrier sono stati sviluppati in modo da avere complessità polinomiale anche nel caso pessimo proprio per offrire un'alternativa nell'eventualità che il Simplesso (generalmente migliore) fallisca. Per quanto l'andamento di Simplesso e Barrier dipenda dal caso e dalla specifica istanza, al livello teorico ci aspettiamo che quest'ultimo possa essere migliore su problemi molto grandi e sparsi.

- Vista la struttura del nostro problema, è lecito aspettarsi prestazioni migliori da parte del Net rispetto al Simplesso Primale.

Per studiare in modo più efficace l'andamento dei solutori, non sono stati considerati direttamente i tempi da loro impiegati quanto piuttosto i *tempi relativi*: fissato il programma (Clp o Cplex), consideriamo il tempo migliore impiegato dai suoi algoritmi e dividiamo i vari valori per questa quantità. Separiamo poi i risultati in quattro classi: indicando con t_{rel} il tempo relativo, diremo che un algoritmo è

- *Buono*, se $t_{rel} \leqslant 1.05$
- *Medio*, se $1.05 < t_{rel} \leqslant 1.4$
- *Cattivo*, se $1.4 < t_{rel} < 2$
- *Pessimo*, se $t_{rel} \geqslant 2$

4.2.1 Selezione di istanze

Come analisi preliminare, i solver sono stati lanciati su delle istanze dei casi critici per decidere se fosse necessaria una qualche forma di selezione. Nelle tabelle 4.1, 4.2 riportiamo i risultati (in secondi) divisi tra Clp e Cplex per alcune di queste istanze.

P	D	B
67.89	68.57	266.81
88.32	**55.2**	332.86
119.83	**21.36**	409.04
254.44	223.71	**122.92**

Tabella 4.1: Tempi necessari al Clp su alcune istanze critiche

A	P	D	N	B
22.04	147.07	21.72	20.92	**16.09**
21.16	132.09	21.35	**19.06**	**19.06**
8.91	210.00	9.17	**8.82**	18.35
148.35	213.28	151.70	138.98	**8.6**

Tabella 4.2: Tempi necessari al Cplex su alcune istanze critiche

Per lanciare tutti i metodi uno dopo l'altro sull'ultima istanza, abbiamo bisogno di circa 1260 secondi (21 minuti); il problema è che di queste non ce ne sono poche, al contrario: sono proprio i casi con più istanze ad avere quelle che impiegano più tempo. In questo modo il calcolo di tutte le istanze appartenenti al solo lower bound di un singolo problema potrebbe impiegare anche una giornata intera.

Questo renderebbe l'analisi molto lenta e quindi, con la situazione complessiva sotto mano, abbiamo proceduto caso per caso a selezionare un sottoinsieme di riferimento. In generale, per far sì che le istanze restanti siano il più significative possibile, queste sono state scelte in modo distribuito; ad esempio all'interno di un fissaggio con 20 istanze, ne terremo una iniziale, una in mezzo ed una finale (del tipo 2, 11, 18). Nella tabella 4.3 è riportato un confronto tra il numero di istanze tenuto ed il totale. Oltre ai casi contrassegnati come critici, abbiamo considerato anche BDSA_2 e BDSA_b1.

Caso	LB	SI_1	SI_2	SI_3	SI_4	SI_5
BDSA_2	70	1173	207	121	121	147
	12	122	150	68	77	78
BDSA_b1	33	136	16	49	38	219
	8	95	16	49	38	138
BDSA_cc	70	535	/	/	/	/
	9	106	/	/	/	/
VBDSA_cc	117	278	/	/	/	/
	9	119	/	/	/	/
VSA_cc1	70	393	458	/	/	/
	16	67	72	/	/	/
VSA_cc2	70	172	124	105	83	131
	70	172	124	105	83	131

Tabella 4.3: Confronto tra il numero di istanze tenuto ed il totale sui casi grandi

A titolo esemplificativo, vediamo nello specifico le scelte effettuate nel caso di VBDSA_cc. Nella prima colonna della tabella 4.4 è indicato il numero di istanze appartenenti o al calcolo del lower bound o alle varie fasi di fissaggio della prima soluzione intera (l'unica salvata), mentre nella seconda sono riportate le posizioni di quelle tenute.

Fase	N° Istanze	Tenute
Lower Bound	117	4,10,14,20,26,46,56,74,112
Fix_1	48	2,21,36,48
Fix_2	29	2,13,21,27
Fix_3	19	1,8,17
Fix_4	20	5,10,12,20
Fix_5	7	1,4,6
Fix_6	20	1,6,13,19
Fix_7	9	1,7,9
Fix_8	5	1,3,5
Fix_9	12	2,7,11
Fix_10	3	1,3
Fix_11	3	Tutte
Fix_12	3	Tutte
Fix_13	22	3,10,16,21
Fix_14	3	Tutte
Fix_15	4	1,3,4
Fix_16	4	1,2,4
Fix_17-Fix_35	...	Tutte

Tabella 4.4: Distribuzione delle istanze tenute nel caso VBDSA_cc

4.2.2 Prima divisione

In totale abbiamo circa 5700 istanze, e di ciascuna abbiamo i tempi migliori dei due programmi con i vari tempi relativi. Queste sono state inizialmente ordinate per dimensione (intesa come prodotto tra il numero di righe e di colonne) e divise a formare cinque gruppi. Per quanto questa questa quantità non sia una misura esatta della difficoltà di un'istanza, una volta effettuata la divisione possiamo considerarle in ordine di "importanza" decrescente: quelle appartenenti al Gruppo 1 impiegano in media un tempo maggiore rispetto a quelle del Gruppo 2, e così via.

Per ogni algoritmo ed ogni gruppo, è stato calcolato il numero di volte in cui il suo tempo relativo appartiene alle diverse classi definite in precedenza (Buono, Medio, ecc..., riportato nei grafici), la media (geometrica) e la *deviazione standard relativa* (o *coefficiente di variazione*), indicata con DSR.

Per un dato campione, questa viene definita come il rapporto tra la deviazione standard σ ed il valore assoluto della media aritmetica μ:

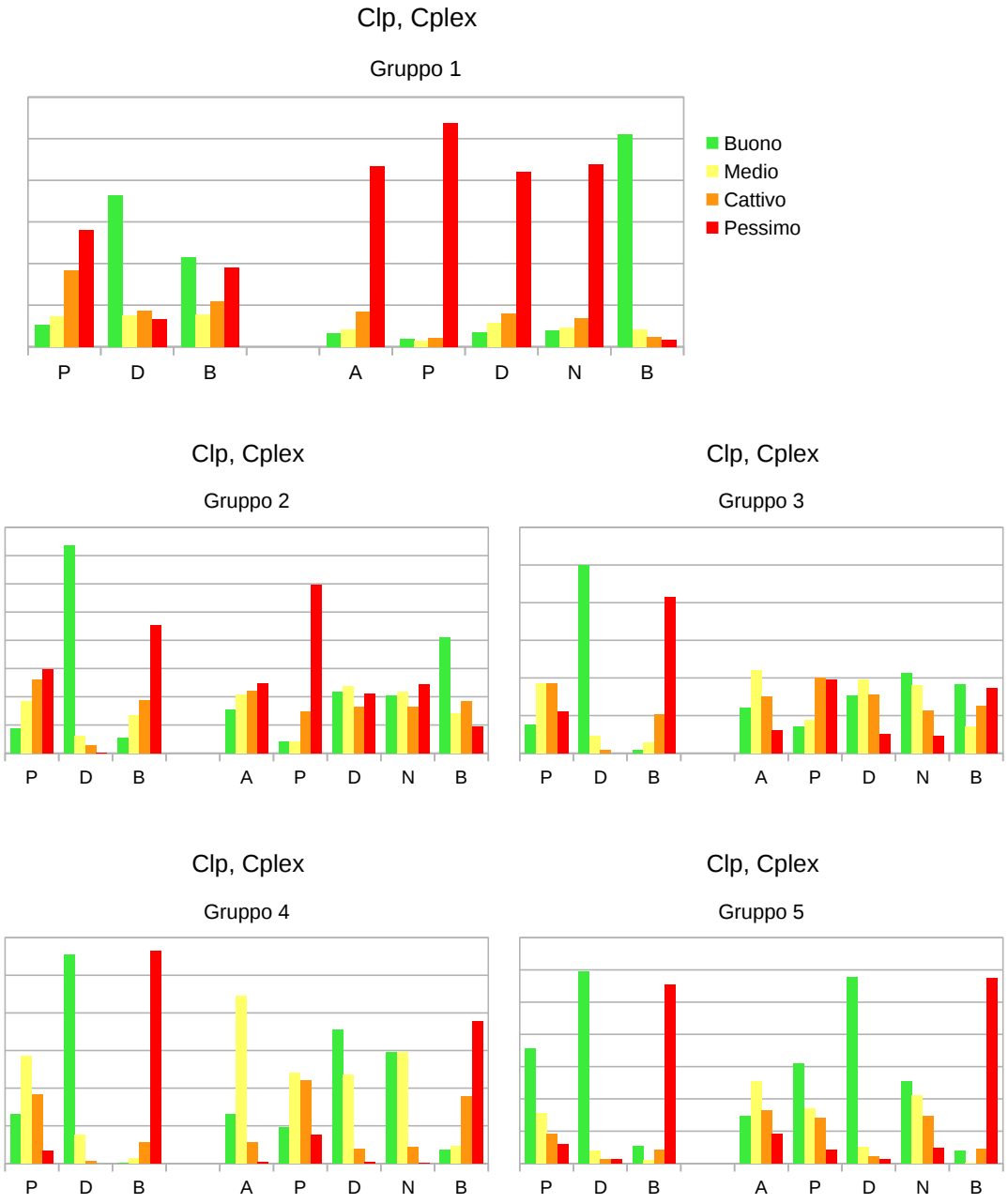
$$\sigma^* = \frac{\sigma}{|\mu|}$$

Abbiamo ritenuto più significativo utilizzare questa quantità piuttosto che la varianza per esprimere la dispersione dei valori.

Riportiamo adesso le descrizioni ed i risultati dei singoli gruppi.

	Intervallo dimensione	Cardinalità
Gruppo 1	dim$> 5e06$	1182
Gruppo 2	$1.5e06 <$ dim $\leqslant 5e06$	830
Gruppo 3	$3e05 <$ dim $\leqslant 1.5e06$	1110
Gruppo 4	$4e04 <$ dim $\leqslant 3e05$	1269
Gruppo 5	dim $\leqslant 4e04$	1322

Tabella 4.5: Disposizione delle istanze nei gruppi per dimensione



Clp : Già dai grafici è evidente come il Duale sia in generale il miglior algoritmo del Clp nei diversi gruppi, riportiamo comunque i vari valori di media e deviazione

Clp	Media			DSR		
	P	D	B	P	D	B
Gruppo 1	2.02	1.23	1.74	0.936	0.479	1.249
Gruppo 2	1.80	1.03	2.40	0.551	0.136	0.702
Gruppo 3	1.51	1.02	2.93	0.438	0.090	0.541
Gruppo 4	1.32	1.02	3.21	0.273	0.071	0.392
Gruppo 5	1.19	1.03	2.65	0.262	0.150	0.387

Tabella 4.6: Media e DSR dei tempi relativi degli algoritmi del Clp

nella tabella 4.6. Il Barrier è nettamente peggiore in quattro dei cinque gruppi; solo nel restante (il Gruppo 1) riesce ad essere migliore del Primale in media, ma presenta una deviazione circa doppia.

Cplex : Il caso del Cplex è invece più interessante, sia a causa del maggior numero di algoritmi in esame sia per la mancanza di un metodo predominante. In particolare:

- Dai grafici è evidente come il Primale sia in difficoltà sui primi due gruppi; in tabella possiamo vedere come la media parta da valori molto alti e vada a scendere fino a raggiungere valori competitivi solo nel Gruppo 5, che è tuttavia il meno interessante a causa della piccola dimensione dei problemi (e quindi della rapidità di risoluzione).

- Al contrario il Barrier si comporta molto bene su problemi di grandi dimensioni: è di gran lunga il migliore nel primo gruppo e resta buono nel secondo. Al diminuire di questo quantità tuttavia, diminuisce anche la sua competitività. Nel Gruppo 3 alterna valori buoni a valori pessimi, mentre negli ultimi due risulta il peggiore.

- Automatic, Duale e Net hanno comportamenti simili. Al livello di media, vediamo come il Duale sia il migliore dei tre in quattro dei cinque gruppi e sia superato dal Net solo nel Gruppo 3.

Da questi risultati risulta che l'Automatic può essere scartato: ha un andamento simile al Duale ma peggiore in media. Inoltre utilizzando sempre algoritmi di tipo

Cplex	Media					DSR				
	A	P	D	N	B	A	P	D	N	B
Gruppo 1	3.29	5.79	3.12	3.18	1.06	0.721	0.842	0.746	0.705	0.433
Gruppo 2	1.66	3.09	1.54	1.56	1.29	0.549	0.795	0.561	0.484	0.476
Gruppo 3	1.37	1.81	1.34	1.27	1.60	0.382	0.597	0.359	0.378	0.529
Gruppo 4	1.18	1.43	1.10	1.12	2.13	0.153	0.369	0.159	0.151	0.381
Gruppo 5	1.35	1.22	1.04	1.25	2.56	0.276	0.247	0.162	0.243	0.318

Tabella 4.7: Media e DSR dei tempi relativi degli algoritmi del Cplex

Simplesso non è in grado di fornire risultati buoni quando è il Barrier ad essere più efficace.

Da questa prima analisi, risulta che nel caso del Clp si può semplicemente utilizzare sempre il Duale. Questo non è necessariamente il metodo migliore, in quanto c'è da considerare che il Barrier presenta risultati positivi proprio sulle istanze più significative. Tuttavia non siamo riusciti a trovare una regola soddisfacente che ci permettesse di decidere quale algoritmo utilizzare, cosa che invece accade nel caso del Cplex.

4.3 Il meta-algoritmo del Cplex

Considerando i dati di cui sopra, proponiamo per il Cplex un primo metodo che indicheremo con *MA* (Meta-Algoritmo):

1. if $(dim \geqslant 1.5e06) \Rightarrow$ `Cplex_Barrier`
2. else if $(dim \geqslant 3e05) \Rightarrow$ `Cplex_Net`
3. else $\Rightarrow$ `Cplex_SimplessoDuale`

Tuttavia è naturale chiedersi se si possa fare di meglio tenendo conto anche di altri fattori. In particolare siamo interessati a migliorare la scelta nelle istanze dei gruppi 2 e 3, in quanto negli altri tre il risultato ottenuto è difficilmente migliorabile (la media dei tempi relativi degli algoritmi scelti è ~ 1); nella prossima sezione quindi considereremo solo questo sottoinsieme ridotto.

		Media				
	Tipologia	A	P	D	N	B
Gruppo 2	BDSA	1.69	3.03	1.59	1.58	1.25
	VBDSA	1.65	2.96	1.52	1.44	1.16
	VSA	1.58	3.56	1.41	1.75	1.68
Gruppo 3	BDSA	1.35	1.77	1.33	1.26	1.52
	VBDSA	1.34	1.88	1.31	1.28	1.62
	VSA	1.60	1.73	1.52	1.28	1.99

Tabella 4.8: Media dei tempi relativi del Cplex divisi per tipologia

4.3.1 Ulteriori divisioni

BDSA/VBDSA/VSA Per prima cosa proviamo ad analizzare i due gruppi divisi nelle tre categorie BDSA/VBDSA/VSA, ottenendo i risultati mostrati in tabella 4.8. Per quanto riguarda il Gruppo 3, il Net resta il migliore (con piccole variazioni) in tutti e tre i casi, mentre effettivamente nel Gruppo 2 possiamo osservare un dato potenzialmente interessante: la media del Barrier nelle istanze provenienti dal VSA è molto più alta delle altre due. Controllando i valori numerici, risulta che più di metà delle istanze su cui questo algoritmo è pessimo nel Gruppo 2 appartengono a questa categoria. Tuttavia, avendo solo due casi di questo tipo non possiamo affermare con sicurezza che il Barrier non funzioni bene su questa classe (o meglio, su questo Gruppo di questa classe). Per il resto non otteniamo altre informazioni rilevanti.

Invece di ordinare le istanze per dimensione, in quanto come già detto questa potrebbe essere una quantità non molto significativa, possiamo provare ad ordinare e dividere le istanze dei due gruppi interessati secondo altri valori.

Densità È noto come la densità di un problema di ottimizzazione possa influenzare l'andamento degli algoritmi. Ed abbiamo già accennato a come questa sia una quantità molto variabile, nel senso che istanze con dimensione simile possono avere densità diverse.

Tuttavia è anche vero che se la dimensione è grande difficilmente la densità sarà

		Media				
		A	P	D	N	B
Colonne	⩾4800	1.71	2.61	1.58	1.60	1.33
	<4800	1.27	1.93	1.25	1.18	1.64
Colonne/Righe	⩾ 25	1.79	2.06	1.67	1.68	1.34
	<25	1.26	2.48	1.23	1.17	1.58

Tabella 4.9: Media dei tempi relativi del Cplex divisi per numero di colonne e rapporto tra numero di colonne e righe

alta, e viceversa. Questo si traduce nel fatto che al livello di gruppi ampi come quelli da noi considerati, un ordinamento per densità cambierà l'ordine di molte istanze ma ne sposterà poche da un gruppo all'altro. Il risultato è che le medie sono simili alle precedenti, se non leggermente peggiori; passiamo quindi ad altro.

Righe Ricordando come colonne e righe giochino un ruolo diverso come fattori della dimensione di un'istanza, possiamo provare ad utilizzare il numero di righe o colonne; in aggiunta possiamo anche considerare il rapporto tra queste quantità (colonne/righe). La prima di queste tre divisioni non porta a nulla di utile, mentre le altre due danno i risultati illustrati nella tabella 4.9.

Rispetto ai risultati iniziali, siamo riusciti in entrambi i casi a selezionare un gruppo "inferiore" in cui il Net abbia prestazioni migliori (1.27 → 1.18/1.17). Per quanto riguarda il Barrier invece, le due medie nei gruppi "superiori" sono maggiori rispetto a quella che avevamo nel Gruppo 2 (1.29 → 1.33/1.34), tuttavia questo trade-off potrebbe anche essere conveniente. Considerando i valori numerici della divisione per colonne, si scopre che nel primo gruppo la percentuale di istanze su cui il Barrier è buono è aumentata di poco (49.6% → 51.3%), tuttavia la media è salita perché anche la percentuale di istanze su cui questo è pessimo è aumentata (11.4% → 16%).

Riassumendo, non siamo riusciti a trovare un modo ottimale per scegliere un algoritmo del Cplex in una certa fascia di istanze, non escludiamo tuttavia che si possa approfondire l'analisi e trovare un metodo migliore. L'unica variazione che possiamo proporre è la seguente:

1. if ($|colonne| \geqslant 4800$) $\Rightarrow$ `Cplex_Barrier`

2. else if ($|colonne| \geqslant 2000$) $\Rightarrow$ `Cplex_Net`

3. else $\Rightarrow$ `Cplex_Duale`

dove il punto 1 è giustificato dal fatto che tutte le istanze del Gruppo 1 hanno un numero di colonne maggiore di 4800, quindi questo può essere unito al nuovo Gruppo 2 (come nel caso MA). Volendo utilizzare lo stesso parametro anche nella scelta del punto 2, abbiamo sostituito ($dim \geqslant 3e5$) con ($|colonne| \geqslant 2000$). Possiamo indicare questo metodo con MAC (Meta-Algoritmo Colonne).

Con più casi appartenenti al VSA, si potrebbe approfondire l'applicabilità o meno del Barrier sulle istanze intermedie di questa classe e quindi adattare il meta-algoritmo di conseguenza.

4.4 Performance

Vediamo adesso come si comportano i due algoritmi proposti rispetto a quelli già presenti su di un sottoinsieme dei casi test. Nella tabella 4.10 sono riportati i tempi totali dei metodi del Cplex, in aggiunta ai tempi di MA e MAC. Per ogni riga, abbiamo evidenziato il tempo migliore.

Caso	A	P	D	N	B	MA	MAC
BDSA_2	1087.08	3029.30	1056.96	1031.86	469.04	**463.92**	464.02
BDSA_a	2.71	4.02	**2.48**	2.72	3.52	2.57	2.52
BDSA_b1	419.39	3673.03	421.85	416.40	396.20	393.18	**391.19**
BDSA_cc	77.98	335.90	71.80	**71.02**	91.04	81.91	90.07
VBDSA_a2	497.27	1205.67	453.38	449.43	163.53	**157.56**	159.57
VBDSA_c2	37.79	47.90	35.89	35.07	22.85	23.20	**21.36**
VBDSA_f	11.38	11.36	8.85	7.96	8.05	**7.22**	7.24
VSA_cc2	1135.94	1164.59	1086.71	1110.66	277.52	**273.22**	273.39

Tabella 4.10: Confronto tra i tempi totali dei vari algoritmi del Cplex su alcuni casi test

Questi risultati non dovrebbero stupire: se un problema presenta istanze molto grandi, il Barrier impiega molto meno tempo in totale in quanto è solitamente più efficiente su queste, ma a loro volta i meta-algoritmi tendono ad impiegare (poco) meno del Barrier in quanto passano ai più efficienti Duale/Net sulle istanze più piccole. Nei problemi di dimensioni minori come BDSA_a invece, può accadere che il Duale impieghi meno, a causa del fatto che MA/MAC usano il Barrier su qualche istanza di troppo.

4.5 Rapporto tra i metodi

Dopo aver analizzato separatamente i due solutori, è giunto il momento di confrontare i risultati tra di loro. Lo faremo in due modi: metteremo prima in relazione i tempi migliori di Clp e Cplex sulle varie istanze, per passare poi al rapporto tra il Duale del primo ed il Meta-Algoritmo del secondo.

Tempi migliori Questo primo confronto è di tipo più teorico: in un certo senso stiamo supponendo di poter scegliere, per ogni istanza, i due algoritmi che impiegano meno tempo.

Se ordiniamo le istanze a seconda del rapporto tra il tempo migliore del Clp e quello del Cplex, si osserva che queste si dispongono in modo simile alla prima divisione in gruppi effettuata; cerchiamo di capirne le ragioni. Nel Gruppo 1 si era visto come il Barrier del Cplex fosse praticamente sempre buono, tuttavia non ci eravamo chiesti di quanto fosse migliore rispetto agli altri. Se osserviamo le medie riportate in precedenza nella tabella 4.7, ci si accorge di come queste siano effettivamente alte, ad indicare che il Barrier sia in realtà *di gran lunga* il migliore su questa classe di problemi. I tempi mostrati nell'ultima riga delle tabelle 4.1, 4.2 ci danno un'idea di cosa può succedere: per entrambi i solver il Barrier è l'algoritmo migliore, *ma la differenza tra quello del Clp e quello del Cplex è enorme* ($8s$ contro $122s$). Quindi i valori più alti del rapporto tra il miglior tempo del Clp e quello del Cplex appartengono ai gruppi 1 e 2.

	BestClp/BestCplex	$\text{Clp}_D/\text{Cplex}_{MA}$
Gruppo 1	3.60	4.16
Gruppo 2	2.00	1.60
Gruppo 3	1.51	1.21
Gruppo 4	1.42	1.32
Gruppo 5	0.99	0.98

Tabella 4.11: Medie geometriche dei rapporti tra i solver

Nei gruppi intermedi questo rapporto scende; può anche capitare, soprattutto tra le istanze molto piccole, che scenda sotto l'1 ad indicare che il Clp impiega meno del Cplex; tuttavia stiamo parlando di valori di molto inferiori al secondo e che quindi non sono significativi ai fini del tempo totale.

Duale/MA In questo secondo confronto, le cose sono un po' diverse. Per capirne il motivo, dividiamo l'analisi nei vari gruppi.

- *Gruppo 1*: Rispetto al caso precedente, qui il rapporto tenderà ad essere *più alto*. Questo perché il numeratore è in media maggiore, a causa del fatto che non scegliamo mai il Barrier che è in alcuni casi migliore, mentre per il denominatore scegliamo praticamente sempre il valore minimo.

- *Gruppo 2-3*: Come abbiamo visto qui non siamo riusciti a separare bene gli algoritmi migliori del Cplex, mentre il tempo impiegato dal Duale del Clp è praticamente sempre il minore. Quindi il rapporto tra i tempi migliori sarà *più basso*.

- *Gruppo 4-5*: Qui siamo in grado di scegliere accuratamente gli algoritmi di entrambi i solver, quindi il rapporto tra i tempi sarà molto vicino a (probabilmente leggermente minore di) quello che avevamo.

Nella tabella 4.11 sono indicate le medie (geometriche) dei due rapporti nei vari gruppi. Come si può osservare, le due medie su di uno stesso gruppo si comportano come previsto.

Concludiamo questa sezione con la tabella 4.12 in cui mostriamo, sugli stessi casi della tabella 4.10, i totali dei tempi migliori del Clp e del Cplex, del Duale del primo e dell'MA del secondo. A causa del fatto che il Cplex è particolarmente adatto sulle istanze grandi, i suoi tempi totali sono generalmente molto più bassi; l'unico caso in cui il Clp impiega meno è anche il più piccolo, BDSA_a.

Caso	BestClp	BestCplex	Clp_D	Cplex_{MA}
BDSA_2	1831.69	424.12	1923.03	463.92
BDSA_a	2.22	2.33	2.26	2.57
BDSA_b1	947.41	327.63	967.51	393.18
BDSA_cc	94.17	56.52	94.17	81.91
VBDSA_a2	559.05	154.94	559.07	157.56
VBDSA_c2	46.41	20.52	56.77	23.20
VBDSA_f	10.99	6.94	11.02	7.22
VSA_cc2	922.92	248.81	1871.92	273.22

Tabella 4.12: Confronto tra i due solver

4.6 Barrier in parallelo

Chiudiamo questo capitolo con alcuni commenti riguardo al Cplex fatto girare in parallelo.

Come avevamo già accennato, se l'Automatic viene lanciato in parallelo utilizza contemporaneamente (modalità *concurrent*) Simplesso Primale, Duale e Barrier e si arresta quando uno degli algoritmi termina, ottenendo quindi tempistiche prossime a quelle di BestCplex. La nostra analisi è stata necessaria a causa della scelta di lavorare in modo sequenziale.

È comunque di un certo interesse fare una breve analisi dell'unico metodo parallelizzabile tra quelli che stiamo considerando: il Barrier. In particolare confrontiamo gli andamenti di tre "versioni"; a seconda del numero di thread che gli permettiamo di utilizzare, le indichiamo con *Barrier1* (B1), *Barrier4* (B4) o *Barrier8* (B8).

In teoria un maggior numero di thread a disposizione migliora le prestazioni dell'algoritmo, tuttavia dai dati in tabella 4.13 possiamo vedere che questo non sempre

Caso	Barrier1	Barrier4	Barrier8
BDSA_2	558.38	336.44	400.37
BDSA_a	4.17	4.87	4.97
BDSA_b1	430.06	250.23	274.02
VBDSA_a	170.69	136.03	175.72
VSA_cc2	304.53	251.84	341.98

Tabella 4.13: Tempi impiegati dai tre Barrier su alcuni casi

accade. O meglio, il B4 risulta spesso essere migliore del B1, ma il B8 al contrario è sempre peggiore di quest'ultimo e in alcuni casi è anche il peggiore in assoluto.

Definiamo ora l'*efficienza* di un metodo lanciato in parallelo come

$$\frac{t_{seq}}{t_{par} * n},$$

dove t_{seq} è il tempo impiegato dall'algoritmo sequenziale, t_{par} il tempo del parallelo e n il numero di thread. Quanto più questo valore è vicino a 1, più il metodo riesce a sfruttare il maggior numero di thread a disposizione. Dopo aver diviso le istanze di questi casi nei soliti cinque gruppi, calcoliamo la media geometrica delle efficienze e dei rapporti tra i diversi Barrier, ottenendo la tabella 4.14.

	Efficienza B4	Efficienza B8	B1/B4	B1/B8	B4/B8
Gruppo 1	0.35	0.14	1.42	1.10	0.78
Gruppo 2	0.28	0.10	1.12	0.80	0.72
Gruppo 3	0.22	0.10	0.90	0.76	0.85
Gruppo 4	0.21	0.09	0.83	0.76	0.91
Gruppo 5	0.19	0.09	0.76	0.74	0.97

Tabella 4.14: Efficienza dei barrier nei vari gruppi

Da questa vediamo che non solo l'efficienza è bassa, ma B4 è migliore del B1 solo nei primi due gruppi, mentre B8 addirittura solo nel Gruppo 1. Qui stiamo parlando di medie, ma se pure è vero che su alcune istanze abbiamo l'ordinamento che ci aspetteremmo (B8<B4<B1), queste sono le più grandi in assoluto (con dimensione>

$1.4e7$) di cui ce ne sono solo una decina (e poco più di 20 nella totalità delle istanze considerate in precedenza).

Possiamo quindi affermare che lanciare il Barrier con 4 thread sia una scelta preferibile sia al sequenziale che a 8 thread, ma che comunque la parallelizzazione di questo algoritmo sui casi da noi presi in considerazione non sia molto efficiente.

Capitolo 5

Bundle

L'analisi del Bundle seguirà un percorso diverso rispetto a quella del capitolo precedente. Partiamo infatti da una considerazione molto importante: il Bundle *non* è un buon metodo per ottenere soluzioni precise nei casi da noi presi in esame. Questo viene comunque utilizzato dalla MAIOR in quanto per i loro scopi non è necessario un elevato grado di accuratezza, ed il Bundle può essere arrestato per ottenere una soluzione relativamente buona in tempi ragionevoli.

Iniziamo quindi presentando un confronto tra la versione (inesatta) attualmente utilizzata e Clp/Cplex, dopodiché verrà imposta una certa qualità della soluzione e vedremo quanto questo influenzi il tempo di calcolo. Cercheremo comunque di ridurre il più possibile le tempistiche agendo sui parametri che regolano l'algoritmo.

L'implementazione che (noi e la MAIOR) stiamo utilizzando è stata scritta scritta dal professor Antonio Frangioni, del gruppo di Ricerca Operativa del Dipartimento di Informatica dell'Università di Pisa.

Caso	Clp_D	Cplex_{MA}	Bundle
BDSA_1	2007.60	335.00	942.5
BDSA_2	1923.03	463.92	2253.40
BDSA_a	2.26	2.57	15.25
BDSA_b1	967.51	393.18	619.71
BDSA_cc	94.17	81.91	560.71
VBDSA_a2	559.07	157.56	345.06
VBDSA_c2	56.77	23.20	124.88
VBDSA_f	11.02	7.22	36.28
VSA_cc2	1871.92	273.22	1629.46

Tabella 5.1: Primo confronto del Bundle con Clp/Cplex su alcuni casi test

5.1 Tempistiche

Quando si guardano i risultati di questa sezione, bisogna sempre tenere in mente che le soluzioni prodotte sono *diverse*, nel senso che il poter dare soluzioni imprecise avvantaggia notevolmente il Bundle in termini di tempo; il punto è che se ottenessimo risultati paragonabili, un algoritmo esatto sarebbe probabilmente una scelta migliore.

Nella tabella 5.1 sono riportati i totali sui casi test visti anche nella 4.12, con l'aggiunta di BDSA_1. Da questa possiamo notare che:

- Sui casi piccoli (BDSA_a, VBDSA_c2, VBDSA_f) il Bundle impiega più tempo sia del Cplex che del Clp
- Sui casi medi e grandi si colloca a volte tra il Clp ed il Cplex (in varia misura), mentre altre volte è peggiore di entrambi (BDSA_2, BDSA_cc)

Ci aspettiamo quindi che il Bundle non sia molto efficiente sulle istanze di piccole dimensioni, mentre possa esserlo in qualche misura su quelle grandi. Per capire cosa accade in questo secondo caso, guardiamo alcune istanze tra le più grandi di BDSA_1 e BDSA_2 (tabella 5.2).

Come possiamo vedere, questo metodo può andare sia molto bene, anche meglio del Cplex, che molto male, impiegando tempi circa 10 volte più lunghi. A seconda di quanto spesso questa seconda eventualità si presenta, possiamo avere casi come

Caso	Clp_D	Cplex_{MA}	Bundle
BDSA_1	4.607	1.446	0.663
	6.829	2.018	0.848
	8.927	2.058	0.987
BDSA_2	9.889	3.474	1.463
	11.715	4.828	1.201
	20.240	4.825	41.428

Tabella 5.2: Tempi su alcune istanze di grande dimensione

BDSA_b1 in cui il tempo totale del Bundle è di poco superiore al Cplex, oppure come BDSA_2 in cui questo è peggiore del Clp.

Allo stesso modo riportiamo in tabella 5.3 i tempi su alcune istanze di piccola dimensione di BDSA_a e VBDSA_f.

Caso	Clp_D	Cplex_{MA}	Bundle
BDSA_a	0.035	0.035	0.268
	0.041	0.058	0.254
	0.028	0.022	0.222
VBDSA_f	0.066	0.050	0.203
	0.053	0.033	0.161
	0.535	0.250	4.091

Tabella 5.3: Tempi su alcune istanze di piccola dimensione

Qui riscontriamo di nuovo gli occasionali picchi, uniti al fatto che su questa categoria di istanze il Bundle è evidentemente peggiore; da ciò discendono tempi totali molto più alti relativamente a quelli di Clp/Cplex.

D'altra parte c'è da considerare che sono state utilizzate le versioni più recenti dei due software, mentre il codice del Bundle è vecchio di 10 anni. Procediamo allora come prima cosa alla sostituzione con la sua ultima versione.

Caso	Clp_D	Cplex_{MA}	Bundle_{05}	Bundle_{14}
BDSA_1	2007.6	334.3	942.5	491.4
BDSA_b1	967.5	393.2	619.7	312.1
BDSA_b2	28.6	20.7	61.0	29.5
BDSA_c2	93.9	51.2	252.6	122.7
VBDSA_3	245.0	57.3	176.2	85.8
VBDSA_c2	56.8	23.2	124.9	64.2
VBDSA_f	11.0	7.2	36.3	17.5
VSA_cc2	1871.9	273.2	1629.5	705.6

Tabella 5.4: Confronto tra Bundle_{05} e Bundle_{14} su alcuni casi test

5.2 Bundle_{14} e Bundle_{15}

Dovendo distinguere tra diverse versioni (successive) dello stesso algoritmo, includeremo nel nome l'anno in cui sono state scritte. Quindi ci riferiremo al Bundle utilizzato finora come $Bundle_{05}$ ed all'ultima versione disponibile come $Bundle_{14}$.

Una volta effettuata la sostituzione dell'algoritmo, questo è stato fatto girare con gli stessi parametri del precedente. Si ottiene che per calcolare la stessa soluzione è necessario un tempo molto minore, pari a circa la metà.

Tuttavia per analizzare il vero comportamento dell'algoritmo è necessario rimuovere il limite sul numero di iterazioni. Quando questo vincolo viene rimosso, stranamente la situazione si inverte: il Bundle_{14} impiega tempi molto maggiori rispetto al Bundle_{05}.

Questo ha portato alla necessità di sviluppare una nuova versione, il $Bundle_{15}$. In realtà le differenze tra questa e la precedente sono estremamente ridotte al livello di codice (meno di una riga), tuttavia i risultati prodotti sono distinti e più che soddisfacenti: a parità di soluzione prodotta, è paragonabile al Bundle_{14} in presenza del limite, ma in sua assenza c'è un significativo miglioramento (piuttosto che peggioramento) rispetto al Bundle_{05}.

Nella tabella 5.5 sono riportati alcuni tempi delle varie versioni; l'asterisco indica che il calcolo è stato interrotto prematuramente. Questo è stato fatto solo nel caso del Bundle_{14} su problemi molto grandi, una volta che fosse chiara la sua impraticabilità.

	Con Limite			Senza Limite		
Caso	Bundle_{05}	Bundle_{14}	Bundle_{15}	Bundle_{05}	Bundle_{14}	Bundle_{15}
BDSA_1	942.5	491.4	517.7	56640.7	67806.3*	33832.5
BDSA_b1	619.7	312.1	265.0	74162.1	96503.7*	46258.9
BDSA_b2	61.0	29.5	27.9	1659.4	8749.6	906.2
BDSA_c2	252.6	122.7	110.9	1677.3	2059.6	982.1
BDSA_cc	560.7	237.7	207.7	35778.4	91124.9*	18135.7
VBDSA_3	176.2	85.8	90.7	9626.4	12111.4*	6026.1
VBDSA_c2	124.9	64.2	75.3	1716.8	3315.0	989.4
VBDSA_f	36.3	17.5	16.0	550.1	324.3	281.5
VSA_cc2	1629.5	705.6	840.2	225803.9	153003.41*	107044.2

Tabella 5.5: Confronto tra i vari Bundle con e senza limite sul numero di iterazioni

Cercheremo adesso di ridurre questi tempi per quanto possibile, ma è evidente che stiamo lavorando con un'ordine di grandezza completamente diverso rispetto a prima.

Torniamo ad utilizzare semplicemente il termine Bundle; tuttavia con esso stiamo intendendo una versione diversa dal precedente Bundle_{05}, ossia il Bundle_{15}. Inoltre d'ora in avanti lavoreremo *senza il limite sul numero d'iterazioni*.

5.3 Tuning dei parametri

È stato più volte sottolineato come un elemento di difficoltà dell'utilizzare il Bundle sia il gran numero di parametri che ne regolano il comportamento; nel fare il tuning prenderemo in considerazione alcuni dei principali.

Innanzitutto ci concentreremo sulla precisione delle soluzioni prodotte; una volta che ci saremo ritenuti soddisfatti passeremo al tuning dei restanti parametri al fine di diminuire i tempi totali.

Questa fase è semplificata dal non dover partire da zero: l'attuale setting MAIOR è stato motivato da un'analisi analoga avvenuta in passato. Tuttavia il nostro fine è diverso, in quanto vogliamo sia prodotta una soluzione con un certo grado di precisione, quindi dovremo in qualche misura discostarci dai valori preimpostati.

Dato che il chiedere una una certa precisione potrebbe aumentare ulteriormente i tempi, in modo analogo a quanto fatto in precedenza (sezione 4.2.1) ci restringiamo ad un sottoinsieme delle istanze. Tuttavia questa volta la restrizione non sarà limitata ai casi critici, ma si estenderà (in misura variabile) a tutti. D'ora in avanti ci riferiremo alle istanze del sottoinsieme come istanze dei casi *Test(Selezione)*. Finito il tuning, confronteremo i tempi del Bundle con quelli di Clp e Cplex sul totale delle istanze.

5.3.1 Precisione

Iniziamo dalla categoria che più ci interessa: i parametri che regolano la precisione della soluzione. Questa è dettata essenzialmente dai valori assegnati ad *EpsLin* e *tStar*, che rappresentano rispettivamente *la precisione relativa* richiesta ed *una stima del passo massimo* che può essere effettuato.

Per precisione relativa intendiamo il gap che si vuole a terminazione tra la soluzione calcolata e la vera. Ovvero, indicando questi due valori con C e V rispettivamente, stiamo parlando della seguente quantità:

$$\frac{V - C}{V}$$

Nel nostro caso non accadrà mai che V sia pari a 0, quindi non abbiamo bisogno di adattare la definizione per contemplare questo caso.

Dimostrare che un qualche punto λ sia ottimo per un problema di ottimizzazione non lineare richiede che venga trovato un subgradiente nullo della funzione nel punto λ. In realtà si pongono condizioni più deboli, in particolare nel caso del Bundle ci basta trovare un σ-subgradiente g per cui valga

$$tStar * ||g|| + \sigma \leqslant EpsLin * |Maxf|, \tag{5.1}$$

dove g è un'opportuna combinazione lineare dei subgradienti calcolati in precedenza, *Maxf* indica una stima dell'ottimo del problema (ottenuta tramite il duale) e $||\cdot||$ è una qualche norma (di solito $||\cdot||_2$). *tStar* può quindi essere visto come una

stima della diminuzione effettiva che può essere ottenuta muovendosi di un passo unitario nella direzione di un subgradiente.

Il valore di *EpsLin* va deciso a priori, a seconda della precisione che si vuole raggiungere. Il valore impostato dalla MAIOR è pari a 1.e-5, che in realtà è già abbastanza basso e non richiede modifiche. Tuttavia anche una volta rimosso il numero di iterazioni si potrebbe non raggiungere questo valore se a *tStar* non è stato assegnato un valore idoneo.

In particolare cerchiamo un valore che sia lo stretto necessario per avere una precisione relativa pari a *EpsLin*. Dato che conosciamo le soluzioni esatte delle istanze, possiamo procedere nel seguente modo: fissiamo un primo valore di *tStar*, ad esempio l'attuale valore 1, calcoliamo le soluzioni del Bundle e da queste i gap rispetto alle soluzioni vere.

-Se i vari gap sono già minori di *EpsLin*, procediamo in modo *discendente* diminuendo tStar di un fattore 10 finché i gap diventano maggiori della soglia, dopodiché fissiamo il penultimo valore. Questo appunto perché vogliamo che *tStar* sia il più piccolo possibile.

-Altrimenti, procediamo in modo *ascendente* aumentando tStar di un fattore 10. In modo analogo, procediamo finché i gap diventano minori della soglia, e fissiamo l'ultimo valore considerato.

Alcune note:

- È difficile dire a priori quale sia un buon valore di partenza. Nel nostro caso abbiamo utilizzato il valore preimpostato per avere un'idea di cosa accadesse una volta rimosso il limite, tuttavia come vedremo abbiamo dovuto aumentare questa quantità per avere delle soluzioni soddisfacenti

- In realtà richiedere che tutti i gap siano minori di *EpsLin* è una condizione troppo forte, è meglio utilizzare qualcosa di meno restrittivo: ad esempio che e che i valori siano minori di $n*$*EpsLin* per un qualche n un numero consistente di volte (90%, 95% del totale)

- Aumentare *tStar* può *diminuire* i tempi. Questo perché se il valore non è accuratamente impostato, può accadere che su alcune istanze il solver sia già

	tStar				
Caso	1	10	100	1000	10000
BDSA_1	3949.9	5075.2	6613.1	6018.8	3738.0
BDSA_b1	32231.2	21582.5	24120.1	20204.5	30276.4
BDSA_b2	226.3	221.3	187.4	179.3	248.4
BDSA_c2	252.9	285.2	385.3	497.4	563.0
VBDSA_3	639.5	902.4	731.9	827.4	855.9
VBDSA_c2	935.6	1122.5	1114.1	1224.6	1151.0
VBDSA_f	30.5	25.1	23.7	26.6	30.4
VSA_cc2	6128.1	5708.6	5016.3	6315.1	7714.3

Tabella 5.6: Variare dei tempi al variare di *tStar*

arrivato ad una soluzione con precisione richiesta ma non se ne accorga e faccia quindi ulteriori iterazioni non necessarie.

Ponendo $n = 5$ nella precedente condizione otteniamo che la precisione relativa è minore della soglia nella seguente percentuale di istanze rispetto al totale:

$tStar = 1$	$tStar = 10$	$tStar = 100$	$tStar = 1000$
76.43%	85.54%	92.08%	94.44%

Se assumere $tStar = 1000$ oppure provare ad arrivare a percentuali maggiori dipende dal prezzo che stiamo pagando in termini di tempo. Nella tabella 5.6 riportiamo alcuni tempi al variare di *tStar*; la situazione è abbastanza varia: per alcuni problemi $tStar = 1000$ oltre ad aumentare la precisione diminuisce i tempi (primo tra tutti BDSA_b1), mentre per altri (BDSA_1) è $tStar = 10000$ a sortire questo effetto.

Comunque in generale il primo di questi valori è migliore; assumiamo quindi $tStar = 1000$, per il quale quasi il 95% delle istanze ha una precisione relativa inferiore a $5e-5$.

5.3.2 Altri parametri

Diamo innanzitutto una descrizione sommaria dei vari parametri che andremo a modificare, divisi negli ambiti di appartenenza.

$\mathcal{B}$-Strategy: Con questo termine intendiamo la strategia che viene seguita per quanto riguarda la gestione del bundle $\mathcal{B} = \{(x_i, f(x_i), z_i) : x_i \in \bar{X}\}$. Andiamo a modificare solo due parametri che si occupano della dimensione del bundle:

- *BPar1*: Si occupa di rimuovere oggetti dal bundle. Tramite il duale del Master Problem è facile capire se un oggetto è inutile, ovvero se la soluzione ottima sarebbe la stessa anche in seguito alla sua rimozione; se un oggetto lo è stato per gli ultimi *BPar1* passi, viene eliminato. Chiaramente se questo parametro è troppo basso si potrebbero perdere informazioni importanti, d'altro canto un bundle piccolo comporta Master Problem meno costosi.

- *BPar2*: Regola la dimensione massima del bundle. In modo analogo a *BPar1* non vogliamo sia troppo piccolo per non perdere informazioni, ma se quest'ultimo è stato scelto bene *BPar2* può essere tenuto alto mentre la $\mathcal{B}$-strategy si occupa di mantenere il numero effettivo di elementi basso. Quindi da una parte un valore basso può influenzare negativamente la convergenza dell'algoritmo, ma d'altro canto un valore inutilmente alto potrebbe far allocare un gran quantitativo di memoria all'MPSolver senza alcuna necessità.

t-Strategy: Con questo termine intendiamo la strategia che viene seguita per quanto riguarda la modifica del parametro di prossimità t. Come accennato nella sezione 1.5 infatti, è meglio avere una regola che permetta di cambiare il suo valore passo per passo. I due parametri che ci interessano sono:

- *tInit*: Il valore iniziale di t. È noto come l'impostanzione di questo parametro possa essere critica, per fortuna però è anche facile: di solito c'è un ordine di grandezza giusto per t, che può essere determinato tramite euristiche effettuate anche partendo da un valore "errato". Inoltre è legato a *tStar*, nel senso che questo è spesso uno o due ordini di grandezza più grande del valore iniziale di t.

- *tSPar2*: Il Bundle permettere di scegliere tra diverse t-strategy (utilizzando il parametro *tSPar1*), e in alcune di queste la condizione sull'aumento/ diminuzione di t è regolata da *tSPar2*.

Step Conditions: Quando viene trovata una nuova direzione di discesa d^* rispetto al punto corrente $\bar{x}$, è necessario avere una condizione per decidere se il nuovo punto $\bar{x} + d^*$ migliori abbastanza la soluzione da effettuare un Serious Step (ovvero considerarlo come il nuovo punto corrente). Di questo si occupa

- *m1*: Regola la condizione di Serious Step (SS). Questo viene effettuato se

$$f(\bar{x} + d^*) - f(\bar{x}) \geqslant |m1| * \Delta v,$$

  nella quale

  - $\mathrm{m1} > 0 \Rightarrow \Delta v = f_\beta(\bar{x} + d^*) - f(\bar{x})$
  - $\mathrm{m1} < 0 \Rightarrow \Delta v = f_\beta(\bar{x} + d^*) - f(\bar{x}) - D_t(d^*)$

  Dato che il primo dei due Δv è sempre maggiore od uguale dell'altro, la seconda condizione è più debole e potrebbe portare ad un maggior numero di SS, e quindi ad una convergenza più rapida. Il valore $m1 = 0$, ovvero effettua un SS ad ogni miglioramento della funzione obiettivo, può essere utilizzata (almeno in teoria) solo su alcune classi di funzioni e con determinate ipotesi sul Master Problem.

- *m3*: Determina quando un nuovo subgradiente cambia sufficientemente il Master Problem, nel senso che siamo abbastanza sicuri un Null Step ci porti ad una nuova direzione diversa e migliore dell'attuale (che non è stata sufficientemente buona); più questo valore è piccolo, più NS si fanno prima di provare a diminuire t. Questo parametro è critico, e dipende dalla scelta della t-strategy da seguire.

Nella tabella 5.7 sono mostrati i valori attualmente in uso ed i nuovi valori che abbiamo provato. Questi sono motivati da test effettuati in passato con questa

Parametro	MAIOR	Nuovo Valore
BPar1	10	20(50)
tInit	0.001	1
tSPar2	0.1	0.001
m1	0.1	-0.1
m3	0.9	3

Tabella 5.7: Valori dei parametri del Bundle

Caso	Base	*BPar1*	*tInit*	*tSPar2*	*m1*	*m3*
BDSA_1	6018.8	6208.0	5552.8	19129.1	5174.79	6602.75
BDSA_b1	20204.5	21253.6	19144.2	34037.2	18690.6	23357.4
BDSA_b2	179.3	164.5	168.2	300.52	161.4	371.3
BDSA_c2	531.5	382.5	877.7	421.2	450.8	410.0
VBDSA_3	827.4	878.6	822.5	3173.5	816.78	1137.7
VBDSA_c2	1224.6	1372.4	1143.6	2228.5	1164.9	1727.77
VBDSA_f	26.6	27.0	38.6	35.8	25.5	34.6
VSA_cc2	6315.1	6500.3	6918.0	12398.2	6153.3	6020.3

Tabella 5.8: Effetto della modifica dei parametri su alcuni casi

implementazione del Bundle, tuttavia non è detto che applicati ai casi da noi in esame apportino dei miglioramenti.

Nel caso di *BPar1* sono stati indicati due valori in quanto se il passaggio da 10 a 20 dovesse essere positivo, si potrebbe provare ad aumentare ulteriormente questo valore ponendolo pari a 50. Come si può vedere manca *BPar2*; in questa sede infatti non siamo interessati a modificarne valore, è stato introdotto in quanto sarà rilevante successivamente nel Bundle Disaggregato.

Se dovessimo provare tutte le combinazioni di valori sarebbe un problema, in quanto ne avremmo ben 32. Tuttavia questi parametri sono in un certo senso indipendenti l'uno dall'altro: possiamo quindi valutare i cambiamenti che si ottengono modificando un parametro alla volta e poi provare le combinazioni tra quelli la cui modifica apporta miglioramenti.

Dai test risulta che i due parametri più interessanti sono *m1* e *tInit* (tabella 5.8). Il primo diminuisce sempre, anche se di poco, i tempi totali; mentre il secondo può

Caso	Clp_D	Cplex_{MA}	Bundle_{05}	Bundle_{15}	Bundle
BDSA_1	2007.6	335.0	56640.6	33832.4	33554.4
BDSA_b1	967.5	393.1	74162.1	46258.9	33002.5
BDSA_b2	28.5	20.7	1659.4	906.2	670.8
BDSA_c2	94.1	81.9	1677.3	982.0	2441.3
VBDSA_3	559.0	157.5	9626.4	6026.0	5803.9
VBDSA_c2	56.7	23.2	1716.8	989.3	1088.6
VBDSA_f	11.0	7.2	550.0	281.5	203.9
VSA_cc2	1871.9	273.2	225803.9	107044.1	82801.9

Tabella 5.9: Evoluzione del Bundle e confronto con Clp/Cplex su alcuni casi test

avere un impatto maggiore anche se non sempre questo è positivo.

Dopo aver verificato che la combinazione di questi due valori fosse effettivamente il meglio che potessimo ottenere concludiamo l'analisi del Bundle confrontandolo con i metodi visti in precedenza.

5.4 Confronto

Contrariamente ai primi risultati riportati all'inizio di questo capitolo (tabella 5.1), adesso il Bundle è "allo stesso livello" di Clp e Cplex, nel senso che le soluzioni prodotte sono (relativamente) esatte. Rimuoviamo quindi la condizione di restringerci alle istanze di Test(Selezione) e consideriamo i vari problemi nella loro interezza.

Oltre al confronto con i solutori esatti, abbiamo anche riportato l'evoluzione del Bundle; in generale (ma non sempre) siamo riusciti sia a diminuire i tempi che ad aumentare la precisione.

Tuttavia com'è evidente il Bundle non regge assolutamente il confronto non solo con il Cplex, ma neanche con il Clp. Confermiamo quindi quanto detto all'inizio del capitolo, ovvero che questo *non* è un buon metodo per calcolare soluzioni "esatte" sui problemi affrontati dalla MAIOR. Vediamo adesso cosa succede nel passare alla versione disaggregata.

Capitolo 6

Bundle Disaggregato

Come abbiamo già accennato, con Bundle Disaggregato intendiamo una versione in cui le colonne attive sono divise in varie componenti invece di essere considerate tutte insieme. Per esplorare questo concetto, riprendiamo la descrizione del problema (SP) data nel capitolo 2. Considerando in aggiunta la modellizzazione lineare dei vincoli globali, otteniamo

$$\begin{aligned} &\min \sum_{i=1}^{n} c_i x_i \\ &Ax = b \\ &l \leqslant Bx \leqslant u \\ &x_i \in \{0,1\} \quad \forall i \in \{1,\ldots,n\} \end{aligned} \tag{SP'}$$

dove $b_i = 1 \quad \forall i \in \{1,\ldots,m\}$, $A \in \mathbb{R}^{m\times n}$ e $B \in \mathbb{R}^{k\times n}$. Per avere tutti vincoli di uguaglianza vengono introdotte k variabili di slack y_j con costo nullo; in questo modo abbiamo

$$\begin{aligned} &\min \sum_{i=1}^{n} c_i x_i \\ &Ax = b \\ &Bx - I_k y = d \qquad \text{(SP")} \\ &x_i \in \{0,1\} \quad \forall i \in \{1,\dots,n\} \\ &y_i \in \{l_i, u_i\} \quad \forall i \in \{1,\dots,k\} \end{aligned}$$

dove $d_i = 0 \quad \forall i \in \{1,\dots,k\}$ e I_k è la matrice identità $k \times k$. Tenendo conto di questo fattore in più, la funzione 2.3 (il rilasciamento lagrangiano) può essere riscritta come

$$\phi(\lambda) = \underbrace{\lambda[b,d]^T}_{(0)} + \underbrace{\sum_{i=1}^{n}(c_i - \lambda A^i)x_i}_{(1),(2),(3)} + \underbrace{\sum_{i=1}^{k}\lambda I_k^i y_i}_{(4)}, \tag{6.1}$$

dalla quale per calcolare il differenziale in un punto utilizziamo

$$g(\lambda) = \underbrace{[b,d]^T}_{(0)} - \underbrace{\sum_{i=1}^{n} A_i x_i}_{(1),(2),(3)} + \underbrace{\sum_{i=1}^{k} I_k^i y_i}_{(4)}. \tag{6.2}$$

Ricordiamo che, per definizione di duale e costo ridotto, al momento della valutazione di queste due funzioni in un λ fissato si ha

- $c_i - \lambda A_i < 0 \Rightarrow x_i = 1, \quad c_i - \lambda A_i > 0 \Rightarrow x_i = 0$
- $\lambda_{m+i} < 0 \Rightarrow y_i = u_i, \quad \lambda_{m+i} > 0 \Rightarrow y_i = l_i.$

La seconda componente è stata divisa in tre in quanto nella sommatoria oltre alle colonne attive vere e proprie (1) sono considerati anche i tripper dei task (2) e dei vincoli globali (3). Quindi al momento della disaggregazione valutiamo le varie componenti separatamente nel seguente modo:

1. Componente (0): il vettore $[b, d]$ non deve essere calcolato, e vista la sua struttura il prodotto $\lambda[b, d]^T$ è semplicemente la somma delle prime m componenti di λ. Inoltre è qui che consideriamo il contributo delle colonne fissate, che si limitano ad incrementare il valore della funzione.

2. Componente (1): è qui che va deciso il livello di disaggregazione. Fissato il numero di colonne per componente γ, dividiamo il contributo delle colonne attive nelle varie componenti

$$\phi_i(\lambda) = \sum_{j=(i-1)\gamma}^{i\gamma} (c_i - \lambda A^i)x_i, \quad g_i(\lambda) = -\sum_{j=(i-1)\gamma}^{i\gamma} A^i x_i$$

3. Componenti (2) e (3): i tripper dei task e dei vincoli globali sono gestiti in modo diverso tra di loro e rispetto al resto delle colonne. Li dividiamo quindi in due componenti distinte.

4. Componente (4): in realtà è improprio dire che $y_i \in \{l_i, u_i\}$, nel senso che nel caso in cui siano presenti colonne fissate vanno piuttosto considerati i valori aggiornati di lower ed upper bound.

6.1 OsiMPSolver

Al livello di codice alcune componenti del Bundle prevedono già la disaggregazione, in quanto la versione completamente disaggregata non è una novità. Per far funzionare la nostra versione intermedia, è necessario intervenire principalmente sull'oracolo.

Purtroppo quello utilizzato dalla MAIOR lavora soltanto con il caso aggregato; è stato quindi necessario adattarlo per considerare più di una componente per volta, e quindi far sì che le componenti non fossero colonne singole ma gruppi di colonne.

Anche dopo aver esteso le funzionalità dell'oracolo, c'è un ulteriore fattore da considerare: è necessario cambiare l'MPSolver (ovvero il solutore per risolvere 1.1) impiegato dal Bundle in quanto quello utilizzato finora (il QPPenalty) lavora solo con la versione aggregata. L'alternativa a disposizione è l'OsiMPSolver, il quale invece di

utilizzare un solver specializzato (come il precedente) permette di scegliere tra Clp e Cplex, i quali a loro volta mettono a disposizione gli stessi algoritmi del capitolo precedente; perciò in modo analogo andranno sviluppati due meta algoritmi.

Quindi alla scelta del grado di disaggregazione più opportuno, si aggiunge la difficoltà di dover considerare (almeno nel caso del Cplex) ben quattro algoritmi diversi. Inoltre per ciascuno di questi, almeno nel caso del Cplex, è necessario indicare il tipo di stabilizzazione da utilizzare: lineare o quadratica.

Questo comporta che, in un certo senso, per quest'ultimo caso vanno considerati un totale di *otto* algoritmi distinti, ciascuno dei quali potrebbe richiedere parametri diversi ed una differente divisione in colonne per funzionare al meglio.

Le scelte da fare sono quindi le seguenti:

- Stabilizzatore. Il Cplex mette a disposizione il Boxstep (lineare) ed il Quadratic (quadratico), mentre il Clp solo il primo
- Algoritmo da utilizzare. Come nel capitolo precedente, abbiamo Simplesso Primale, Duale, Net e Barrier per il Cplex, e gli stessi meno il Net per il Clp
- Tuning dei parametri. Per ciascun algoritmo va ripetuta un'analisi analoga a quella effettuata per il Bundle Aggregato
- Disaggregazione. Quante colonne mettere in ciascun sottoinsieme è un parametro che probabilmente varierà da algoritmo ad algoritmo e sarà in qualche misura dipendente dalla dimensione del problema

Iniziamo dando una prima valutazione dello scarto nel passaggio dal QPPenalty all'OsiMPSolver; essendo il primo un solutore specializzato ci aspettiamo che i tempi aumentino. Dato che il Cplex è quasi sicuramente migliore del Clp, in una fase iniziale considereremo solo questo programma. A seconda dei risultati che otterremo, decideremo quanta importanza dare al Clp. Di nuovo, nella maggior parte dei casi test utilizzeremo un numero di istanze ridotte (Test(Selezione), come nel caso del capitolo precedente) fino al momento del confronto finale. Procederemo in questo modo anche nel prossimo capitolo per analizzare il Subgradiente.

		Bundle$_X$DQ	
	QPPenalty	$\gamma = \infty$	$\gamma = 1$
BDSA_b2	10.1	1314.2	56.8
	32.9	3816.1	986.1
	0.9	20.0	1.6
VBDSA_c2	5.7	239.5	3264.0
	52.5	6499.6	363.1
	24.8	4832.9	1832.7

Tabella 6.1: Confronto tra il QPPenalty e l'OsiMPSolver su alcune istanze

Per distinguere i due casi useremo i nomi $Bundle_X$ e $Bundle_C$, seguiti a loro volta da una lettera per indicare l'algoritmo (P, D, N, B) e da una per lo stabilizzatore (se presente, B per Boxstep e Q per Quadratic). Infine, γ indicherà il numero di colonne per componente. La versione aggregata verrà indicata con $\gamma = \infty$.

Per dare un'idea della differenza in termini di tempo, abbiamo riportato i risultati su alcune istanze di dimensioni medie dei problemi BDSA_b2 e VBDSA_c2 (6.1). È evidente come i tempi aumentino in modo drastico: in alcuni casi passiamo da $\sim 30s$ a $\sim 4000s$. D'altra parte il caso totalmente disaggregato, che pure ci aspettavamo non fosse praticabile, è spesso molto migliore del caso aggregato; consideriamo comunque che su istanze qualitativamente diverse questo comportamento potrebbe invertirsi. Tenendo presente il punto di partenza alto dell'OsiMPSolver, la speranza è che la disaggregazione parziale sia così vantaggiosa da riuscire a portare i tempi a valori inferiori al QPPenalty.

Per sviluppare i due meta-algoritmi procediamo come segue:

1. Inizializzazione: cerchiamo una prima divisione in colonne rozza che diminuisca i tempi totali e renda possibile un'analisi preliminare dei metodi ed impostiamo dei parametri che vadano bene in generale

2. Disaggregazione: per ogni algoritmo, cerchiamo il modo migliore di dividere in colonne

3. Tuning: per ciascuno dei vari metodi, facciamo un'analisi più approfondita per differenziare (se necessario) l'impostazione dei parametri

		Bundle$_X$DQ			
	QPPenalty	$\gamma = \infty$	$\gamma = 1$	$\gamma = 50$	$\gamma = 100$
BDSA_b2	10.1	1314.2	56.8	126.0	200.9
	32.9	3816.1	986.1	445.2	473.0
	0.9	20.0	1.6	1.6	1.6
VBDSA_c2	5.7	239.5	3264.0	22.7	35.5
	52.5	6499.6	363.1	22.0	25.0
	24.8	4832.9	1832.7	103.3	46.6

Tabella 6.2: Primi valori di divisione

4. Conclusione: confrontiamo i tempi per ottenere il meta-algoritmo disaggregato

6.2 Cplex

Nell'utilizzare i vari algoritmi del Cplex, c'è da considerare che cambiare stabilizzatore influenza pesantemente l'andamento dei metodi. Per questo teniamo separati i risultati del Boxstep da quelli del Quadratic, percorrendo le varie fasi due volte. Una volta ottimizzati, li confronteremo per avere il meta-algoritmo.

6.2.1 Inizializzazione

Come abbiamo visto con il cambio di MPSolver le tempistiche aumentano di molto, quindi è importante trovare una prima divisione che renda più maneggevole l'analisi. A priori non è chiaro quale possa essere un buon valore di γ, considerando che non siamo nemmeno certi la disaggregazione possa diminuire i tempi. Provando diversi valori, risulta che in effetti è questo il caso e i tempi migliori si ottengono con quantità inferiori a 100 (6.2).

Di questi casi, solo in uno i tempi restituiti dalla disaggregazione sono effettivamente migliori rispetto al Bundle standard (seconda istanza del VBDSA_c2), comunque in generale sono diminuiti di molto.

Per quanto riguarda i parametri, per adesso manteniamo quelli del Bundle; l'unica eccezione è BPar2 in quanto, regolando la dimensione del bundle, è ragionevole

		Bundle$_X$DQ			
	QPPenalty	$\gamma = \infty$	$\gamma = 1$	$\gamma = 50$	$\gamma = 100$
BDSA_b2	10.1	9791.3	456.9	19.8	37.0
	32.9	11748.5	1183.4	33.2	37.2
	0.9	64.4	3.56	1.2	2.1
VBDSA_c2	5.7	4831.6	6892.3	95.9	136.0
	52.5	7340.2	322.1	9.8	22.3
	24.8	5181.0	237.1	6.5	6.0

Tabella 6.3: Primi valori del Barrier

renderlo dipendente dal numero di componenti. Dopo una breve analisi è stato fissato il valore di 500*γ.

Prima di passare all'analisi dei vari algoritmi, abbiamo verificato che $\gamma = 50$ fosse un valore di riferimento anche per gli altri metodi; continuiamo ad utilizzare il Quadratic come stabilizzatore. A titolo di esempio, ci limitiamo a riportare un confronto analogo al precedente utilizzando però il Barrier.

Fissato questo come punto di partenza, ci concentriamo su quello che sarà il passaggio più critico della disaggregazione: la scelta di γ.

6.2.2 Il Boxstep ed i Lower Bound

Per quanto scegliere il Boxstep spesso riduca i tempi su molte delle istanze che abbiamo analizzato (come vedremo nel momento del confronto con il Quadratic), questo tende ad avere dei "picchi" su quelle provenienti dal calcolo del Lower Bound.

Quando abbiamo descritto la disposizione delle istanze abbiamo accennato al fatto che quelle generate nel Lower Bound tendono ad essere le più difficili; a maggior ragione se il caso in esame è tra i più grandi a disposizione (BDSA_b1, VBDSA_cc, ...). Su queste (anche dopo la fase di tuning) il Boxstep ha tempistiche che lo rendono impraticabile. Per questo nell'analisi che segue abbiamo impiegato sulle istanze appena descritte solo la versione con il Quadratic.

Nella fase finale riprenderemo il discorso e vedremo se sarà necessario fare qualche aggiustamento anche su casi di dimensioni più contenute.

6.2.3 Disaggregazione

Nel cercare il livello di disaggregazione, si potrebbe fissare un solo valore per metodo (che a priori potrebbe anche essere lo stesso per tutti); tuttavia è probabile che istanze di dimensioni diverse possano dare risultati migliori con valori di disaggregazione diversi, e che quindi scegliere un unico valore per tutte sia rozzo.

Procediamo nel seguente modo: campioniamo i valori intorno a 50 ad intervalli di 10, lanciamo le varie versioni di un metodo (DQ($\gamma = 10$), DQ($\gamma = 20$),...) e dopo aver diviso le istanze in fasce per dimensione valutiamo l'andamento dei sottometodi, in modo da rendere la disaggregazione adattiva.

Iniziamo dal Quadratic: per quanto riguarda Primale e Net, risulta che i valori migliori siano nell'intervallo 10-60, mentre nel caso di Barrier e Duale non serve andare oltre i 40.

Visto che la suddivisione in componenti interessa solo le colonne attive, abbiamo utilizzato come misura della dimensione *il numero di colonne attive per il numero di righe scoperte*. Per non creare ambiguità con i capitoli precedenti, ci riferiamo a questa quantità come *Attive*Righe* piuttosto che *dimensione* (o *dim*).

Riportiamo quindi i risultati dei quattro casi. Nelle tabelle saranno riportate le medie geometriche dei tempi relativi degli algoritmi disaggregati al variare di γ.

	Bundle$_X$PQ					
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$	$\gamma = 40$	$\gamma = 50$	$\gamma = 60$
$>1e07$	1.90	1.41	1.29	1.16	**1.1**	1.17
$(5e06,1e07]$	1.77	1.33	1.26	1.20	1.15	**1.13**
$(1e06,5e06]$	1.49	1.26	**1.20**	1.25	1.24	1.27
$(5e05,1e06]$	1.37	**1.22**	**1.22**	1.26	1.32	1.32
$(1e05,5e05]$	**1.13**	1.23	1.35	1.50	1.58	1.71
$\leqslant 1e05$	**1.07**	1.31	1.51	1.70	1.8	1.98

	Bundle$_X$DQ			
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$	$\gamma = 40$
$>1e07$	2.97	**1.81**	1.88	2.00
$(5e06,1e07]$	2.41	**1.44**	1.45	1.47
$(1e06,5e06]$	1.61	**1.31**	**1.31**	1.33
$(5e05,1e06]$	1.44	1.25	**1.23**	1.30
$(1e05,5e05]$	**1.11**	1.26	1.40	1.55
$\leqslant 1e05$	**1.05**	1.31	1.53	1.69

	Bundle$_X$NQ					
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$	$\gamma = 40$	$\gamma = 50$	$\gamma = 60$
$>1e07$	1.58	1.31	1.19	1.17	**1.15**	1.17
$(5e06,1e07]$	1.51	1.33	1.31	1.25	1.19	**1.16**
$(1e06,5e06]$	1.38	1.21	**1.18**	1.24	1.25	1.30
$(5e05,1e06]$	1.35	**1.21**	**1.21**	1.27	1.36	1.39
$(1e05,5e05]$	**1.14**	1.22	1.34	1.48	1.63	1.76
$\leqslant 1e05$	**1.08**	1.27	1.46	1.64	1.84	1.99

Da questi risultati otteniamo le seguenti regole per la disaggregazione:

- PQ, NQ:
  - if Attive*Righe $< 5e05 \Rightarrow \gamma = 10$
  - if $5e05 \leqslant$ Attive*Righe $< 5e06 \Rightarrow \gamma = 30$
  - if Attive*Righe $\geqslant 5e06 \Rightarrow \gamma = 50$
- DQ:
  - if Attive*Righe $< 5e05 \Rightarrow \gamma = 10$
  - if $5e05 \leqslant$ Attive*Righe $\Rightarrow \gamma = 20$
- BQ:
  - $\gamma = 10$

Per quanto riguarda il Boxstep invece abbiamo:

	Bundle$_X$BQ			
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$	$\gamma = 40$
$>1e07$	**1.06**	1.20	1.45	1.75
$(5e06,1e07]$	**1.08**	1.19	1.41	1.65
$(1e06,5e06]$	**1.04**	1.24	1.56	1.98
$(5e05,1e06]$	**1.06**	1.39	1.74	2.10
$(1e05,5e05]$	**1.03**	1.53	2.06	2.60
$\leqslant 1e05$	**1.03**	1.44	1.86	2.28

	Bundle$_X$PB			
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$	$\gamma = 40$
$>1e07$	1.73	**1.26**	1.33	1.46
$(5e06,1e07]$	2.07	1.33	**1.21**	1.27
$(1e06,5e06]$	1.75	1.26	**1.20**	1.24
$(5e05,1e06]$	1.55	**1.22**	**1.22**	1.24
$(1e05,5e05]$	1.38	**1.19**	1.23	1.34
$\leqslant 1e05$	**1.16**	1.22	1.29	1.46

	Bundle$_X$DB			
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$	$\gamma = 40$
$>1e07$	1.53	**1.16**	1.20	1.25
$(5e06,1e07]$	1.75	1.27	**1.20**	1.22
$(1e06,5e06]$	1.50	**1.26**	1.27	1.33
$(5e05,1e06]$	1.37	**1.25**	1.31	1.43
$(1e05,5e05]$	1.23	**1.21**	1.36	1.52
$\leqslant 1e05$	**1.11**	1.23	1.35	1.53

	Bundle$_X$NB			
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$	$\gamma = 40$
$>1e07$	1.55	**1.14**	1.15	1.34
$(5e06,1e07]$	1.87	1.28	1.23	**1.19**
$(1e06,5e06]$	1.64	1.29	**1.27**	1.30
$(5e05,1e06]$	1.48	**1.35**	1.37	1.48
$(1e05,5e05]$	1.31	**1.25**	1.37	1.54
$\leqslant 1e05$	**1.11**	1.22	1.37	1.51

	Bundle$_X$BB		
Attive*Righe	$\gamma = 10$	$\gamma = 20$	$\gamma = 30$
$>1e07$	1.29	**1.12**	1.25
$(5e06,1e07]$	1.35	**1.19**	1.37
$(1e06,5e06]$	**1.13**	1.22	1.53
$(5e05,1e06]$	**1.06**	1.42	1.99
$(1e05,5e05]$	**1.08**	1.37	1.83
$\leqslant 1e05$	**1.07**	1.31	1.59

Da questi risultati otteniamo le seguenti regole per la disaggregazione:

- PB, NB:
  - if Attive*Righe $< 1e05 \Rightarrow \gamma = 10$
  - If $1e05 \leqslant$ Attive*Righe $< 1e06 \Rightarrow \gamma = 20$
  - if Attive*Righe $\geqslant 1e06 \Rightarrow \gamma = 30$
- DB:
  - if Attive*Righe $< 1e05 \Rightarrow \gamma = 10$
  - If $1e05 \leqslant$ Attive*Righe $< 5e06 \Rightarrow \gamma = 20$
  - if Attive*Righe $\geqslant 5e06 \Rightarrow \gamma = 30$
- BB:
  - if Attive*Righe $< 5e06 \Rightarrow \gamma = 10$
  - if Attive*Righe $\geqslant 5e06 \Rightarrow \gamma = 20$

6.2.4 Tuning

In fase di tuning proviamo gli stessi valori che avevamo proposto per il caso aggregato (5.7). Avendo cercato di differenziare l'analisi dei parametri per ognuno degli otto algoritmi, per riportare i valori avremmo bisogno di otto tabelle analoghe alla 5.8. Questo non farebbe altro che generare confusione, ci limitiamo quindi a riportare i risultati.

Come accadeva nel caso aggregato, in generale modificare *m3* non è conveniente, mentre cambiare il segno di *m1* migliora leggermente i tempi (e quindi assumiamo $m1 = -0.1$ per tutti i metodi). Per quanto riguarda invece i singoli metodi:

- Quadratic, Barrier: cambiamo *tInit* e *BPar1*. Il primo è molto più incisivo del secondo, comunque la combinazione dei due porta in alcuni casi a ridurre i tempi in modo significativo (quasi della metà)

- Quadratic, Primale: qui invece cambiare il valore di *tInit* ne aumenta i tempi; tuttavia se questo non viene modificato l'algoritmo è *instabile*, nel senso che può accadere entri in dei loop che non gli permettono di arrivare ad una soluzione. Quindi assumiamo $tInit = 1$

- Boxstep: cambiare stabilizzatore modifica di molto il comportamento degli algoritmi. Per tutti e quattro è fondamentale modificare *BPar1* e *tSPar2*, altrimenti accade spesso che non si riesca a calcolare la soluzione di un'istanza. Nel caso aggregato avevamo accennato che avremmo provato $BPar1 = 20$ e che si sarebbero potuti considerare anche valori più alti; in questo caso abbiamo dovuto utilizzare $BPar1 = 500$ per essere ragionevolmente sicuri che non ci fossero problemi in fase di calcolo. Tuttavia in questo modo aumenta il costo del Master Problem, e può capitare che il costo di alcune istanze aumenti.

6.2.5 Risultati

Mettendo insieme i risultati delle sezioni precedenti, cerchiamo "il miglior Quadratic" ed "il miglior Boxstep"; metteremo poi questi a confronto per determinare il meta-algoritmo del Bundle$_X$.

	Bundle$_X$			
Attive*Righe	PQ	DQ	NQ	BQ
$>1e07$	2.03	6.18	1.90	**1.08**
$(5e06,1e07]$	1.89	3.14	1.71	**1.32**
$(1e06,5e06]$	1.33	1.70	**1.24**	1.30
$(5e05,1e06]$	1.26	1.41	**1.17**	1.30
$(1e05,5e05]$	1.22	**1.1**	1.16	1.38
$\leqslant 1e05$	1.24	**1.08**	1.18	1.37

Tabella 6.4: Analisi del Quadratic

	Bundle$_X$			
Attive*Righe	PB	DB	NB	BB
$>1e07$	**1.16**	1.34	1.49	1.71
$(5e06,1e07]$	**1.17**	1.27	1.38	2.43
$(1e06,5e06]$	**1.13**	1.39	1.69	2.76
$(5e05,1e06]$	**1.06**	1.42	1.86	2.57
$(1e05,5e05]$	**1.11**	1.36	1.49	3.13
$\leqslant 1e05$	**1.16**	1.20	1.29	2.97

Tabella 6.5: Analisi del Boxstep (BDSA-VBDSA)

Nelle tabelle 6.4 e 6.5 sono riportate le medie geometriche dei tempi relativi divise per fasce. Quella relativa al Boxstep non tiene conto di tutte le istanze, ma solo di quelle dei casi BDSA e VBDSA; questo perché il caso VSA è qualitativamente diverso ed è stato quindi trattato a parte. Ricordiamo inoltre che non sono considerate quelle appartenenti al Lower Bound dei casi più grandi. Da queste otteniamo (ci limitiamo ad indicare l'algoritmo, senza riportare le specifiche su parametri/disaggregazione):

- Quadratic:
  - if Attive*Righe $< 5e05 \Rightarrow$ `Duale`
  - if $5e05 \leqslant$ Attive*Righe $< 5e06 \Rightarrow$ `Net`
  - if Attive*Righe $\geqslant 5e06 \Rightarrow$ `Barrier`
- Boxstep (BDSA-VBDSA):
  - `Primale`
- Boxstep (VSA):
  - if Attive*Righe $< 5e05 \Rightarrow$ `Duale`
  - if $5e05 \leqslant$ Attive*Righe $< 5e06 \Rightarrow$ `Primale`
  - if Attive*Righe $\geqslant 5e06 \Rightarrow$ `Net`

Nel caso del Quadratic la scelta degli algoritmi è del tutto analoga a quella del Capitolo 4: Barrier sui casi grandi, Net su quelli intermedi e Duale su quelli piccoli; il Boxstep invece si comporta in modo completamente diverso dando maggiore importanza al Primale.

	Bundle$_X$	
Attive*Righe	Boxstep	Quadratic
$>1e07$	**1.15**	1.89
$(5e06,1e07]$	**1.12**	2.55
$(1e06,5e06]$	**1.08**	2.84
$(5e05,1e06]$	**1.05**	2.61
$(1e05,5e05]$	**1.06**	2.28
$\leqslant 1e05$	**1.09**	1.63

Tabella 6.6: Confronto tra Boxstep e Quadratic

Confrontando infine Boxstep e Quadratic (6.6) nel solito modo risulta che il primo sia sempre una scelta migliore rispetto al secondo. Tuttavia i tempi totali non sempre rispecchiano questa decisione: a volte, soprattutto su problemi medio-grandi, accade che il Quadratic impieghi meno. Il motivo è che alcuni problemi presentano istanze costose (sull'ordine delle centinaia o migliaia di secondi) su cui lo stabilizzatore lineare impiega più tempo del quadratico. Essendo queste poche, una o due per problema (almeno in quelle di Test(Selezione)), la media geometrica non ne risente; tuttavia al livello di tempo totale l'impatto è evidente.

Come avevamo già notato queste istanze problematiche, se presenti, si trovano nel calcolo del Lower Bound; quindi *estendiamo l'utilizzo del Quadratic alle istanze appartenenti al Lower Bound di tutti i casi.* Su quelli più piccoli il tempo può aumentare di qualche secondo, ma su quelli medi e grandi ne guadagniamo migliaia.

Indicato con Bundle$_X$MA questo meta-algoritmo, che utilizza il Quadratic nel calcolo del Lower Bound ed il Boxstep su tutte le altre (entrambi seguendo le regole di algoritmo/disaggregazione/parametri viste nelle sezioni precedenti), non ci resta che confrontarlo con Clp e Cplex sui problemi interi.

Ci sono diverse osservazioni da fare sui risultati in 6.7. Infatti

- Considerando quanto il cambiare l'MPSolver avesse aumentato i tempi, il fatto che il caso disaggregato (con solutore generico) riesca in alcuni casi ad impiegare molto meno dell'aggregato (con solutore specializzato) è un risultato significativo

Caso	Clp_D	Cplex_MA	Bundle	Bundle$_X$MA
BDSA_1	2007.6	335.0	33554.4	5506.7
BDSA_b2	28.5	20.7	670.8	517.6
BDSA_c2	94.1	81.9	2441.3	505.2
VBDSA_3	559.0	157.5	5803.9	7088.4
VBDSA_c2	56.7	23.2	1088.6	365.6
VBDSA_f	11.0	7.2	203.9	55.6
VSA_cc2	1871.9	273.2	82801.9	17024.0
BDSA_b1	967.5	393.1	33002.5	56138.4
BDSA_cc	94.17	81.91	13734.0	72438.5

Tabella 6.7: Confronto del Bundle Disaggregato (Cplex) con Clp/Cplex e Bundle Aggregato

- È in fase di sviluppo una versione del solutore specializzato che ne estenda le funzionalità al caso disaggregato. Questo dovrebbe, considerando i risultati di inizio capitolo, ridurre ulteriormente i tempi rendendo il nuovo approccio più competitivo; chiaramente in questa fase non è possibile fare un qualche tipo di valutazione

- Sui casi particolarmente grandi (come i due in fondo, separati dal resto) la versione disaggregata è particolarmente lenta. La colpa è nuovamente delle istanze del Lower Bound, sulle quali il Boxstep è risultato inutilizzabile ma anche il Quadratic non dà prestazioni accettabili

- Per quanto il Bundle$_X$MA possa reggere il confronto con il Bundle aggregato, per poterlo utilizzare stiamo supponendo di avere a disposizione il Cplex. Ma allora non avremmo alcuna utilità a scegliere questa versione del Bundle, in quanto nei casi in esame il Cplex continua ad essere di gran lunga il miglior metodo per ottenere soluzioni precise

- D'altra parte su casi (non presenti nel nostro studio) in cui è il Bundle ad essere migliore del Cplex, questo nuovo approccio potrebbe essere valido

6.3 Clp

Visti i risultati ottenuti con il Cplex, difficilmente utilizzare il Clp nella disaggregazione può essere un'alternativa valida al Bundle o al Clp_D.

Inoltre nell'analizzare il Clp ci siamo trovati davanti ad un ostacolo: la scarsa documentazione reperibile sull'uso del codice. Questo comporta che non è stato possibile risolvere un problema rilevante che ci si è presentato; non siamo riusciti a cambiare l'algoritmo utilizzato nella risoluzione. In questo modo ci siamo dovuti limitare ad utilizzare il solutore di default, il Simplesso Duale.

Da un lato questo era risultato il migliore nel Capitolo 4, tuttavia come abbiamo appena visto nel caso Cplex Boxstep non è assolutamente detto che questo comportamento si mantenga.

Per quanto riguarda la fase di tuning, i risultati sono analoghi al corrispettivo del Cplex. Tuttavia l'algoritmo risulta nel suo complesso più instabile, presentando un gran numero di istanze su cui i tempi "esplodono". Questo, in aggiunta al fatto che anche su istanze medie e piccole i tempi sono elevati, scoraggiano l'utilizzo del Bundle_C.

Per quanto probabilmente ci sia un qualche margine di miglioramento, abbiamo preferito chiudere qui lo studio della disaggregazione e passare all'analisi di alcuni algoritmi di tipo Simplesso.

Capitolo 7

Subgradiente

Al contrario del Bundle nell'utilizzare il Subgradiente non imporremo un vincolo forte sulla precisione della soluzione. Questo perché in generale un algoritmo di questo tipo è molto rapido nella fase iniziale del calcolo, generando soluzioni approssimate molto rapidamente; ma non è poi in grado di avvicinarsi alla soluzione esatta in tempi ragionevoli. Per questo il Subgradiente può essere utilizzato come *warm starter* per altri metodi; ovvero la soluzione generata dopo un certo numero di iterazioni può essere ripresa da altri metodi (come il Bundle) come punto di partenza.

In ogni caso manterremo gli stessi valori dei parametri che regolano la precisione, *tStar* ed *EpsLin*, del caso del Bundle. Come vedremo meglio tra poco, questo comunque non basta a garantire che riusciremo a trovare una soluzione relativamente esatta.

Inoltre i metodi di tipo Subgradiente hanno bisogno di un valore sensato che faccia da lower bound per la soluzione. Nel nostro caso abbiamo a disposizione la soluzione esatta e quindi utilizziamo questa; in questo modo però stiamo simulando una situazione poco realistica, in quanto (ovviamente) se sappiamo già la soluzione non c'è alcun bisogno di lanciare altri solutori.

In questo capitolo ci stiamo basando su di un lavoro di A. Frangioni, E. Gorgone, B. Gendron già precedentemente menzionato [14]. Anche per quanto riguarda il codice, stiamo utilizzando il Subgradiente scritto da A. Frangioni e E. Gorgone

(Dipartimento di Elettronica, Informatica e Sistemistica, Università della Calabria) che è stato utilizzato per produrre i risultati del suddetto articolo.

Diamo adesso una descrizione più approfondita dello schema generale.

7.1 Componenti del Subgradiente

Per definire uno specifico metodo, è necessario fissare un certo numero di elementi. Anche se in generale queste scelte interagiscono tra di loro, possono essere descritte indipendentemente; vediamo brevemente le varie componenti.

Stepsize Un'aspetto cruciale è la scelta dell'ampiezza del passo (*Stepsize Rules*), che avevamo indicato con ν_i. Una proprietà interessante è che questo valore può essere scelto senza alcuna conoscenza della funzione che andiamo a minimizzare; si può dimostrare infatti che qualunque scelta dell'ampiezza che soddisfi la condizione

$$\sum_{i=1}^{\infty} \nu_i = \infty, \quad \sum_{i=1}^{\infty} \nu_i^2 < \infty,$$

ad esempio $\nu_i = 1/i$, porta ad un algoritmo convergente. Per quanto questo dimostri la robustezza del Subgradiente, che è in grado di convergere senza informazioni riguardo la funzione, in pratica è meglio utilizzare regole meno generali per avere prestazioni migliori.

Una delle prime Stepsize Rules introdotte, proposta da Polyak nel 1969 [15], è definita come

$$\nu_i = \frac{\beta_i (f_i - f_*)}{||z_i||^2}, \tag{7.1}$$

per un dato parametro $\beta_i \in (0, 2)$. Tuttavia qui stiamo assumendo di conoscere f_*, l'ottimo della funzione, e che la direzione del passo sia quella di z_i, cosa non sempre vera. Quindi viene utilizzata una versione più generale della regola, della forma

$$\nu_i = \frac{\beta_i(f_i - f_i^{lev})}{||d_i||^2}, \tag{7.2}$$

nella quale f_i^{lev} è una qualche approssimazione di f^* disponibile al passo i e d_i dipende dalla tecnica di deflessione utilizzata.

Come determinare f_i^{lev} è un passo cruciale di questa regola. In generale durante la minimizzazione di f si ha disposizione solo l'upper bound $f_i^{rec} = \min\{f_l, l = 1, \ldots, i\}$ di f^* ma nessun lower bound. Noi in realtà sappiamo il valore esatto di f^*, ma anche se così non fosse nei casi presi in esame sarebbe possibile trovarne un lower bound. A seconda di come vengono scelti β_i e f_i^{lev} distinguiamo tre regole:

1. *Polyak*: la regola originaria introdotta da Polyak in [15], nella quale β_i e f_i^{lev} sono indipendenti da i

2. *ColorTV*: il nome è dovuto al fatto che associamo un colore ad un'iterazione a seconda del miglioramento $\Delta f_i = f_{i-1} - f_i$; un'iterazione è *verde* se Δf_i è positivo e la direzione precedente è ancora di decrescita, *gialla* se Δf_i è positivo ma è necessario cambiare direzione, e *rossa* se Δf_i è negativo. A questo punto il parametro β_i dipende dalle iterazioni consecutive di uno stesso colore: viene aumentato significativamente dopo una sequenza di verdi, leggermente aumentato dopo una sequenza di gialli e ridotto dopo una sequenza di rossi. f_i^{lev} invece viene aggiornato non appena $f_i < f_i^{lev}$, ovvero abbiamo dimostrato che non è un lower bound valido

3. *FumeroTV*: questa regola modifica sia f_i^{lev} che β_i. Inizialmente f_i^{lev} viene posto pari al valore del lower bound a disposizione, e con il procedere dell'algoritmo viene modificato tenendo conto di f_i^{rec}. Il comportamento di β_i invece è diviso in due fasi: nella prima questo viene diminuito dopo un numero consecutivo di iterazioni che non migliorano la soluzione e non viene mai incrementato. Nella seconda invece viene diminuito allo stesso modo, ma è anche previsto un aumento dopo un numero consecutivo di iterazioni che migliorano la soluzione.

Deflessione La non differenziabilità di f è la causa principale dell'oscillazione dei punti generati, per i quali può capitare che $z_i \approx -z_{i-1}$. In questo modo due passi ν_i e ν_{i-1} "ragionevolmente lunghi" si traducono in un unico passo molto corto, riducendo così la velocità di convergenza. Una soluzione a questo problema è quella di deflettere il subgradiente modificando la ricorrenza in $\hat{x}_{i+1} = x_i - \nu_i d_i$. Una formula generale per la scelta del parametro d_i è

$$d_i = \alpha_i z_i + (1 - \alpha_i) d_{i-1}, \tag{7.3}$$

per un dato *parametro di deflessione* $\alpha_i \in [0, 1]$. Esistono anche altre regole per la scelta della deflessione (*Deflection Rules*), ma ci limitiamo ad analizzarne di questa forma. O meglio, consideriamo:

1. *STSubgrad*: poniamo $\alpha_i = 1$, ovvero ci limitiamo al subgradiente standard senza applicare alcuna deflessione

2. *Volume*: α_i viene scelto come la soluzione ottima di un problema quadratico unidimensionale che, in pratica, cerca di trovare la combinazione convessa di z_i e d_{i-1} di minima norma.

Proiezione C'è anche un'altra causa (indipendente dalla precedente) di oscillazione: la direzione non tiene conto dell'insieme ammissibile X. Nel caso in cui x_i sia sul bordo di X e z_i (o d_i) sia praticamente ortogonale alla sua frontiera, x_{i+1} può restare molto vicino al punto precedente anche per valori grandi del passo ν_i; di nuovo, questo riduce la velocità di convergenza. Per evitare questo problema si può proiettare z_i (o d_i) sul *cono tangente* T_i ad X in x_i. In questo modo "correggiamo" la direzione garantendo un passo più significativo.

Tuttavia non siamo interessati a questo elemento, in quanto il problema a cui applichiamo il Subgradiente non è vincolato.

7.2 Test

Per quanto riguarda i parametri da impostare, abbiamo utilizzato i risultati riportati nell'articolo [14]. In questo venivano analizzati due diversi rilassamenti lagrangiani di un problema di flusso, il *Flow Relaxation* ed il *Knapsack Relaxation*. I problemi che dobbiamo risolvere sono in qualche modo affini a quest'ultimo, perciò utilizzeremo i settaggi risultati ottimi per questo caso.

Possiamo quindi passare direttamente ai test. Proviamo il Subgradiente con le tre Stepsize Rules descritte e Volume come Deflection Rule; indichiamo questi tre metodi con SubGrad_C (ColorTV), SubGrad_F (FumeroTV) e SubGrad_P (Polyak).

Nel calcolare le soluzioni è necessario imporre un limite L sul numero di iterazioni, a causa della difficoltà del Subgradiente nel convergere. Abbiamo quindi deciso di confrontare i tre algoritmi in modo diverso dal solito: abbiamo fissato non uno ma otto valori limite (100, 200, 300, 500, 1000, 2000, 4000, 8000) e per ciascuno di essi abbiamo confrontato il gap della soluzione ottenuta adottando quel limite ed il tempo impiegato.

In questo modo possiamo farci un'idea più precisa di cosa accade nelle varie fasi dei tre metodi. A seconda delle esigenze infatti, un utente potrebbe preferire un metodo inizialmente più rapido oppure uno che produce soluzioni più precise raggiunto il limite massimo di iterazioni.

Riportiamo i risultati su alcuni casi test di tre dei valori di L contemplati: 1000 (7.1), 2000 (7.2) e 8000 (7.3).

Da questi risultati otteniamo che il SubGrad_F è il più competitivo sia in termini di precisione che (soprattutto) in termini di tempo. Inoltre i suoi valori sono molto simili tra le varie tabelle perché, contrariamente alle aspettative, spesso questo metodo riesce a convergere in meno di 8000 iterazioni.

Tuttavia c'è da dire che quest'ultimo dato non è necessariamente significativo: su alcune istanze accade che il SubGrad_F termini in meno di 8000 iterazioni ma la soluzione prodotta abbia un gap relativamente elevato. Quindi questo metodo è particolarmente indicato nel caso in cui il tempo sia il fattore fondamentale; se invece si vogliono ottenere soluzioni migliori è meglio utilizzare uno degli altri due

	SubGrad$_C$		SubGrad$_F$		SubGrad$_P$	
$L = 1000$	t	Gap	t	Gap	t	Gap
BDSA_1	202.65	$1.00e{-4}$	157.28	$7.75e{-5}$	160.82	$4.96e{-4}$
BDSA_b1	178.93	$2.15e{-5}$	129.98	$1.93e{-5}$	193.88	$2.86e{-5}$
BDSA_b2	30.76	$2.59e{-5}$	18.56	$2.54e{-5}$	33.31	$3.90e{-5}$
BDSA_c2	76.02	$4.74e{-5}$	38.12	$4.96e{-5}$	23.41	$2.58e{-4}$
VBDSA_3	71.07	$9.22e{-5}$	42.23	$8.98e{-5}$	79.35	$1.92e{-4}$
VBDSA_c2	161.89	$2.53e{-5}$	107.86	$1.87e{-5}$	146.60	$5.04e{-5}$
VBDSA_f	4.09	$2.67e{-5}$	3.11	$2.27e{-5}$	2.34	$2.99e{-5}$
VSA_cc2	165.40	$1.00e{-5}$	125.44	$8.75e{-6}$	102.83	$2.16e{-5}$

Tabella 7.1: Tempo e Gap dei vari subgradienti con limite 1000

	SubGrad$_C$		SubGrad$_F$		SubGrad$_P$	
$L = 2000$	t	Gap	t	Gap	t	Gap
BDSA_1	351.61	$9.53e{-5}$	163.37	$7.67e{-5}$	311.31	$3.86e{-4}$
BDSA_b1	230.93	$2.11e{-5}$	141.85	$1.93e{-5}$	364.10	$2.83e{-5}$
BDSA_b2	50.07	$2.45e{-5}$	24.08	$2.23e{-5}$	65.12	$3.40e{-5}$
BDSA_c2	124.36	$4.67e{-5}$	41.73	$4.82e{-5}$	36.33	$2.51e{-4}$
VBDSA_3	115.22	$8.69e{-5}$	44.53	$8.83e{-5}$	155.57	$1.19e{-4}$
VBDSA_c2	270.54	$2.40e{-5}$	113.49	$1.85e{-5}$	288.18	$3.87e{-5}$
VBDSA_f	4.60	$2.67e{-5}$	3.11	$2.27e{-5}$	3.62	$3.04e{-5}$
VSA_cc2	318.16	$9.63e{-6}$	172.97	$8.40e{-6}$	197.72	$1.80e{-5}$

Tabella 7.2: Tempo e Gap dei vari subgradienti con limite 2000

	SubGrad$_C$		SubGrad$_F$		SubGrad$_P$	
$L = 8000$	t	Gap	t	Gap	t	Gap
BDSA_1	1158.34	$9.44e{-5}$	163.37	$7.67e{-5}$	1195.82	$2.68e{-4}$
BDSA_b1	504.15	$2.09e{-5}$	148.68	$1.92e{-5}$	1336.50	$2.73e{-5}$
BDSA_b2	164.63	$2.18e{-5}$	27.55	$2.19e{-5}$	251.90	$2.69e{-5}$
BDSA_c2	332.93	$4.66e{-5}$	41.73	$4.82e{-5}$	112.29	$2.25e{-4}$
VBDSA_3	350.70	$8.36e{-5}$	49.99	$8.83e{-5}$	613.51	$7.69e{-5}$
VBDSA_c2	851.32	$2.33e{-5}$	113.81	$1.83e{-5}$	1139.03	$3.11e{-5}$
VBDSA_f	7.70	$2.67e{-5}$	3.11	$2.27e{-5}$	11.25	$3.01e{-5}$
VSA_cc2	1174.13	$8.87e{-6}$	210.16	$8.28e{-6}$	747.28	$1.15e{-5}$

Tabella 7.3: Tempo e Gap dei vari subgradienti con limite 8000

aumentando la soglia L.

7.3 Subgradiente Incrementale

Nel passare al Subgradiente Incrementale ci siamo trovati di fronte a due difficoltà, una pratica ed una teorica.

Essendo stato scritto dagli stessi autori, parte del codice del Subgradiente è in comune con quello del Bundle (principalmente le classi base). Tuttavia c'è una componente fondamentale che viene chiamata dai due solver in modo diverso: l'oracolo.

O meglio, nel caso del Subgradiente aggregato non ci sono problemi ad utilizzare quello messo a disposizione dalla MAIOR; tuttavia questo non contemplava il caso disaggregato e siamo stati noi a modificarlo per includere questa possibilità. Per quanto funzionasse bene nel caso del Bundle, con il passaggio all'Incrementale è stato necessario modificarlo ulteriormente.

Al livello teorico invece, il calcolo dello stepsize dei passi incrementali richiede una stima della costante di Lipschitz globale del problema. Per trovarne una si può procedere come segue: per ogni componente in cui abbiamo diviso il problema, sommiamo tutte le colonne che ne fanno parte e calcoliamo le norme di queste colonne risultanti; il massimo delle norme può essere utilizzata come costante di Lipschitz globale. Tuttavia questo parametro, regolando l'ampiezza del passo, può influire molto sulla convergenza del metodo.

Non sappiamo esattamente quale dei due fattori abbia influito maggiormente, il risultato è stato che non siamo riusciti ad utilizzare il Subgradiente Incrementale in maniera soddisfacente. Avendo comunque raccolto una quantità significativa di dati, abbiamo optato per abbandonare questo solutore (potenzialmente interessante) e passare alla fase conclusiva del lavoro.

Capitolo 8

Conclusioni

Concludiamo questo lavoro di tesi confrontando tra di loro i vari metodi che abbiamo ottenuto al termine dei vari capitoli. Per entrare più nel dettaglio poniamo (in analogia con quanto già fatto nel caso del Subgradiente) diversi limiti di iterazioni e riportiamo tempi e gap delle soluzioni trovate. Dividiamo inoltre i valori in fasce; in questo modo valutiamo anche l'applicabilità a problemi di diverse dimensioni.

Un problema non banale di queste sperimentazioni è come paragonare le performance di diversi solutori. In analogia con il caso del Subgradiente, abbiamo registrato il tempo ed il gap ottenuto da ogni metodo a differenti limiti di iterazioni L. Per quanto riguarda il Bundle abbiamo fissato (in un certo senso arbitrariamente) i valori

$$L \in \{50, 100, 150, 250, 500, 1000, 2000, 4000\}.$$

Questi devono essere modificati per i vari metodi per tenere conto della velocità del calcolo di un'iterazione. Nel caso del Bundle Disaggregato le singole iterazioni sono molto più lente; per questo metodo scegliamo quindi

$$L \in \{5, 10, 15, 25, 50, 100, 200, 400\}.$$

Al contrario le iterazioni del Subgradiente sono molto più veloci, e quindi poniamo

$$L \in \{100, 200, 300, 500, 1000, 2000, 4000, 8000\}$$

I valori esatti dei limiti in realtà non sono importanti, in quanto li usiamo per poter rappresentare la velocità di convergenza dei diversi metodi intesa come tempo necessario per raggiungere un gap fissato con il (noto) valore ottimo.

Tuttavia dobbiamo in qualche modo tenere conto del fatto che istanze di dimensioni diverse si comporteranno in modo molto diverso, quindi i risultati devono essere aggregati di conseguenza. Riprendiamo allora la divisione effettuata nel capitolo 6, che utilizzava la quantità Attive*Righe per determinare sei gruppi di istanze.

Per ciascuno abbiamo considerato il tempo necessario ai diversi solutori per calcolare tutte le istanze del gruppo e la media geometrica dei gap, al variare del limite L. Tutto questo ci permette di confrontare i tre diversi metodi mostrando (per ogni gruppo) il variare della precisione al crescere del tempo.

	Bundle		BundleDis		Subgrad		Clp	Cplex
Attive*Righe $\leqslant 1e05$								
L	39.6	2.80e-3	26.0	6.64e-4	65.5	4.91e-5	15.6	11.3
	73.4	1.81e-4	47.1	7.25e-5	122.5	2.97e-5		
	103.4	1.95e-5	66.4	9.48e-6	172.2	2.33e-5		
	153.5	1.54e-6	94.4	6.40e-8	229.3	1.66e-5		
	234.3	1.54e-7	129.2	8.76e-10	262.0	1.48e-5		
	318.1	7.09e-8	146.7	3.20e-10	276.1	1.44e-5		
	414.2	5.56e-8	171.9	2.07e-10	283.3	1.43e-5		
	554.1	4.54e-8	232.3	1.97e-10	286.6	1.43e-5		
$1e05 <$ Attive*Righe $\leqslant 5e05$								
L	30.2	8.30e-2	38.2	1.87e-3	53.5	2.68e-4	51.1	37.4
	58.7	2.05e-2	79.8	3.08e-4	97.8	1.85e-4		
	87.3	8.64e-3	121.3	6.58e-5	134.4	1.36e-4		
	144.0	2.24e-3	198.5	5.71e-6	179.9	1.09e-4		
	281.1	1.19e-4	314.3	1.01e-8	203.7	1.05e-4		
	488.2	3.75e-6	375.7	2.31e-9	208.1	1.05e-4		
	653.5	1.04e-6	401.5	1.85e-9	211.3	1.05e-4		
	793.4	7.65e-7	434.0	1.77e-9	212.5	1.04e-4		

	Bundle		BundleDis		Subgrad		Clp	Cplex
$5e05 <$ Attive*Righe $\leqslant 1e06$								
L	24.3	2.47e-1	43.5	3.44e-3	48.3	7.51e-4	73.6	63.7
	47.1	6.30e-2	97.2	7.62e-4	86.1	5.76e-4		
	69.8	3.00e-2	158.7	1.65e-4	117.5	4.41e-4		
	115.2	1.02e-2	295.0	1.60e-5	160.4	3.42e-4		
	228.5	1.62e-3	542.8	1.72e-8	187.1	3.21e-4		
	459.7	1.36e-4	680.2	1.22e-9	193.2	3.20e-4		
	849.4	1.00e-5	744.1	7.70e-10	198.4	3.19e-4		
	1278.6	2.61e-6	776.9	5.08e-10	204.2	3.19e-4		
$1e06 <$ Attive*Righe $\leqslant 5e06$								
L	131.1	1.07e00	229.2	1.21e-2	271.17	2.06e-3	602.1	340.1
	255.9	3.10e-1	642.0	3.62e-3	489.06	1.40e-3		
	380.2	1.46e-1	1256.4	1.62e-3	688.71	9.85e-4		
	627.1	5.51e-2	2848.1	3.28e-4	1023.71	6.54e-4		
	1261.0	1.21e-2	5991.2	6.13e-7	1464.55	5.66e-4		
	2580.3	1.57e-3	9131.1	2.22e-8	1601.79	5.56e-4		
	5288.1	9.54e-5	10457.5	8.37e-9	1684.04	5.55e-4		
	9769.4	5.53e-6	11830.3	6.50e-9	1765.89	5.55e-4		
$5e06 <$ Attive*Righe $\leqslant 1e07$								
L	112.3	1.85e00	275.3	1.78e-2	259.0	6.99e-3	1358.8	348.4
	220.0	6.31e-1	797.4	7.34e-3	507.7	4.63e-3		
	327.8	3.31e-1	1623.1	2.67e-3	764.6	3.24e-3		
	542.8	1.46e-1	3890.1	4.84e-4	1132.4	2.30e-3		
	1085.5	3.61e-2	7657.8	1.82e-7	1529.1	2.04e-3		
	2213.5	7.23e-3	8941.0	2.42e-9	1781.7	2.02e-3		
	4552.1	7.93e-4	10087.4	1.64e-9	2054.7	2.01e-3		
	9414.1	5.62e-5	14014.0	1.60e-9	2349.4	2.00e-3		
Attive*Righe $> 1e07$								
L	222.3	3.33e00	7446.3	3.83e-2	472.8	1.48e-2	9413.2	1506.3
	437.9	1.36e00	20053.8	1.88e-2	939.6	1.04e-2		
	653.8	7.59e-1	34640.0	1.35e-2	1393.4	7.52e-3		
	1085.7	3.32e-1	68382.5	5.85e-3	2232.0	4.86e-3		
	2184.4	9.17e-2	125818.6	5.38e-7	3424.4	3.84e-3		
	4461.2	2.42e-2	206541.8	1.71e-8	4242.8	3.73e-3		
	9218.9	4.88e-3	297530.8	9.23e-9	5256.7	3.66e-3		
	19447.6	6.43e-4	450132.6	7.07e-9	6759.8	3.59e-3		

Tabella 8.1: Confronto finale al variare del limite L

Nel riportare i risultati abbiamo disposto i gruppi in ordine crescente del valore Attive*Righe ed al loro interno medie e tempi in ordine crescente del limite L.

A causa del gran numero di istanze e metodi da considerare, avere solo questa tabella non è molto utile a causa della sua scarsa leggibilità. Consideriamo allora anche sei rappresentazioni grafiche, una per gruppo: in ciascuna di queste riporteremo la media dei gap in funzione del tempo dei vari metodi. Dalla tabella possiamo comunque fare alcune osservazioni:

- Sull'insieme più piccolo (il primo presentato) la situazione è chiara: Clp e Cplex impiegano meno degli altri solver con il minimo limite di iterazioni. Evitiamo quindi di riportarne il grafico
- Nel secondo in realtà il primo valore di Bundle e Bundle Disaggregato è inferiore al tempo del Clp, ma solo il primo. Come nell'insieme precedente allora non varrebbe la pena di utilizzare solutori diversi da quelli esatti e quindi non ne riportiamo il grafico
- Per quanto riguarda l'ultimo invece è evidente come il Bundle Disaggregato, pur raggiungendo una precisione maggiore, non sia assolutamente comparabile agli altri metodi. C'era da aspettarsi un comportamento di questo genere considerando le difficoltà che aveva mostrato su diverse istanze dei casi grandi; riportiamo comunque questi valori.

Inoltre nei vari grafici abbiamo inserito due linee ad indicare il tempo impiegato dal Clp (linea blu) e dal Cplex (linea rossa). In questo modo possiamo sempre tenere d'occhio i due valori di riferimento.

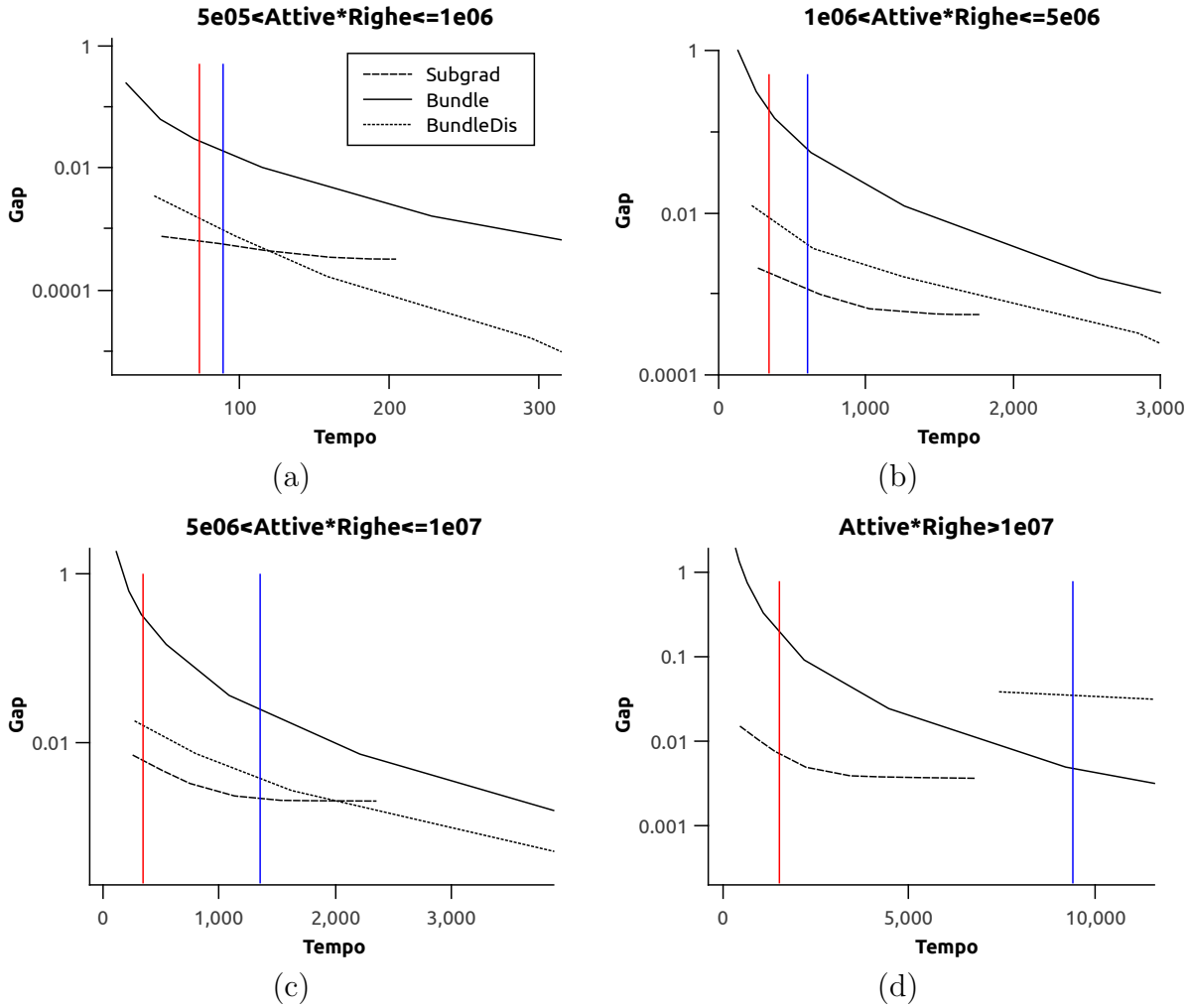


Figura 8.1: Risultati dei quattro gruppi rilevanti

Da queste possiamo infine concludere che:

- Non ci sono differenze sostanziali tra i vari gruppi, il comportamento relativo dei metodi è molto simile nelle quattro figure. L'unica eccezione è ovviamente il Bundle Disaggregato, che non è applicabile sulle istanze di grandi dimensioni (d)

- L'andamento delle versioni aggregata e disaggregata del Bundle è simile, tuttavia a parità di tempo la precisione del secondo (quando applicabile) è molto maggiore

- Il subgradiente è sempre il migliore quando L (e quindi il tempo di calcolo) è basso, ma converge molto lentamente diventando peggiore del Bundle Disaggregato dopo una soglia relativamente bassa. Considerando però che la versione attuale del Bundle Disaggregato prevede l'uso di software a pagamento, il Subgradiente sarebbe probabilmente la scelta migliore per un utente interessato a limitare i tempi (come ad esempio la MAIOR)

Abbiamo considerato un caso concreto ed abbiamo cercato di svilupparne un'analisi il più completa possibile, tuttavia c'è un certo margine di miglioramento. È possibile che nel Bundle Disaggregato ci sia sfuggito qualcosa che lo rendesse migliore sulle istanze di grandi dimensioni; ma d'altra parte lo sviluppo di una componente MPSolver specializzata che comprenda il caso disaggregato potrebbe rendere superfluo un approfondimento in quella direzione.

Il Subgradiente è risultato molto buono senza che venisse fatta una fase di tuning specifica per i casi presi in esame, è quindi possibile che si potesse ottimizzare ulteriormente. Inoltre, non siamo riusciti a provarne la versione Incrementale.

Sono comunque state mostrate un certo numero di alternative valide, e la nostra speranza è che questo studio possa aiutare un qualche utente che si trova ad affrontare Master Problem derivanti da tecniche di generazione di colonne.

Bibliografia

[1] http://www.di.unipi.it/optimize/Courses/ROM/1415/Appunti1415.pdf

[2] https://www.me.utexas.edu/∼jensen/ORMM/supplements/methods/lpmethod/S4_interior.pdf

[3] L. Bacaud, C. Lemaréchal, A. Renaud e C. Sagastizàbal *Bundle Methods in Stochastic Optimal Power Management: A Disaggregated Approach Using Preconditioners*

[4] S. Carosi *Il Set Partitioning come modello per la formazione di turni del personale in ambito extra-urbano*

[5] F. Bernazzani, S. Carosi, A. Frangioni, A. Gaffi e L. Girardi *Miglioramenti algoritmici nella soluzione di problemi reali di schedulazione di veicoli e personale*

[6] L. Poderico *Riprogettazione del BDSA della Maior utilizzando la SetPartLib*

[7] L. Poderico *Progetto di una famiglia di algoritmi di Set Partitioning con vincoli*

[8] http://www.coin-or.org/

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

[10] P. Carraresi, A. Frangioni e M. Nonato, *Applying Bundle Methods to the Optimization of Polyhedral Functions: An Applications-Oriented Development*

[11] A. Frangioni *Generalized Bundle Methods*

[12] A. Frangioni *Solving Semidefinite Quadratic Problems Within Nonsmooth Optimization Algorithms*

[13] A. Frangioni, E. Gorgone *Bundle Methods for Sum-Functions with "Easy" Components: Applications to Multicommodity Network Design*

[14] A. Frangioni , E. Gorgone, B. Gendron *On the Computational Efficency of Subgradient Methods: a Case Study in Combinatorial Optimization*

[15] B.T. Polyak *Minimization of unsmooth functionals*