UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI ELETTRONICA ED INFORMATICA
TESI DI LAUREA

NUOVE PROCEDURE DI SEPARAZIONE PER LA PROGRAMMAZIONE LINEARE INTERA

Relatore: Prof. Matteo Fischetti

Laureando: Giuseppe Andreello

Padova, 24 marzo 2003

Alla mia famiglia

Indice

Sommario

Problema considerato L'oggetto di questo lavoro di tesi è stato testare l'efficacia dei tagli 0-1/2 usando Ilog CPLEX8, un ambiente per la programmazione linerare intera.

Come è stato risolto È stata studiata una strategia di applicazione del separatore e di selezione dei tagli generati sulla base delle loro caratteristiche geometriche. Una implementazione in C++ di queste nuove funzionalità è stata affiancata all'implementazione originale, in C, della procedura di separazione.

Risultati principali I tagli 0-1/2 sono particolarmente efficaci per le versioni "lp" dei problemi di "satisfiability" e "maximum satisfiability", la cui struttura è di tipo combinatorio.

Più in generale si riescono a generare facilmente e sembrano essere utili per problemi la cui formulazione lp ha un numero di vincoli molto maggiore del numero di variabili.

Nel caso opposto, invece, si generano con molta difficoltà e influenzano poco la soluzione del problema.

Significato dei risultati I tagli 0-1/2 sono utili in casi abbastanza semplici da individuare.

Inoltre, si è dimostrata molto efficace la selezione dei tagli da aggiungere alla formulazione sulla base delle loro proprietà geometriche.

Introduzione

Scopo del lavoro La teoria dei tagli $0-1/2$ è nota da tempo: è stata pubblicata per la prima volta nella tesi di dottorato del professor Caprara (vedi [2]).

Le prove condotte all'epoca, con una implementazione del separatore in linguaggio C e con il software MINTO, hanno purtroppo fornito risultati poco incoraggianti.

Lo scopo principale di questo lavoro di tesi è di studiare il modo di inserire i tagli $0-1/2$ in modo efficace in un Branch&Cut che si basi su Ilog CPLEX8, una moderna libreria software per la soluzione di MIP.

Natura del problema considerato Una caratteristica fondamentale dei tagli $0-1/2$ è che su problemi adatti se ne possono generare migliaia; l'esperienza passata ha insegnato che l'applicazione grezza di una grande quantità di tagli peggiora le prestazioni del Branch&Cut: il lavoro si è quindi incentrato soprattutto sullo studio di criteri di selezione basati sulle caratteristiche geometriche dei tagli.

Contributi reperibili in letteratura Sui criteri di selezione di tagli durante un Branch&Cut si trovano pochissimi lavori: in pratica quello che ha ispirato i criteri usati per i tagli $0-1/2$ è il lavoro di E. Balas, S.Ceria, G. Cornuéiols pubblicato nel 1996 (si veda [5]).

Per misurare gli speedup fra le varie configurazioni e analizzare i risultati delle prove si sono usati i metodi di Bixby et al. descritti alla presentazione della versione 6.5 di CPLEX (si veda [4]).

Per quanto riguarda il test-bed, invece, si è trovata una interessante collezione di problemi di natura combinatoria presso il sito del DIMACS Challenge [3], che si possono agevolmente convertire in istanze di programmazione lineare intera.

Indicazione dei metodi di soluzione del problema Una prima taratura del separatore è stata fatta intuitivamente; in questo modo si è ottenuta una base di partenza buona per una successiva fase di ottimizzazione sistematica su un sottoinsieme delle istanze a disposizione.

Il test-bed è costituito dalle stesse istanze di problemi di *satisfiability* usate in passato dal professor Caprara, in quanto hanno tutte le caratteristiche dei problemi su cui i tagli $0-1/2$ dovrebbero funzionare al meglio.

Grande attenzione è stata dedicata all'implementazione degli algoritmi. Per la generazione dei tagli a partire dal punto frazionario e dalla formulazione IP si è riutilizzata l'implementazione C del Professor Caprara: il suo utilizzo con le librerie di CPLEX e in particolare l'implementazione della selezione dei tagli ha richiesto la scrittura di una discreta quantità di codice. Si è deciso, assieme al collega Gianni Codato che ha lavorato su tematiche analoghe durante la sua tesi di laurea, di implementare un insieme di classi in C++ che possa costituire un valido punto di partenza anche per i futuri lavori del gruppo di ricerca basati su CPLEX. Questo insieme di classi è stato chiamato GBF: General Branch&Cut Framework.

Elenco del contenuto dei capitoli I primi due capitoli descrivono il materiale di partenza: nel capitolo 1 è stata riportata la descrizione teorica dei tagli $0 - 1/2$; nel 2, invece, viene descritto il funzionamento di Ilog CPLEX.

Il capitolo 3 contiene invece la descrizione della metrica di qualità usata per selezionare i tagli e della strategia con cui è stato applicato il separatore stesso.

I risultati computazionali ottenuti sono riportati nel capitolo 4, assieme alla descrizione dei metodi usati per calcolare gli speedup.

La struttura di GBF e i criteri che ne hanno ispirato lo sviluppo sono descritti nel capitolo 5, mentre nel capitolo 6 è riportata parte della documentazione delle classi generata automaticamente a partire dai sorgenti.

Capitolo 1

I tagli $0 - 1/2$

In questo capitolo sono riportati la formulazione completa della teoria dei tagli 0-1/2, tratta da [2], la descrizione dell'implementazione del separatore messo a mia disposizione dal Prof. Caprara e i risultati sperimentali ottenuti in passato.

1.1 Tagli di Chvátal-Gomory $\{0, \frac{1}{2}\}$

Data una matrice intera $m \times n$ $A = (a_{ij})$ ed un vettore intero m-dimensionale b, si definiscano i poliedri $P := \{x \in R^n : Ax \leq b\}$, $P_I := \text{conv}\{x \in Z^n : Ax \leq b\}$, e si assuma $P_I \neq P$; senza perdita di generalità si supporrà che ogni riga della matrice (A, b) contenga almeno un elemento dispari. Un *Taglio di Chvátal-Gomory* (indicato per brevità come *taglio CG*) è una disuguaglianza valida per P_I definita da $\lambda^T Ax \leq \lfloor \lambda^T b \rfloor$, con $\lambda \in R^m_+$ tale che $\lambda^T A \in Z^n$; si può dimostrare che tagli CG non dominati possono essere ottenuti solo quando $\lambda \in [0, 1)^m$.

La *chiusura di rango 1* di P è definita da $P_1 := \{x \in P : \lambda^T Ax \leq \lfloor \lambda^T b \rfloor, \text{ per } \lambda \in [0, 1)^m \text{ tale che } \lambda^T A \in Z^n\}$. Si definisce *taglio di Chvátal-Gomory* $\{0, \frac{1}{2}\}$ (indicato per brevità come *taglio* $\{0, \frac{1}{2}\}$) un taglio CG per il quale $\lambda \in \{0, 1/2\}^m$, e si indica con

$$P_{1/2} := \{x \in P : \lambda^T Ax \leq \lfloor \lambda^T b \rfloor, \text{ per } \lambda \in \{0, 1/2\}^m \text{ tale che } \lambda^T A \in Z^n\},$$

il poliedro ottenuto dall'intersezione di P con tutti i semispazi indotti da tagli $\{0, \frac{1}{2}\}$. Vi è una differenza importante nella definizione di P_1 e di $P_{1/2}$ a partire da A e b: mentre P_1 dipende unicamente dal poliedro P, e non dal sistema di disuguaglianze $Ax \leq b$ che lo descrive, $P_{1/2}$ è funzione di A e di b; in ogni caso, nè P_1 nè $P_{1/2}$ sono univocamente determinati a partire da P_I; inoltre $P_I \subseteq P_1 \subseteq P_{1/2} \subseteq P$.

Nonostante $P_1 = P$ se e solo se $P = P_I$, talvolta $P_{1/2} = P$ anche se $P \neq P_I$, come ad esempio quando $b/2 \in Z^m$; in ogni caso i tagli $\{0, \frac{1}{2}\}$ hanno un ruolo importante nella teoria poliedrale, in base ai risultati seguenti.

È ben noto che una matrice $\{0, \pm 1\}$ $r \times n$ Q è totalmente unimodulare se e solo se $P = P_I$ per ogni $d \in Z^r$, con $P := \{x \in R^n : \begin{bmatrix} Q \\ -I \end{bmatrix} x \leq \begin{bmatrix} d \\ 0 \end{bmatrix}\}$; analogamente, una matrice $\{0, 1\}$ $r \times n$ Q è bilanciata se e solo se $P = P_I$ per ogni $d \in \{1, +\infty\}^r$ (si veda ad esempio Schrijver, 1986).

Teorema 1.1 *Sia Q una matrice $\{0, \pm 1\}$ $r \times n$, e sia $P := \{x \in R^n : \begin{bmatrix} Q \\ -I \end{bmatrix} x \leq \begin{bmatrix} d \\ 0 \end{bmatrix}\}$. Q è totalmente unimodulare se e solo se $P = P_{1/2}$ per ogni $d \in Z^r$.*

Dim. Se Q è totalmente unimodulare, allora $P = P_{1/2} = P_I$ per ogni $d \in Z^r$. Viceversa, si assuma Q non totalmente unimodulare: per un risultato di Camion (1965), esiste una sua sottomatrice quadrata B tale che la somma degli elementi in ciascuna riga o colonna di B è pari, e tale che la somma complessiva degli elementi di B è divisibile per quattro. Si indichino rispettivamente con I_B e J_B gli insiemi degli indici delle righe e colonne di B in Q, e si definisca $d \in Z^r$ come segue: $d_i := \sum_{j \in J_B} q_{ij}/2$ se $i \in I_B$; $d_i := M$ altrimenti, con $M \geq \max_{i \notin I_B}\{\lceil \sum_{j \in J_B} q_{ij}/2 \rceil\}$. Si costruisce nel seguito un punto $\tilde{x} \in P$ ed un taglio $\{0, \frac{1}{2}\}$ definito da $\tilde{\lambda} \in \{0, 1/2\}^{r+n}$ che separa $\tilde{x}$ da $P_{1/2}$, dimostrando che $P \neq P_{1/2}$. Per $j = 1, \ldots, n$, si ponga $\tilde{x}_j := 1/2$ se $j \in J_B$, $\tilde{x}_j := 0$ altrimenti; per $i = 1, \ldots, r$, si ponga $\tilde{\lambda}_i := 1/2$ se $i \in I_B$, $\tilde{\lambda}_i := 0$ altrimenti; per $j = 1, \ldots, n$ si ponga $\tilde{\lambda}_{r+j} := 1/2$ se $\sum_{i \in I_B} q_{ij}$ è dispari, $\tilde{\lambda}_i := 0$ altrimenti: per costruzione, $\tilde{\lambda}^T \begin{bmatrix} Q \\ -I \end{bmatrix} \in Z^n$, e $\tilde{\lambda}^T \begin{bmatrix} Q \\ -I \end{bmatrix} \tilde{x} = \frac{1}{4} \sum_{i \in I_B} \sum_{j \in J_B} q_{ij} > \lfloor \frac{1}{4} \sum_{i \in I_B} \sum_{j \in J_B} q_{ij} \rfloor = \lfloor \tilde{\lambda}^T d \rfloor$. □

Teorema 1.2 *Sia Q una matrice $\{0,1\}$ $r \times n$, e sia $P := \{x \in R^n : \begin{bmatrix} Q \\ -I \end{bmatrix} x \leq \begin{bmatrix} d \\ 0 \end{bmatrix}\}$. Q è bilanciata se e solo se $P = P_{1/2}$ per ogni $d \in \{1, +\infty\}^r$.*

Dim. Se Q è bilanciata, allora $P = P_{1/2} = P_I$ per ogni $d \in \{1, +\infty\}^r$. Viceversa, si assuma Q non bilanciata: per definizione esiste una sua sottomatrice quadrata B di ordine dispari, tale che la somma degli elementi di ciascuna riga o colonna di B è uguale a due. Utilizzando la stessa costruzione introdotta nella dimostrazione del Teorema 1.1, è possibile definire un vettore $d \in \{1, +\infty\}^r$ tale che $P \neq P_{1/2}$, dimostrando l'enunciato. □

In alcuni casi rilevanti $P_{1/2} = P_1 \neq P_I$, come ad esempio quando P è definito dalle disuguaglianze lato (edge inequalities) e dai vincoli di non negatività del problema dell'insieme indipendente (stable set). Inoltre, in alcuni casi $P_{1/2} = P_1 = P_I$ come ad esempio quando P è l'insieme delle soluzioni del sistema definito dai vincoli di grado e di non negatività del problema del matching, vedi Edmonds (1965) ed Edmonds e Johnson (1970). Anche nel caso in cui $P_1 \neq P_{1/2}$, la famiglia dei tagli $\{0, \frac{1}{2}\}$ spesso contiene numerose classi di disuguaglianze valide per P_I, che talvolta definiscono faccette e che sono utili in un approccio basato su piani di taglio per ottimizzare una funzione obbiettivo lineare su P_I. Questo motiva lo studio del poliedro $P_{1/2}$, occupandosi in particolare del seguente *problema di separazione di un taglio* $\{0, \frac{1}{2}\}$ (chiamato nel seguito $\{0, \frac{1}{2}\}$ *SEP*), nella sua versione di riconoscimento: *Dato $x^* \in P$, determinare $\lambda \in \{0, 1/2\}^m$ tale che $\lambda^T A \in Z^n$ e $\lambda^T A x^* > \lfloor \lambda^T b \rfloor$, o dimostrare che un tale λ non esiste.* Per la ben nota equivalenza tra ottimizzazione e separazione menzionata nel paragrafo introduttivo, la disponibilità di un algoritmo polinomiale per $\{0, \frac{1}{2}\}$ SEP consentirebbe di ottimizzare in tempo polinomiale una funzione obbiettivo lineare su $P_{1/2}$.

Il presente paragrafo è organizzato come segue. Nel sottoparagrafo 1.1.1 vengono descritte la notazione e le definizioni usate nel seguito. Nel sottoparagrafo 1.1.2 si dimostra che $\{0, \frac{1}{2}\}$ SEP è equivalente a trovare un elemento di un clutter binario di peso minimo: quest'ultimo problema è noto essere NP-difficile, e quindi lo è anche $\{0, \frac{1}{2}\}$ SEP; vengono inoltre descritte alcune semplici procedure di riduzione per $\{0, \frac{1}{2}\}$ SEP, e discussi due casi speciali rilevanti risolubili in tempo polinomiale, che si verificano quando A è opportunamente legata alla matrice di incidenza lato-cammino di un albero. Nel sottoparagrafo 1.1.3 si mostra che $\{0, \frac{1}{2}\}$ SEP può essere risolto in tempo polinomiale quando il sistema di disuguaglianze $Ax \leq b$ viene opportunamente rilassato: questo consente di derivare un algoritmo di separazione efficiente per una sottoclasse dei tagli $\{0, \frac{1}{2}\}$ che in molti casi contiene ampie famiglie di disuguaglianze valide per P_I. Nel sottoparagrafo 1.1.4 vengono infine discusse applicazioni ai politopi associati ai problemi del partizionamento in clique (clique partitioning), del commesso viaggiatore asimmetrico (asymmetric traveling salesman), dell'uncapacitated facility location, del sottografo aciclico (acyclic subgraph) e del linear ordering.

1.1.1 Notazione e definizioni di base

Dati $z \in Z$ e $q \in Z_+$, si definisce $z \bmod q := z - \lfloor z/q \rfloor q$; la notazione $a \equiv b \ (\bmod q)$ si intenderà equivalente a $a \bmod q = b \bmod q$.

Data una matrice intera $Q = (q_{ij})$, $\overline{Q} = (\overline{q}_{ij}) := Q \bmod 2$ indica il *supporto binario* di Q, ovverosia $\overline{q}_{ij} = 1$ se q_{ij} è dispari, $\overline{q}_{ij} = 0$ altrimenti.

Dato un grafo non orientato (non necessariamente semplice) $G = (V, E)$ ed un insieme di nodi $S \subseteq V$, siano $\delta(S) := \{ij \in E : i \in S, j \notin S\}$ ed $E(S) := \{ij \in E : i \in S, j \in S\}$; per semplificare la notazione, per $i \in V$ si scriverà $\delta(i)$ al posto di $\delta(\{i\})$.

Un *ciclo* di G è un sottoinsieme C di E tale che $|C \cap \delta(v)|$ è pari per ogni $v \in V$. Sia dato un insieme di lati $T \subseteq E$ che induce una foresta massimale di G: ogni lato in $e \in E \setminus T$ è contenuto in un *ciclo fondamentale*, indicato con C_e, del sottografo indotto da $T \cup \{e\}$; si indica con $M_{cycle}(G, T)$ la matrice $\{0, 1\}$ le cui righe sono i vettori di incidenza dei cicli fondamentali di G relativi a T.

Un *taglio* di G è un sottoinsieme F di E della forma $F = \delta(S)$ per un qualche $S \subseteq V$. Sia $T \subseteq V \times V$ un qualsiasi albero che ricopre V (non si richiede che $T \subseteq E$): per ogni $t \in T$, sia $S_t \subset V$ una qualsiasi delle due componenti del grafo definito dall'insieme di nodi V e dall'insieme di lati $T \setminus \{t\}$. I tagli $\delta(S_t)$, $t \in T$, vengono chiamati *tagli fondamentali* di G (rispetto a T); si indica con $M_{cut}(G, T)$ la matrice $\{0, 1\}$ le cui righe sono i vettori di incidenza dei tagli fondamentali di G rispetto a T. Si noti che l'intersezione di un qualsiasi taglio con un qualsiasi ciclo contiene un numero pari di lati.

Talvolta, a ciascun lato $e \in E$ è assegnata una *label di parità* $f_e \in \{0, 1\}$: in tal caso un insieme di lati $F \subseteq E$ viene chiamato *dispari* se $\sum_{e \in F} f_e \equiv 1 \pmod 2$, *pari* altrimenti.

Una matrice $\{0, 1\}$ $p \times q$ M è la *matrice di incidenza lato-cammino di un albero* (*edge-path incidence matrix of a tree*, indicata più brevemente come *matrice EPT*) se esiste un albero T con $p + 1$ nodi tale che ogni colonna di M è il vettore di incidenza dei lati di un cammino in T; ogni matrice EPT M può essere *rappresentata* tramite un grafo G ed un albero T tali che $M = M_{cut}(G, T)$. Le matrici EPT hanno un ruolo importante nella teoria delle matrici network e possono essere riconosciute in tempo polinomiale, si vedano ad esempio Schrijver (1986) e Nemhauser e Wolsey (1988). Esempi di matrici EPT sono le matrici $\{0, 1\}$ con al più due elementi uguali ad 1 per colonna, e quelle per cui gli elementi uguali ad 1 in ciascuna colonna sono in posizioni consecutive. È noto che una matrice EPT rimane tale permutando, cancellando o duplicando alcune sue righe o colonne, inoltre se M è una matrice EPT (rappresentata da $G = (V, E)$ e T), allora anche $M' := \begin{bmatrix} M \\ e_i^T \end{bmatrix}$ è EPT, avendo indicato con e_i^T l'i-esima riga della matrice identità. Si consideri infatti il lato uv di G associato all'i-esima colonna di M: M' può essere rappresentata da $G' = (V', E')$ e T', con $V' := V \cup \{w\}$, $E' := (E \setminus \{uv\}) \cup \{uw\}$, e $T' := T \cup \{vw\}$.

Data una matrice $\{0, 1\}$ $r \times t$ Q ed un vettore $d \in \{0, 1\}^r$, $d \neq 0$, il *clutter binario* associato a (Q, d) è definito da

$$\mathcal{C}(Q, d) := \{z \in \{0, 1\}^t : Qz \equiv d \pmod 2\}.$$

Dato un clutter binario, si può definire il corrispondente problema del *clutter binario di peso minimo* (*minimum-weight binary clutter problem*, indicato nel seguito come *MW-BCP*):

MW-BCP: *Dato* $w \in R_+^t$, *determinare* $\min\{w^T z : z \in \mathcal{C}(Q, d)\}$.

Esempi ben noti di clutter binari sono dati dall'insieme dei cicli dispari e dall'insieme dei tagli dispari in un grafo con label di parità, ed anche dall'insieme dei complementi dei tagli in un grafo. In particolare, l'insieme dei vettori caratteristici dei cicli dispari in G è il clutter binario $\mathcal{C}(Q, d)$ associato a

$$Q := \begin{bmatrix} f^T \\ M_{cut}(G,T) \end{bmatrix}, d := \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{1.1}$$

con $T \subseteq V \times V$ albero ricoprente V scelto in maniera arbitraria (ad esempio $T := \{1j : j \in V \setminus \{1\}\}$; in questo caso MW-BCP può essere risolto in tempo polinomiale individuando un un ciclo dispari di peso minimo in G, esempi di algoritmi efficienti sono riportati in Grötschel e Pulleyblank (1981) ed in Gerards e Schrijver (1986).

Analogamente, l'insieme dei vettori di incidenza dei tagli dispari di G è il clutter binario $\mathcal{C}(Q, d)$ associato a

$$Q := \begin{bmatrix} f^T \\ M_{cycle}(G,T) \end{bmatrix}, d := \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{1.2}$$

con T foresta massimale qualsiasi di G; il problema MW-BCP può essere risolto efficientemente anche in questo caso, trasferendo l'informazione di parità dai lati ai nodi di G, tramite la definizione per ciascun $v \in V$ della label $\{0, 1\}$ $p_v := (\sum_{e \in \delta(v)} f_e)$ `mod` 2: in questo modo $\sum_{e \in \delta(S)} f_e = \sum_{v \in S} \sum_{e \in \delta(v)} f_e - 2 \sum_{e \in E(S)} f_e$, e quindi $\delta(S)$ è dispari se e solo se S contiene un numero dispari di nodi con label $p_v = 1$. Eseguita questa trasformazione, il problema MW-BCP richiede di individuare un taglio siffatto di peso minimo, ed è risolubile in tempo polinomiale utilizzando l'algoritmo di Padberg e Rao (1982).

Più in generale, il teorema di decomposizione di Seymour (1980) per i matroidi regolari implica che MW-BCP può essere risolto in tempo polinomiale quando $Q = \overline{U}$ per una qualche matrice U totalmente unimodulare.

Infine, l'insieme dei vettori di incidenza dei complementi dei tagli di G è un clutter binario, associato a $Q = M_{cycle}(G, T)$ per una qualsiasi foresta massimale T di G, ed a d definito da $d_i := 1$ se il ciclo fondamentale C_e associato all'i-esima riga di Q ha cardinalità dispari, $d_i := 0$ altrimenti. A differenza degli esempi precedenti, il problema MW-BCP è noto essere NP-difficile in questo caso, dato che richiede di determinare un complemento di un taglio di peso minimo, ovverosia un taglio di peso massimo, e corrisponde quindi al ben noto problema del MAX-CUT. Dato che la versione di riconoscimento di quest'ultimo problema è NP-completa, si veda Garey e Johnson (1979), anche la versione di riconoscimento di MW-BCP lo è.

1.1.2 $\{0, \frac{1}{2}\}$ SEP ed i clutter binari

Un possibile schema di derivazione dei tagli CG è il seguente. Dati $\mu \in Z_+^m$ e $q \in Z_+$ tali che $\mu^T A \equiv 0$ (`mod` q) e $\mu^T b = kq + r$, con $k \in Z$ e $r \in \{1, \ldots, q-1\}$, $\mu^T Ax \leq kq$ è una disuguaglianza valida per P_I, che può equivalentemente essere scritta come $\mu^T(b - Ax) \geq r$: un dato $x^* \in P$ viola $\mu^T Ax \leq kq$ se e solo se $\mu^T(b - Ax^*) < r$. Si osservi che, per ogni q, è sufficiente considerare moltiplicatori $\mu_i \in \{0, \ldots, q-1\}$: un valore μ_i più elevato non cambia il risultato

delle operazioni modulo q ma rende la violazione minore; inoltre, dato il *vettore degli slack* $s^* := b - Ax^*$, la violazione dipende solamente da (A, b) `mod` q. I tagli $\{0, \frac{1}{2}\}$ sono prodotti dalla procedura sopraindicata quando $q = 2$, e quindi $\{0, \frac{1}{2}\}$ SEP (nella sua versione di ottimizzazione) può essere riformulato come segue.

$\{0, \frac{1}{2}\}$ **SEP:** *Dato* $x^* \in P$, *determinare* $\min\{s^{*T}\mu : \mu \in \mathcal{F}(\overline{A}, \overline{b})\}$, *con*

$$s^* := b - Ax^* \geq 0, \ e$$

$$\mathcal{F}(\overline{A}, \overline{b}) := \{\mu \in \{0,1\}^m : \overline{b}^T\mu \equiv 1 \ (\texttt{mod } 2), \overline{A}^T\mu \equiv 0 \ (\texttt{mod } 2)\}.$$

Per costruzione, esiste un taglio $\{0, \frac{1}{2}\}$ violato dal punto x^* se e solo se $\min\{s^{*T}\mu : \mu \in \mathcal{F}(\overline{A}, \overline{b})\} < 1$.

Risulta quindi chiaro lo stretto legame tra $\{0, \frac{1}{2}\}$ SEP ed MW-BCP, che è meglio formalizzato nel seguente teorema.

Teorema 1.3 *I problemi* $\{0, \frac{1}{2}\}$ *SEP ed MW-BCP sono equivalenti.*

Dim. La trasformazione di una qualsiasi istanza di $\{0, \frac{1}{2}\}$ SEP in una istanza equivalente di MW-BCP è molto semplice: è sufficiente definire $w := s^*$, $d := [1|0, \ldots, 0]^T$, e $Q := \begin{bmatrix} \overline{b}^T \\ \overline{A}^T \end{bmatrix}$.

Viceversa, si consideri una qualsiasi istanza di MW-BCP, si definisca $n := r+t+1$, $m := t+1$, $b := [2, \ldots, 2|1]^T$, $A := \left[\begin{array}{c|c} Q^T & \\ d^T & 2I \end{array}\right]$, e si costruisca il punto x^* tale che $Ax^* \leq b$ come segue: $x^*_j := 0$ per $j = 1, \ldots, r$; $x^*_{r+i} := 1 - w_i/2$ per $i = 1, \ldots, t$; e $x^*_{r+t+1} := 1/2$. Per costruzione, $s^* := b - Ax^*$ risulta essere uguale a $[w_1, \ldots, w_t|0]^T$, inoltre, per ogni $\mu \in \{0,1\}^{t+1}$, $\overline{b}^T\mu \equiv 1$ (`mod` 2) se e solo se $\mu_{t+1} = 1$: il problema $\{0, \frac{1}{2}\}$ SEP richiede quindi di determinare $z \in \{0,1\}^t$ tale che $Qz \equiv d$ (`mod` 2) minimizzando w^Tz, ovverosia coincide con MW-BCP. □

Corollario 1.1 $\{0, \frac{1}{2}\}$ *SEP nella sua versione di riconoscimento è NP-completo.*

Riduzioni

La dimensione di un'istanza di $\{0, \frac{1}{2}\}$ SEP può in alcuni casi essere ridotta applicando alcuni criteri di riduzione, ben noti nel contesto dei clutter binari; alcuni di questi criteri sono descritti nel seguito.

(a) Ogni riga i di $Ax \leq b$ tale che $s^*_i \geq 1$ può essere rimossa.

(b) Se $(\overline{A}, \overline{b})$ contiene righe identiche, solo quella con s^*_i minore deve essere considerata.

(c) Si definisca il *grafo di intersezione delle righe* (*row intersection graph*) $G(\overline{A})$ contenente un nodo v_i per ciascuna riga i di $\overline{A}$, ed un lato $[v_i, v_k]$ se e solo se $\overline{A}$ contiene una colonna j tale che $\overline{a}_{ij} = \overline{a}_{kj} = 1$; si indichino inoltre

con $C_1, \ldots, C_t$ le componenti connesse di $G(\overline{A})$. Se $t \geq 2$, $\overline{A}$ può essere trasformata in una matrice diagonale a blocchi, e quindi $\{0, \frac{1}{2}\}$ SEP può essere decomposto in t sottoproblemi indipendenti.

(d) Si supponga che esista una riga h di $\overline{A}$ tale che per un qualche $j \in \{1, \ldots, n\}$, $\overline{a}_{hj} = 1$ ed $\overline{a}_{hk} = 0$ per ogni $k \in \{1, \ldots, n\}$, $k \neq j$; questa situazione si verifica ad esempio quando il sistema $Ax \leq b$ contiene un vincolo di upper o lower bound della forma $\pm x_j \leq b_h$. Si supponga inoltre che il vincolo corrispondente alla riga h sia soddisfatto con uguaglianza dal punto x^*, ovverosia $s^*_h = 0$. Assumendo senza perdita di generalità che $h = m$ e $j = n$, l'input $(\overline{A}, \overline{b}, s^*)$ di $\{0, \frac{1}{2}\}$ SEP ha la forma:

$$\overline{A} = \left[\begin{array}{c|c} M & d \\ 0 \cdots 0 & 1 \end{array}\right], \overline{b} = \begin{bmatrix} \beta \\ \overline{b}_m \end{bmatrix}, \text{ e } s^* = \begin{bmatrix} \sigma^* \\ 0 \end{bmatrix}.$$

Ogni soluzione ammissibile $\mu \in \{0,1\}^m$ di $\{0, \frac{1}{2}\}$ SEP soddisfa in questo caso $\mu_m \equiv \sum_{i=1}^{m-1} \mu_i d_i \pmod 2$: si può allora definire un'istanza ridotta di $\{0, \frac{1}{2}\}$ SEP il cui input è dato da (M, f, σ^*), con $f := \beta$ se $\overline{b}_m = 0$, $f := (\beta + d) \bmod 2$ altrimenti. Esiste allora una corrispondenza biunivoca tra le soluzioni ammissibili $\mu \in \{0,1\}^m$ per l'istanza originale e $\nu \in \{0,1\}^{m-1}$ per l'istanza ridotta, con $\mu_k = \nu_k$ per $k = 1, \ldots, m-1$, e $\mu_m \equiv d^T \nu \pmod 2$.

Casi speciali di $\{0, \frac{1}{2}\}$ SEP risolubili in tempo polinomiale

Il primo caso speciale risolubile in tempo polinomiale si ha quando $\overline{A}^T$ è una matrice EPT.

Teorema 1.4 *Il problema $\{0, \frac{1}{2}\}$ SEP può essere risolto in tempo polinomiale se $\overline{A}^T$ è una matrice EPT.*

Dim. Sia $\overline{A}^T$ rappresentata da $G = (V, E)$ e da T, con $|V| = n + 1$, $|E| = m$, ed $\overline{A}^T = M_{cut}(G, T)$: per costruzione $\mathcal{F}(\overline{A}, \overline{b}) = \mathcal{C}(Q, d)$, definendo Q e d come in (1.1), con $f := \overline{b}$. $\{0, \frac{1}{2}\}$ SEP può essere risolto in tempo polinomiale in quanto richiede di determinare un ciclo dispari di peso minimo in G, dove s^*_i e $\overline{b}_i$ rappresentano rispettivamente il peso e la label di parità del lato associato all'i-esima riga di $\overline{A}$. □

Gerards e Schrijver (1986) descrivono un algoritmo efficiente per $\{0, \frac{1}{2}\}$ SEP nel caso in cui A sia una matrice intera tale che $\sum_j |a_{ij}| \leq 2$ per ogni riga i. Più in generale, il teorema 1.4 implica che $\{0, \frac{1}{2}\}$ SEP è risolubile in tempo polinomiale quando ciascuna riga di A contiene al più due elementi dispari; in questo caso infatti $\overline{A}^T$ è la matrice EPT associata al grafo $G = (V, E)$ ed alla stella T, dove $V := \{1, \ldots, n+1\}$, $T := \{[n+1, j] : j = 1, \ldots, n\}$, ed E contiene un lato jk per ogni riga i di $\overline{A}$ con $\overline{a}_{ij} = \overline{a}_{ik} = 1$, ed un lato $[n+1, j]$ per ogni riga i contenente un unico elemento non nullo $\overline{a}_{ij}$.

Si considera ora il caso in cui $\overline{A} = \left[\begin{smallmatrix} M \\ I \end{smallmatrix}\right]$, che si verifica ad esempio quando $x \geq 0$ fa parte del sistema $Ax \leq b$.

Teorema 1.5 *$\{0, \frac{1}{2}\}$ SEP può essere risolto in tempo polinomiale se $\overline{A} = \left[\begin{smallmatrix} M \\ I \end{smallmatrix}\right]$ ed M è una matrice EPT.*

Dim. Sia M rappresentata da $G=(V,E)$ e da T, ovverosia $M=M_{cut}(G,T)$: ciascuna riga di M è associata ad un lato di T, mentre le altre righe di $\overline{A}$ possono essere associate ai lati in E. In tal modo, le colonne di $\overline{A}$ rappresentano i vettori di incidenza dei cicli fondamentali (rispetto a T) del grafo $\tilde{G}:=(V,E\cup T)$, e quindi $\overline{A}^T=M_{cycle}(\tilde{G},T)$; ne segue che $\mathcal{F}(\overline{A},\overline{b})=\mathcal{C}(Q,d)$, con Q e d definite come in (1.2), essendo $f:=\overline{b}$. $\{0,\frac{1}{2}\}$ SEP può essere quindi risolto in tempo polinomiale, in quanto richiede di determinare un taglio dispari di peso minimo in $\tilde{G}$, dove s_i^* e $\overline{b}_i$ rappresentano rispettivamente il peso e la label di parità del il lato di $\tilde{G}$ associato all'i-esima riga di $\overline{A}$. □

Padberg e Rao (1982) descrivono un algoritmo efficiente per $\{0,\frac{1}{2}\}$ SEP nel caso in cui $P:=\{x\in R^n : Dx\le d, 0\le x\le g\}$, e D è la matrice di incidenza nodo-lato di un grafo, cioè quando P_I è il politopo del capacitated b-matching. Più in generale, il teorema 1.5 implica che $\{0,\frac{1}{2}\}$ SEP può essere risolto in tempo polinomiale quando $P:=\{x\in R^n : d^1\le Dx\le d^2, g^1\le x\le g^2\}$, e $\overline{D}$ è una matrice EPT; in tal caso infatti $\overline{A}=\left[\begin{smallmatrix}M\\I\end{smallmatrix}\right]$, e $M=\left[\begin{smallmatrix}\overline{D}\\\overline{D}\\I\end{smallmatrix}\right]$ è una matrice EPT, essendo ottenuta da $\overline{D}$ duplicando righe ed aggiungendo righe della matrice identità.

1.1.3 Ottimizzazione su di un rilassamento di $P_{1/2}$

A seguito del corollario 1.1, è improbabile che esista un algoritmo polinomiale per ottimizzare una funzione obbiettivo lineare su $P_{1/2}$. Sia $P':=\{x\in R^n : A'x\le b'\}\supseteq P$ un rilassamento di P nel quale il sistema $A'x\le b'$ è ottenuto "indebolendo" $Ax\le b$, in modo tale da rendere il problema $\{0,\frac{1}{2}\}$ SEP associato ad (A',b') risolubile in tempo polinomiale nella dimensione di (A,b): è allora chiaramente possibile ottimizzare in tempo polinomiale sul poliedro $P\cap P'_{1/2}$.

Ci sono molti possibili rilassamenti che soddisfano la proprietà sopra menzionata; tra tutti viene qui studiato quello ottenuto facendo sistematicamente uso di lower ed upper bound sulle variabili, in modo da produrre un sistema indebolito $A'x\le b'$ nel quale ogni riga di A' contenga al massimo due elementi dispari. Più specificamente, si assuma che i vincoli di bound $0\le x\le d$ siano parte del sistema $Ax\le b$, ammettendo il caso di $d_j=+\infty$ per alcune variabili j, e per ogni riga i si definisca $O_i:=\{j : a_{ij}\text{ dispari}\}$.

L-indebolimento

L'indebolimento più semplice si ha quando i vincoli di lower bound $-x_j\le 0$, $j=1,\dots,n$, sono sommati sistematicamente alle disuguaglianze in $Ax\le b$ in modo da ridurre a due (al massimo) il numero di elementi dispari in ogni riga, sostituendo ciascuna disuguaglianza $\sum_j a_{ij}x_j\le b_i$ tale che $|O_i|\ge 3$, con i suoi $\binom{|O_i|}{2}$ *L-indebolimenti*

$$a_{ih}x_h+a_{ik}x_k+\sum_{j\notin O_i}a_{ij}x_j+\sum_{j\in O_i\setminus\{h,k\}}(a_{ij}-1)x_j\le b_i$$

per ogni coppia $h,k\in O_i$, $h<k$: in questo modo il sistema indebolito $A'x\le b'$ contiene $O(mn^2)$ vincoli, ma solo $O(n^2)$ di essi devono essere considerati espli-

citamente per un dato x^*, grazie alla riduzione (b) descritta nel sottoparagrafo 1.1.2.

U-indebolimento

In maniera analoga, utilizzando i vincoli di upper bound $x_j \leq d_j$, è possibile indebolire $Ax \leq b$ sostituendo ciascuna disuguaglianza $\sum_j a_{ij}x_j \leq b_i$ tale che $|O_i| \geq 3$, con i suoi $\binom{|O_i|}{2}$ *U-indebolimenti*

$$a_{ih}x_h + a_{ik}x_k + \sum_{j \notin O_i} a_{ij}x_j + \sum_{j \in O_i \setminus \{h,k\}} (a_{ij}+1)x_j \leq b_i + \sum_{j \in O_i \setminus \{h,k\}} d_j$$

per tutte le coppie $h, k \in O_i$, $h < k$.

LU-indebolimento

Più in generale, è possibile usare entrambi i vincoli di upper e lower bound sulle variabili per derivare $A'x \leq b'$, sostituendo ciascuna disuguaglianza $\sum_j a_{ij}x_j \leq b_i$ tale che $|O_i| \geq 3$ con i suoi *LU-indebolimenti*

$$a_{ih}x_h + a_{ik}x_k + \sum_{j \notin O_i} a_{ij}x_j + \sum_{j \in L}(a_{ij}-1)x_j + \sum_{j \in U}(a_{ij}+1)x_j \leq b_i + \sum_{j \in U} d_j$$

per ogni $h, k \in O_i$, $h < k$, e per ogni partizione (L, U) di $O_i \setminus \{h, k\}$.

Sebbene $A'x \leq b'$ contenga in generale un numero esponenziale di righe, il corrispondente problema $\{0, \frac{1}{2}\}$ SEP può essere risolto in tempo polinomiale dato che, fissato il punto x^*, per ogni terna (i, h, k) è in realtà necessario considerare solamente i due LU-indebolimenti con right-hand-side pari e dispari aventi slack minimo, determinabili in tempo $O(n)$ tramite un semplice schema di programmazione dinamica che considera, per ciascun $j \in O_i \setminus \{h, k\}$, le due possibilità $j \in L$ e $j \in U$.

Teorema 1.6 *È possibile ottimizzare in tempo polinomiale sul rilassamento di $P_{1/2}$ definito da $P \cap P'_{1/2}$, dove $P' := \{x \in R^n : A'x \leq b'\}$ ed $A'x \leq b'$ è ottenuto a partire da $Ax \leq b$ utilizzando l'LU-indebolimento.*

1.1.4 Applicazioni

Sia $\mathcal{H}$ la famiglia dei tagli $\{0, \frac{1}{2}\}$ derivabili dal sistema indebolito $A'x \leq b'$ ottenuto da $Ax \leq b$ per mezzo dell'LU-indebolimento; per numerosi poliedri studiati in letteratura $\mathcal{H}$ contiene ampie famiglie di disuguaglianze valide, alcune delle quali definiscono faccette del poliedro P_I: $P \cap P'_{1/2}$ può quindi essere in alcuni casi una buona approssimazione di P_I. Nel seguito vengono discussi alcuni casi rilevanti.

Il politopo del partizionamento in clique

Il problema del *partizionamento in clique* (*clique partitioning*) deriva da applicazioni di tipo clustering: dato un grafo non orientato completo $G=(V,E)$, un insieme di lati è detto partizionamento in clique di G se V può essere partizionato in sottoinsiemi disgiunti $W_1,\dots,W_k$ tali che $A=\bigcup_{i=1}^{k} E(W_i)$. Sia

$$P_I := \text{conv}\{x\in\{0,1\}^E : x_{ij}+x_{jk}-x_{ik}\le 1 \;\; i,j,k\in V, |\{i,j,k\}|=3\}$$

il politopo del partizionamento in clique; i vincoli $x_{ij}+x_{jk}-x_{ik}\le 1$ sono chiamati *disuguaglianze triangolo* (*triangle inequalities*). Molte classi di disuguaglianze che definiscono faccette di P_I sono state studiate da Grötschel e Wakabayashi (1990), ed includono le seguenti *disuguaglianze ciclo dispari con 2-corde* (*2-chorded odd cycle inequalities* : *2COC*): sia $C=\{e_1,\dots,e_k\}$, $k\ge 5$ dispari, un ciclo di G, con $e_i=v_iv_{i+1}$ $(i=1,\dots,k-1)$ e $e_k=v_kv_1$ (per semplicità, $v_{k+1}:=v_1$ e $v_{k+2}:=v_2$). L'insieme $\overline{C}:=\{v_iv_{i+2} : i=1,\dots,k\}$ contiene le cosiddette *2-corde* di C. La disuguaglianza 2COC associata a C è definita da

$$\sum_{ij\in C} x_{ij} - \sum_{ij\in\overline{C}} x_{ij} \le \frac{k-1}{2}.$$

Nessun algoritmo di separazione per queste disuguaglianze è stato proposto in letteratura prima di Caprara e Fischetti (1993). Müller (1993) propone un algoritmo di separazione basato sul calcolo di cicli dispari per una classe di disuguaglianze simile, associata al politopo del *sottografo diretto aciclico transitivo* (*transitive acyclic subdigraph*).

Le disuguaglianze 2COC sono tagli $\{0,\frac{1}{2}\}$ ottenuti combinando le seguenti disuguaglianze:

$$x_{v_iv_{i+1}}+x_{v_{i+1}v_{i+2}}-2x_{v_iv_{i+2}}\le 1 \;\; \text{per } i=1,\dots,k,$$

ciascuna delle quali è un L-indebolimento di una disuguaglianza triangolo. Questi non sono i soli tagli $\{0,\frac{1}{2}\}$ ottenibili dalle disuguaglianze triangolo indebolite della forma $x_{ij}+x_{jk}-2x_{ik}\le 1$; sia ad esempio $C=\{e_1,\dots,e_k\}$, $k\ge 3$ dispari, un ciclo di G con $e_i=v_iv_{i+1}$ per $i=1,\dots,k$; dato $z\in V\setminus\{v_1,\dots,v_k\}$, è possibile sommare i vincoli

$$x_{v_iz}+x_{zv_{i+1}}-2x_{v_iv_{i+1}}\le 1 \;\; \text{per } i=1,\dots,k,$$

pesati ad $1/2$, ed ottenere arrotondando la *disuguaglianza ruota dispari* (*odd wheel inequality*)

$$\sum_{i=1}^{k} x_{v_iz} - \sum_{ij\in C} x_{ij} \le \frac{k-1}{2}.$$

Queste disuguaglianze definiscono faccette di P_I (Chopra e Rao, 1993), e possono essere separate in tempo polinomiale (Deza, Grötschel e Laurent, 1992).

Dato che le disuguaglianze triangolo indebolite appartengono alla famiglia $\mathcal{H}$, è possibile ottimizzare in tempo polinomiale su di un rilassamento di $P_{1/2}$ definito da un insieme di disuguaglianze che contiene tutte le disuguaglianze 2COC e ruota dispari.

Il politopo del commesso viaggiatore asimmetrico

Dato un grafo diretto completo privo di autoanelli $G = (V, A)$, il politopo P_I del *commesso viaggiatore asimmetrico* (*asymmetric traveling salesman*) è definito dalla chiusura convessa dei vettori di incidenza dei circuiti Hamiltoniani (*tour*) di G, ovverosia

$$P_I := \text{conv}\{x \in \{0,1\}^A :$$

$$\sum_{j \in V} x_{ij} = 1, \quad i \in V \tag{1.3}$$

$$\sum_{i \in V} x_{ij} = 1, \quad j \in V \tag{1.4}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \quad S \subset V, |S| \geq 2 \ \}. \tag{1.5}$$

Le disuguaglianze (1.5) sono dette *vincoli di eliminazione di subtour* (*subtour elimination constraints*); nonostante il loro numero sia esponenziale, queste disuguaglianze possono essere separate efficientemente tramite algoritmi di flusso massimo, ed è quindi possibile ottimizzare in tempo polinomiale su $P := \{x \in R_+^A : x \text{ soddisfa (1.3)-(1.5)}\}$.

Due archi (i, j) ed (h, k) sono detti *incompatibili* se $i = h$, o $j = k$, o $(i, j) = (k, h)$; si può osservare che gli L-indebolimenti di (1.3)-(1.5) sono rappresentati dalle disuguaglianze $x_{ij} + x_{hk} \leq 1$ per tutte le coppie incompatibili di archi (i, j) ed (h, k). La famiglia $\mathcal{H}$ contiene quindi le seguenti disuguaglianze, introdotte da Balas (1989): un *trail chiuso alternante* (*closed alternating trail : CAT*) è una sequenza di archi $T := \{a_1, \ldots, a_s\}$ tale che ciascun a_i è incompatibile con a_{i-1} ed a_{i+1}, e compatibile con tutti gli altri archi di T (per semplicità, $a_0 := a_s$ ed $a_{s+1} := a_1$). Il CAT è detto *dispari* se tale è la cardinalità s di T. Sommando con peso $1/2$ i vincoli $x_{a_i} + x_{a_{i+1}} \leq 1$ per $i = 1, \ldots, s$, ed arrotondando, si ottiene la seguente *disuguaglianza debole CAT dispari* (*weak odd CAT inequality*):

$$\sum_{(i,j) \in T} x_{ij} \leq \frac{|T| - 1}{2}.$$

L'esperienza computazionale riportata in Fischetti e Toth (1994) mostra che queste disuguaglianze sono utili per velocizzare la convergenza di un algoritmo branch-and-cut per la risoluzione di istanze reali difficili del problema del commesso viaggiatore asimmetrico.

Le disuguaglianze deboli di tipo CAT dispari possono essere liftate ottenendo disuguaglianze che definiscono faccette di P_I, con alcune eccezioni "patologiche" che si hanno per valori piccoli di $|V|$. I vincoli così ottenuti sono i tagli $\{0, \frac{1}{2}\}$ derivati sostituendo nella derivazione di Chvátal-Gomory ciascuna disuguaglianza $x_{ij} + x_{hk} \leq 1$ con $i = h$ o $j = k$, con la corrispondente equazione (1.3) o (1.4). Queste disuguaglianze generalizzano le disuguaglianze comb; come per queste ultime, la complessità del problema della separazione delle disuguaglianze CAT dispari liftate non è nota.

Il politopo dell'uncapacitated facility location

Il problema dell'*uncapacitated* (o *simple*) *facility location* ha numerose applicazioni in localizzazione ed è stato ampiamente studiato; si rimanda ad esempio a Cornuéjols, Nemhauser e Wolsey (1990). Dato un grafo bipartito completo $G = (V_1 \cup V_2, E)$, una soluzione ammissibile del problema è data da un sottoinsieme E' di E tale che $|E' \cap \delta(i)| = 1$ per ogni $i \in V_1$.

Il politpo dell'uncapacitated facility location è definito da

$$P_I := \text{conv}\{\ (x,y) \in \{0,1\}^{E \cup V_2} :$$

$$x_{ij} - y_j \leq 0, \quad i \in V_1, j \in V_2 \tag{1.6}$$

$$\sum_{ij \in \delta(i)} x_{ij} = 1, \quad i \in V_1 \quad \}, \tag{1.7}$$

dove $x_{ij} = 1$ se e solo se il lato ij è scelto in E', e $y_j = 1$ se e solo se $|E' \cap \delta(j)| \neq 0$.

I tagli $\{0, \frac{1}{2}\}$ contengono le seguenti *disuguaglianze ciclo dispari* (*odd cycle inequalities*)

$$\sum_{ij \in C} x_{ij} - \sum_{j \in V_2(C)} y_j \leq \frac{k-1}{2},$$

in cui C è un ciclo di G di lunghezza $2k$, con $k \geq 3$ dispari, e $V_i(C)$ contiene i k nodi di V_i visitati da C $(i = 1, 2)$; chiaramente $|V_1(C)| = |V_2(C)| = k$ poichè G è bipartito. Queste disuguaglianze sono ottenute sommando con peso $1/2$ i vincoli

$$x_{ij} - y_j \leq 0, \text{ per } \ ij \in C$$

$$\sum_{ij \in \delta(i) \cap C} x_{ij} \leq 1, \quad \text{per } i \in V_1(C), \tag{1.8}$$

ed arrotondando. Dato che (1.8) è un L-indebolimento di (1.7), le disuguaglianze ciclo dispari appartengono alla famiglia $\mathcal{H}$.

I politopi del sottografo aciclico e del linear ordering

Dato un grafo diretto completo privo di autoanelli $G = (V, A)$, il politopo P_{AC} del *sottografo aciclico* (*acyclic subgraph*) è definito da

$$P_{AC} := \text{conv}\{\ x \in \{0,1\}^A :$$

$$\sum_{(i,j) \in C} x_{ij} \leq |C| - 1 \ \text{ per tutti i cicli } C \subseteq A \quad \}, \tag{1.9}$$

ed è stato studiato da Grötschel, Jünger e Reinelt (1984, 1985a,b). Siano $C_1, \ldots, C_k$ cicli distinti di G. Per ogni $(i,j) \in A$, sia

$$\mu_{ij} := |\{h : (i,j) \in C_h\}|,$$

e

$$M := \bigcup_{h=1}^{k} C_h,$$

$$M^* := \{(i,j) \in M : \mu_{ij} \text{ dispari}\};$$

sia inoltre (M_1^*, M_2^*) una partizione di M^* (eventualmente $M_1^* = \emptyset$ o $M_2^* = \emptyset$) e si assuma $\sum_{h=1}^{k} |C_h| + |M_1^*| - k$ dispari. Sommando con pesi 1/2 i vincoli

$$\sum_{(i,j)\in C_h} x_{ij} \leq |C_h| - 1 \quad \text{per } h = 1, \ldots, k$$

$$x_{ij} \leq 1 \quad \text{per } (i,j) \in M_1^*$$

$$-x_{ij} \leq 0 \quad \text{per } (i,j) \in M_2^*$$

ed arrotondando, si ottiene il taglio

$$\sum_{(i,j)\in M\setminus M^*} \frac{\mu_{ij}}{2} x_{ij} + \sum_{(i,j)\in M_1^*} \frac{\mu_{ij}+1}{2} x_{ij} + \sum_{(i,j)\in M_2^*} \frac{\mu_{ij}-1}{2} x_{ij} \leq$$

$$\leq \frac{\sum_{h=1}^{k} |C_h| + |M_1^*| - k - 1}{2}; \qquad (1.10)$$

questa classe di disuguaglianze è stata introdotta per la prima volta da Caprara e Fischetti (1993).

Si noti che i coefficienti di alcune variabili in (1.10) possono essere maggiori di 1; se però si impone la restrizione ulteriore che

$$\mu_{ij} \leq 2, \quad (i,j) \in M,$$

cioè ciascun arco sia comune al massimo a due cicli, scegliendo $M_1^* = M^*$ ed $M_2^* = \emptyset$ la disuguaglianza (1.10) diviene

$$\sum_{(i,j)\in M} x_{ij} \leq |M| - \frac{k+1}{2}, \qquad (1.11)$$

con k dispari, dato che $\sum_{h=1}^{k} |C_h| + |M^*| - k = 2|M| - k$ deve essere dispari. Nel caso in cui i cicli $C_1, \ldots, C_k$ soddisfino alcune ulteriori condizioni "tecniche", si vedano le condizioni (2.15)-(2.17) in Grötschel, Jünger e Reinelt (1985a), il vincolo (1.11) è una cosiddetta *disuguaglianza Möbius ladder* – la famiglia di queste disuguaglianze contiene anche membri non aventi la forma (1.10), nel caso in cui $\mu_{ij} \geq 3$ per qualche $(i,j) \in M$. Riguardo al problema della separazione di (1.11), si osserva che queste disuguaglianze possono essere derivate dal seguente indebolimento di (1.9):

$$\sum_{(i,j)\in C_h} x_{ij} + \sum_{(i,j)\in C_h\cap M^*} x_{ij} \leq |C_h| + |C_h \cap M^*| - 1 \quad \text{per } h = 1, \ldots, k. \quad (1.12)$$

Nel caso speciale in cui $|C_h \setminus M^*| \leq 2$ per ogni h, queste ultime disuguaglianze hanno ciascuna al più due coefficienti dispari nel left-hand-side; il problema $\{0, \frac{1}{2}\}$ SEP associato al sistema (1.12) può essere risolto efficientemente se in

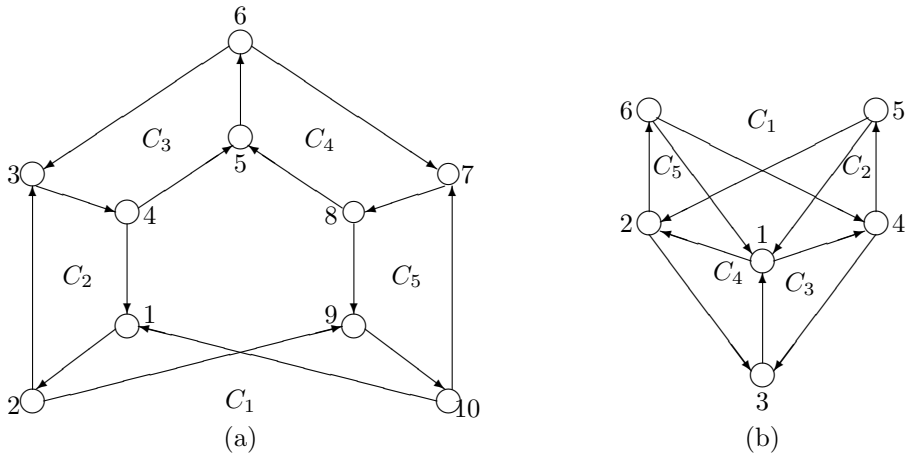


Figura 1.1: Due Möbius ladders.

maniera euristica ci si limita a considerare un numero polinomiale di disuguaglianze (1.12), come ad esempio quelle derivate dai vincoli (1.9) con $|C_h| \leq t$ per un dato t.

Il politopo del *linear ordering* è definito da

$$P_{LO} := \text{conv}\{\ x \in \{0,1\}^A : \quad (1.9) \ \text{ e}$$

$$x_{ij} + x_{ji} = 1, \quad 1 \leq i < j \leq |V| \quad \}. \tag{1.13}$$

È noto che nella definizione di P_{LO} le disuguaglianze (1.9) possono essere sostituite dalle *disuguaglianze triangolo* (*triangle inequalities*)

$$x_{ij} + x_{jk} + x_{ki} \leq 2, \quad i,j,k \in V, i < j, i < k, j \neq k, \tag{1.14}$$

dato che ogni disuguaglianza (1.9) associata ad un ciclo $C = \{(i_1,i_2),(i_2,i_3), \ldots,(i_{|C|},i_1)\}$ con $|C| \geq 4$ può essere ottenuta sommando $\sum_{(i,j)\in C'} x_{ij} \leq |C'| - 1$, $x_{i_1 i_2} + x_{i_2 i_3} + x_{i_3 i_1} \leq 2$, e $-x_{i_1 i_3} - x_{i_3 i_1} = -1$, dove $C' := \{(i_1,i_3),(i_3,i_4),\ldots, (i_{|C|},i_1)\}$ (e quindi $|C'| = |C| - 1$).

Si osservi che le disuguaglianze (1.13) hanno ciascuna due coefficienti dispari nel left-hand-side, mentre le (1.14) ammettono l'U-indebolimento:

$$x_{ij} + 2x_{jk} + x_{ki} \leq 3, \quad i,j,k \in V, |\{i,j,k\}| = 3. \tag{1.15}$$

È quindi possibile separare in tempo polinomiale sulla famiglia dei tagli $\{0, \frac{1}{2}\}$ ottenuti combinando (1.13) e (1.15); questa famiglia contiene tra le altre le disuguaglianze Möbius ladder descritte nel teorema 3.11 in Grötschel, Jünger e Reinelt (1985b). Per esempio, la disuguaglianza Möbius ladder definita dal grafo della figura 1.1 (a) è ottenuta combinando

$$x_{12} + 2x_{23} + x_{31} \leq 3, \quad x_{34} + 2x_{41} + x_{13} \leq 3, \quad -x_{13} - x_{31} = -1$$

$$x_{34} + 2x_{45} + x_{53} \leq 3, \quad x_{56} + 2x_{63} + x_{35} \leq 3, \quad -x_{35} - x_{53} = -1$$

$$x_{56} + 2x_{67} + x_{75} \leq 3, \quad x_{78} + 2x_{85} + x_{57} \leq 3, \quad -x_{57} - x_{75} = -1$$
$$x_{78} + 2x_{89} + x_{97} \leq 3, \quad x_{9,10} + 2x_{10,7} + x_{79} \leq 3, \quad -x_{79} - x_{97} = -1$$
$$x_{9,10} + 2x_{10,1} + x_{19} \leq 3, \quad x_{12} + 2x_{29} + x_{91} \leq 3, \quad -x_{19} - x_{91} = -1,$$

mentre quella associata al grafo della figura 1.1 (b) è derivata da

$$x_{12} + 2x_{23} + x_{31} \leq 3, \quad x_{14} + 2x_{43} + x_{31} \leq 3$$
$$x_{14} + x_{45} + 2x_{51} \leq 3, \quad x_{12} + x_{26} + 2x_{61} \leq 3$$
$$x_{26} + x_{65} + 2x_{52} \leq 3, \quad x_{45} + x_{56} + 2x_{64} \leq 3, \quad -x_{65} - x_{56} = -1.$$

1.2 Situazione di partenza

Nel 1996 il Professor Caprara ha implementato in linguaggio C un separatore per tagli $0-1/2$, e ha provato ad usarlo in un Branch&Cut basato sulle librerie MINTO. Questa implementazione è molto semplice da riutilizzare: l'interfaccia è costituita da un'unica funzione che ha come parametri la formulazione lp e il punto frazionario da separare e restituisce i tagli che separano il punto. Tutte le espressioni lineari sono memorizzate come coppie di vettori indici-valori.

I primi risultati sono stati poco incoraggianti: se da una parte l'applicazione dei tagli faceva diminuire il numero di nodi necessari a trovare la soluzione intera ottima, dall'altra faceva aumentare il tempo per nodo a un punto tale che il tempo totale era più elevato di quello necessario senza tagli.

Il test-bed usato dal Prof. Caprara è lo stesso usato per questo lavoro di tesi.

Capitolo 2

Ilog CPLEX

Ilog CPLEX8.1 è un software per la programmazione lineare mista molto usato sia in ambito industriale che di ricerca.

CPLEX deve la sua fama sia alle sue elevate prestazioni che alla stabilità della sua implementazione dell'algoritmo del simplesso. Infatti, fino a cinque anni fa, l'ostacolo principale alla soluzione di istanze difficili era l'instabilità numerica; questa era dovuta sia alle dimensioni delle matrici che ai rapporti fra i coefficienti stessi (si pensi all'uso tipico del "big M" per la modellazione di alcune classi di problemi). Capitava spesso, soprattutto in ambito di ricerca, che un branch & bound non potesse essere portato a termine per l'insolubilità di alcuni nodi.

L'uso di CPLEX è quindi inevitabile se non altro come termine di paragone per la misura delle prestazioni di un nuovo software.

In realtà però, per questo lavoro di tesi, CPLEX è stato usato anche per implementare l'algoritmo di Branch&Cut; infatti, dalla versione 7, CPLEX offre la possibilità di essere esteso con dei moduli esterni chiamati *callbacks*: questi consistono di parti di codice scritte in C, C++ o Java che implementano funzionalità come la scelta della variabile di branching, la separazione di una variabile frazionaria etc. che vengono eseguite al momento opportuno all'interno della normale procedura di branch & cut.

Nel nostro caso, quindi, è stato possibile studiare l'efficacia dei tagli 0-1/2 all'interno di un già collaudato e apprezzato risolutore, rendendo molto più significative le misure effettuate: se fosse stato necessario implementare un solver MIP completo, infatti, le misure sarebbero state influenzate dalla qualità di tutta l'implementazione, e non solo dall'efficacia dei tagli.

La possibilità di implementare un separatore all'interno di un solver completo e veloce, inoltre, consente di stabilire se l'aumento di prestazioni fornito rispetto allo stato dell'arte della tecnologia giustifichi uno sviluppo ulteriore del separatore e la sua implementazione all'interno di software commerciali.

2.1 Caratteristiche principali

In questa sezione si analizzano brevemente le caratteristiche delle parti principali di CPLEX8, alla luce di quello che prevedono le tecniche più diffuse di soluzione di problemi MIP.

Per una descrizione dettagliata delle funzionalità si faccia riferimento alla documentazione del software; per avere invece un'idea dell'impatto sulle prestazioni delle singole componenti si vedano [4] e la sezione 4.3.1.

2.1.1 Simplesso, preprocessor e prober

Si dispone di tre algoritmi per risolvere il rilassamento continuo: *primale*, *duale* e *barrier*.

Il preprocessor e il prober sono applicati prima della soluzione del rilassamento, per cercare di migliorare la formulazione; il primo cerca di togliere variabili e vincoli inutili, mentre il secondo analizza le implicazioni logiche dell'assegnamento di valori alle variabili. Il modello su cui CPLEX applica il solver ha quindi, in genere, meno vincoli, meno variabili e un po' di coefficienti modificati.

Parte di queste due funzionalità vengono disattivate automaticamente se si usa una callback per inserire tagli durante la soluzione di una istanza MIP. In particolare si tratta delle trasformazioni irreversibili; dall'interno di una callback, infatti, si ha disposizione sempre e solo il modello originale; la traduzione viene effettuata in modo automatico dalle API, e naturalmente deve essere sempre possibile.

2.1.2 Euristica

Questa è una componente molto utile; come ogni solver MIP, a intervalli casuali CPLEX tenta di individuare soluzioni ammissibili in modo euristico; durante una ottimizzazione la maggrior parte delle soluzioni ammissibili sono trovate per mezzo dell'euristica, anzichè con la normale esplorazione dell'albero.

2.1.3 Famiglie di tagli

CPLEX fornisce 9 famiglie di tagli: Clique, Cover, Disjunctive, Flow Cover, Flow Paths, Gomory Fractional, Generalized Upper Bound Cover, Implied Bound, Mixed Integer Rounding.

Per un lavoro di ricerca su una nuova famiglia di tagli, il numero di famiglie a disposizione garantisce il fatto di poter capire se la nuova famiglia sia veramente utile oltre a essere efficace in se e per se.

2.1.4 Strategie di Branch & Cut

Il Branch&Cut di CPLEX è piuttosto diverso dall'algoritmo che si trova nei libri di testo (per esempio [1]). Si dà qui una descrizione molto superficiale delle caratteristiche tipiche di un Branch & Cut, sufficiente però a far capire in cosa si distingue l'implementazione di CPLEX8.

Gli algoritmi della famiglia Branch&Cut, in genere, gestiscono il modello in modo dinamico: i tagli generati dal separatore, di validità globale, sono mantenuti in una struttura dati chiamata *pool*; a ogni nodo viene fatta una scansione del pool: se un taglio del pool è violato dalla soluzione corrente del rilassamento continuo, il taglio viene aggiunto alla formulazione e di conseguenza la soluzione viene aggiornata, tipicamente sfruttando la formulazione duale del simplesso. Se il pool non contiene tagli violati si prova ad applicare il separatore alla soluzione corrente, arricchendo così il pool di nuovi tagli. Ad intervalli regolari si effettua un *purging* del pool per cancellare i tagli inutili perchè dominati da altri tagli più profondi; allo stesso modo anche fra i vincoli della formulazione originale possono essere disattivati quelli che non partecipano alla definizione della regione di politopo che si sta esplorando.

I vantaggi di questo tipo di implementazione sono due:

1. l'applicazione diretta del separatore è tipicamente piuttosto costosa; mantenere i tagli nel pool evita di dover generare più volte lo stesso taglio;

2. i tagli vengono aggiunti alla formulazione in molto locale; in questo modo la formulazione di ciascun nodo non è appesantita da tagli che non partecipano alla soluzione corrente.

CPLEX funziona in un modo molto più semplice; la sua taratura di default si basa sull'algoritmo 1. Per motivi di chiarezza non è stata esplicitata la gestione delle soluzioni intere trovate, del valore di best-bound, di cut-off etc. L'algoritmo qui descritto non è riportato esplicitamente nella documentazione, ma è stato ricavato osservando l'output delle ottimizzazioni.

Evidentemente, alla base di questo algoritmo ci sono le seguenti assunzioni:

1. se un taglio è utile, deve essere inserito nella formulazione in modo statico;

2. troppi tagli possono essere controproducenti: appesantiscono la formulazione e possono allungare a dismisura il tempo necessario a visitare l'albero, anche se questo è stato "potato" dalla presenza dei tagli stessi;

3. grazie all'efficienza del suo simplesso duale, CPLEX è molto veloce nel visitare nodi in modalità depth-first;

4. CPLEX8 ha un'euristica molto efficace, che di solito riesce a trovare soluzioni intere molto prima di quanto si potrebbe con l'enumerazione; in questo modo il valore di cut-off sale velocemente e l'albero viene ridotto di conseguenza.

Questa configurazione è stata scelta testandola su una vasta collezione di istanze di proprietà di Ilog (si veda [4]). Se necessario, anche senza ricorrere all'uso delle callbacks, il funzionamento di CPLEX8 può essere profondamente modificato in ogni aspetto dell'algoritmo agendo sui parametri che mette a disposizione.

2.2 Modalità d'uso di CPLEX

CPLEX offre due modalità d'uso.

Algoritmo 1: Cut & Dive; taratura default di CPLEX8.

Input: a: rilassamento continuo del IP

1: applica il preprocessor e il prober ad a
2: risolvi a
3: **while** non hai aggiunto troppi tagli e puoi generarne ancora **do**
4: separa la soluzione frazionaria di a, aggiungendo i tagli in modo statico
5: **end while**
6: genera due figli facendo branch su una variabile frazionaria
7: **repeat**
8: $a \leftarrow$ un nodo in coda, di costo pari al best-bound
9: prova a generare dei tagli
10: crea due nodi con un branch su una variabile frazionaria
11: $a \leftarrow$ uno dei due figli appena creati
12: **dive**(a)
13: **until** non ci sono più nodi nella coda di attesa

dive:

Input: a: un nodo attivo

14: **loop**
15: risolvi il rilassamento del nodo a
16: **if** a è feasible, la soluzione è frazionaria e inferiore all'incumbent **then**
17: genera due figli facendo branch su una variabile frazionaria
18: $a \leftarrow$ uno dei due figli
19: **else**
20: **return**
21: **end if**
22: **end loop**

La prima, e la più semplice, è la modalità interattiva: in questa modalità CPLEX si presenta come una qualsiasi altra applicazione con cui si interagisce a riga di comando. Si ha la possibilità di descrivere il modello secondo una sintassi comoda e intuitiva o di importarlo da un file; si possono assegnare valori opportuni ai parametri per configurare l'ottimizzatore, e si può chiedere l'ottimizzazione dell'istanza. L'ottimizzazione può essere interrotta per assegnare un valore diverso a uno dei parametri per poi essere ripresa.

In alternativa, le funzionalità di CPLEX sono a disposizione sotto forma di libreria linkabile in modo dinamico. In questo modo è possibile usare CPLEX come solver a scatola chiusa all'interno di un'applicazione (per esempio un controllore real-time o un software interattivo), in modo che l'utente dell'applicazione non debba interagire direttamente con CPLEX. Solo in questa modalità si possono sfruttare le funzionalità avanzate che consentono di costruire un Branch&Cut usando i componenti di CPLEX.

Un software di Branch&Cut sviluppato in ambito di ricerca richiede un buon numero di funzionalità dedicate alla parametrizzazione del codice, alla raccolta di dati statistici relativi all'ottimizzazione in corso etc. Basandoci sul fatto che CPLEX è uno strumento di riferimento per questo settore e prevedendo che molti dei progetti di ricerca del gruppo di Ricera Operativa richiederanno l'implementazione di un Branch&Cut basato su CPLEX, io e Gianni Codato abbiamo deciso di progettare e implementare GBF (General Branch&Cut Framework): un framework di base da usare come punto di partenza per le varie implementazioni.

In questo modo si favorisce il riutilizzo di codice da un progetto all'altro, evitando inutili perdite di tempo dovute alla reimplementazione di funzionalità di base o alla fusione di codici sviluppati secondo criteri incompatibili.

Il framework, la cui descrizione è contenuta nei capitoli 5 e 6, è stato effettivamente usato per le nostre tesi di laurea e si è rivelato molto utile.

Capitolo 3

Qualità dei tagli e loro gestione

È stato chiaro fin dalle prime implementazioni della callback – dei semplici oggetti-wrapper per il codice C scritto dal Prof. Caprara – che aggiungere troppi tagli è decisamente deleterio. Lo si può capire anche dal fatto che CPLEX funzioni secondo l'algoritmo Cut&Dive.

Visto che sui problemi di questo test-bed il separatore $0-1/2$ riesce a produrre una grandissima quantità di tagli ci si è posti il problema di come selezionarli. Partendo dalla metrica presentata in [5] è stata quindi sviluppata una procedura di selezione basata sulle proprietà geometriche dei tagli; questa, accompagnata da una strategia di applicazione che si basa sul Cut&Dive ha prodotto risultati molto buoni.

Il fatto che la selezione si basi solo sulle proprietà geometriche dei tagli fa sì che il metodo sia generale e applicabile a qualsiasi famiglia di tagli.

3.1 Metrica di qualità

Il problema di programmazione lineare intera in cui si possono usare i tagli $0-1/2$ è così definito:

$$A \in \mathbb{Z}^{m \times n} \tag{3.1}$$

$$\boldsymbol{c}, \boldsymbol{x} \in \mathbb{Z}^n \tag{3.2}$$

$$\boldsymbol{b} \in \mathbb{Z}^m \tag{3.3}$$

$$\begin{cases} \min_{\boldsymbol{x}}(\boldsymbol{c}^T \boldsymbol{x}) \\ A\boldsymbol{x} \geq \boldsymbol{b} \end{cases} \tag{3.4}$$

La disequazione che descrive un taglio è

$$\gamma := \quad \boldsymbol{a}^T \boldsymbol{x} \geq b \tag{3.5}$$

Il taglio individua un semispazio il cui bordo è l'iperpiano

$$\pi := \quad \boldsymbol{a}^T \boldsymbol{x} = b \tag{3.6}$$

La distanza euclidea dalla soluzione frazionaria $\boldsymbol{x}^*$ al piano p è definita come

$$\mathrm{d}(\boldsymbol{x}^*, \pi) := \frac{|\boldsymbol{a}^T \boldsymbol{x}^* - b|}{\|\boldsymbol{a}\|} \tag{3.7}$$

L'efficacia di un taglio violato è proporzionale alla sua distanza dal punto frazionario da separare. Quindi, se abbiamo un gran numero di tagli a disposizione un primo criterio da usare per selezionarli è quello della profondità geometrica.

La definizione di efficacia è:

$$\mathrm{eff}(\boldsymbol{x}^*, \gamma) := \frac{b - \boldsymbol{a}^T \boldsymbol{x}^*}{\|\boldsymbol{a}\|} \tag{3.8}$$

I segni del numeratore sono scelti in modo tale che per i tagli violati $\mathrm{eff}(\boldsymbol{x}^*, \gamma) > 0$ e viceversa. Dopo aver ordinato il pool per efficacia, si può anche imporre ai tagli da aggiungere una soglia di efficacia minima:

$$\mathrm{eff}(\boldsymbol{x}^*, \gamma) \geq \xi \tag{3.9}$$

Il passo successivo è quello di evitare di aggiungere alla formulazione tagli troppo simili fra loro; aggiungere un taglio molto simile a tagli già presenti appesantisce la formulazione senza migliorarla. Due tagli sono simili se hanno (quasi) la stessa efficacia e i loro iperpiani sono (quasi) paralleli; due iperpiani π_1, π_2 sono ortogonali fra loro se lo sono i loro vettori direzionali, ovvero se è

$$\boldsymbol{a}_1^T \boldsymbol{a}_2 = 0 \tag{3.10}$$

Più in generale si può scegliere un $\epsilon \in [0, 1)$ e imporre che per tutti i tagli aggiunti, a due a due, sia verificata la condizione

$$\mathrm{par}(\gamma_1, \gamma_2) := \frac{|\boldsymbol{a}_1^T \boldsymbol{a}_2|}{\|\boldsymbol{a}_1\| \, \|\boldsymbol{a}_2\|} \leq \epsilon \tag{3.11}$$

Scegliere $\epsilon = 0$, per esempio, assicura che i supporti dei due tagli siano disgiunti (iperpiani ortogonali fra loro).

Le stesse considerazioni si trovano in [5] per l'uso dei tagli *lift & project*. In quel caso però, il separatore usato aveva l'interessante caratteristica di poter generare tagli costruiti a partire da una precisa variabile con valore frazionario; la condizione (3.11) è usata con $\epsilon = 0.999$ per scartare i tagli duplicati e gli autori dichiarano di non aver fatto prove con valori inferiori.

3.2 Strategia di applicazione

Anche questa è una componente fondamentale di un Branch&Cut; in [5] è stato fatto un lavoro molto interessante per formulare un algoritmo che sia in grado di decidere automaticamente quanto spesso chiamare il separatore durante la visita dell'albero. In quell'articolo però si descrive il progetto completo di un algoritmo Branch&Cut.

Il lavoro di questa tesi è invece finalizzato alla misura della utilità di una famiglia di tagli. Per questo motivo si è deciso di applicare i tagli $0-1/2$ allo stesso modo in cui CPLEX applica i suoi nella configurazione di default (vedi algoritmo Cut&Dive a pagina 20). L'algoritmo Cut&Dive prevede di usare il separatore più volte al nodo radice e in seguito solo dopo i backtrace, fino a quando è possibile generare tagli efficaci o non se ne sono aggiunti troppi. I tagli generati sono aggiunti staticamente alla formulazione. Per questioni di efficienza può essere utile chiamare il separatore con una frequenza minore di ogni backtrace.

La singola chiamata al separatore procede secondo l'algoritmo 2.

Alcune osservazioni:

1. la condizione (3.11) di non parallelismo viene imposta solo ai tagli aggiunti in una singola chiamata e non a tutti i tagli aggiunti durante l'ottimizzazione; questo consente di evitare che un nuovo taglio molto efficace non possa essere aggiunto perché parallelo a un taglio aggiunto in precedenza e ormai dominato.

2. la soglia per la condizione (3.9) non è fissa; al primo tentativo riuscito di generazione viene fissata al minimo fra il valore fornito in ingresso e il 70% dell'efficacia massima dei tagli generati. Poi, durante l'ottimizzazione, si conta il numero di volte in cui nel pool non ci sono tagli che soddisfino la condizione di efficacia; ogni 20, la soglia di efficacia minima viene abbassata; questo compensa il fatto che mano a mano che il rilassamento si avvicina al convex hull è sempre più difficile avere tagli molto efficaci; in questo modo non si impedisce l'aggiunta di tagli nelle fasi avanzate dell'ottimizzazione.

3. dopo che il pool è stato ordinato in ordine decrescente di efficacia, se ha raggiunto una dimensione eccessiva si cancellano un po' di tagli a partire da quello meno efficace; questo rappresenta un modo semplice ma efficace di effettuare una pulizia del pool.

   Per esempio, un taglio efficace scartato per la condizione (3.11), alle chiamate successive sarà sicuramente poco efficace perchè sono stati aggiunti da poco tagli simili a lui; il taglio scenderà in fondo al pool per l'ordinamento e quindi, dopo qualche chiamata, sarà eliminato.

4. Il motivo per cui i tagli scartati per parallelismo non sono cancellati subito ma mantenuti nel pool è il sequente: la presenza del pool supplisce alla natura della procedura **sep012cut**; come spiegato nel capitolo sui tagli $0-1/2$, per poter generare tagli, la procedura indebolisce la formulazione riducendo la matrice dei vincoli alla matrice di incidenza di un grafo: è possibile che tagli poco profondi generati a una chiamata siano molto più efficaci rispetto a un altro x^* per il quale non possono essere direttamente generati.

5. La procedura **sep012cut** viene fatta operare non sulla formulazione originale, ma sulla formulazione arricchita da tutti i tagli aggiunti in precedenza; questo si basa sul fatto che per generare tagli $0-1/2$ la procedura può operare solo sui vincoli che abbiano uno *slack* nell'intervallo $[0,1)$. In presenza di un *gap* elevato è verosimile che dopo l'aggiunta di pochi

Algoritmo 2: Separa; procedura seguita per separare un punto frazionario

Input: x^*: punto frazionario da separare; P: pool di tagli generati in precedenza; m: massima dimensione del pool; S: insieme vuoto di tagli; ϵ: massimo parallelismo; ξ: minima efficacia; LP: insieme di tagli della formulazione corrente

1: $P = P \cup \{\textbf{sep012cut}(x^*, LP)\}$
2: ordina P per efficacia decrescente
3: **if** questa è la prima volta che si generano tagli **then**
4: $\quad \xi = \min(\xi; 0.7\,\text{eff}(P[0]))$
5: **end if**
6: **if** $|P| > m$ **then**
7: $\quad$ elimina gli ultimi $|P| - m$ tagli di P
8: **end if**
9: $miss = 0$
10: **if** $\text{eff}(P[0]) > \xi$ **then**
11: $\quad LP = LP \cup P[0]$
12: **else**
13: $\quad miss = miss + 1$
14: $\quad$ **if** miss == 20 **then**
15: $\quad\quad miss = 0$
16: $\quad\quad \xi = \xi - 0.03$
17: $\quad$ **end if**
18: $\quad$ **return**
19: **end if**
20: $S = S \cup \{P[0]\}$
21: $i = 1$
22: **while** $\text{eff}(P[i]) > \xi$ **do**
23: $\quad$ **if** $\forall\, c \in S : \text{par}(c, P[i]) < \epsilon$ **then**
24: $\quad\quad LP = LP \cup P[i]$
25: $\quad\quad S = S \cup \{P[i]\}$
26: $\quad$ **end if**
27: $\quad i = i + 1$
28: **end while**
29: **return**

sep012cut:

Input: x^*: punto frazionario da separare; LP: formulazione lp.

30: **return** tagli generati dalla procedura C per separare x^* a partire dai vincoli di LP.

gruppi di tagli efficaci e di vincoli di branching, i vincoli della formulazione originale siano troppo distanti dal punto frazionario e quindi non più utilizzabili per produrre nuovi tagli $0 - 1/2$.

Capitolo 4

Test

Questo capitolo contiene la descrizione delle prove sperimentali effettuate per verificare l'efficacia delle tecniche di selezione proposte. Sono presentati, nell'ordine, il test-bed, il metodo usato per calcolare gli speedup e i risultati ottenuti.

4.1 Testbed

I tagli 0-1/2 si possono generare solo a partire dalla parte intera della formulazione di un problema MIP.

La maggior parte dei test è stata condotta su problemi di *Soddisfacibilità Booleana*; alcune prove sono state eseguite anche su istanze appartenenti alla collezione MIPLIB3 e su istanze di problemi di schedulazione.

4.1.1 Soddisfacibilità Booleana

Il problema di Soddisfacibilità Booleana (SAT) e la sua variante Massima Soddisfacibilità ($MAX\text{-}SAT$) sono problemi fondamentali nei campi dell'intelligenza artificiale, della logica e della complessità computazionale. La definizione di questi problemi è la seguente:

Una variabile booleana x è una variabile che può assumere esclusivamente i valori *vero* o *falso* (di solito indicati rispettivamente con 1 e 0). Una formula booleana è una combinazione di variabili booleane che usa gli operatori logici *negazione* ($\overline{x}$), *disgiunzione* ($\vee$), e *congiunzione* ($\wedge$). Una variabile o la sua negazione prende il nome di *letterale*, mentre una disgiunzione di letterali si chiama *clausola*.

Dato un insieme di clausole $C_1, C_2, \ldots, C_m$ sulle variabili $x_1, x_2, \ldots, x_n$, il problema di soddisfacibilità booleana consiste nel determinare se la formula

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

sia soddisfacibile; ovvero, se esista una assegnamento di valori alle variabili tale che il valore della formula sia *vero*.

Il problema di massima soddisfacibilità consiste nel trovare un assegnamento di valori alle variabili che renda *vero* il valore del massimo numero di clausole.

Una istanza di problema SAT si può ridurre a una istanza di problema IP nel seguente modo: il problema IP ha una variabile intera 0-1 per ogni variabile del SAT; per ogni clausola c'_k si scrive un vincolo c_k della forma $\sum_i x_i + \sum_j (1 - x_j) \geq 1$, dove $\{x_j\}$ è l'insieme delle variabili che compaiono negate nella clausola, mentre $\{x_i\}$ è l'insieme delle varibili presenti nella clausola senza negazione.

La funzione obiettivo della formulazione IP è ininfluente; per esempio si può usare $\min(z)$, dove z è una variabile con limite minimo 0. Dal punto di vista pratico, però, ho ritenuto opportuno introdurre una funzione obiettivo non costante, per evitare che gli algoritmi di CPLEX girassero attorno al politopo: il fatto che la funzione obiettivo sia costante rende troppo casuali sia le soluzioni dei rilassamenti continui che le scelte di branching. Questa impressione deriva da fatto che con funzione obiettivo costante il separatore 0-1/2 genera sempre tagli molto profondi; il comportamento tipico di un separatore, invece, è quello di perdere efficacia (ovvero di generare tagli poco profondi) con l'avanzare dell'esplorazione dell'albero di branching.

Trasformare in IP una istanza di MAX-SAT è leggermente più complicato: al membro destro di ogni vincolo c_k si sottrae la variabile z_k (intera 0-1) e si sottrae 1 al termine noto. La funzione obiettivo è $\max\left(\sum_k z_k\right)$.

La collezione di problemi SAT usata per questo lavoro è quella della seconda edizione del DIMACS Challenge; si possono trovare i files originali e maggiori informazioni in [3]. Convertendo questi problemi in LP e scartando le istanze con meno di 900 o più di 10000 vincoli si sono ottenute 444 istanze, metà delle quali di tipo SAT e metà di tipo di MAXSAT.

4.2 Misura delle prestazioni

Misurare l'efficacia di un algoritmo di Branch& Cut non è semplice.

Il comportamento degli algoritmi di Branch&Cut dipende fortemente dall'istanza a cui sono applicati; quando la dimensione del testbed diventa significativa, è praticamente impossibile ottenere una taratura dell'insieme dei parametri messi a disposizione dal software che sia la più veloce su tutte le istanze. La documentazione stessa di CPLEX invita l'utente che abbia problemi di prestazioni a provare a modificare la taratura di default delle varie componenti del software (presolver, euristica, generatori delle varie famiglie di tagli, scelta della variabile su cui fare branch e del nodo successivo, etc); questo nonostante CPLEX sia tarato su una collezione di problemi molto grande (si veda [4]). Questo metodo di misura dello speedup è stato elaborato a partire da quanto si trova in [4] e [5].

Una volta ottenuti i tempi di soluzione delle istanze con le varie configurazioni testate è opportuno dividere la collezione nei seguenti insiemi:

1. Istanze risolte da *entrambe* le configurazioni in un tempo inferiore a una certa soglia di tempo minimo – una soglia fra i 10 e i 20 secondi dovrebbe essere ragionevole.

   Queste istanze vanno scartate perchè poco significative: sono evidentemente troppo facili.

2. Istanze risolte da entrambe le configurazioni in un tempo maggiore alla soglia di tempo minimo.

   Su queste istanze si calcola una media dei rapporti fra i tempi.

3. Istanze risolte da una sola delle due configurazioni.

   Per queste invece si confronta il numero di istanze risolte da una configurazione ma non dall'altra e viceversa.

4. Le rimanenti istanze non risolte da nessuna delle due configurazioni.

   Di queste prove si calcola la media dei rapporti fra i gap rimanenti allo scadere del tempo massimo.

Avendo deciso di scartare alcune istanze perché troppo semplici, è importante scartare quelle risolte da entrambe le configurazioni; se invece si scartano solo quelle risolte in poco tempo da una delle due (per esempio una versione vecchia dell'algoritmo), la si penalizza notevolmente. Infatti, in questo modo si escludono dal testbed anche le istanze difficili su cui quella versione è molto più efficace dell'altra.

Il fatto che su alcune istanze si possa avere un guadagno enorme – dai tre ordini di grandezza in su – fra una configurazione e l'altra suggerisce di usare la media geometrica (M_g) al posto di quella aritmetica (M_a) (vedi [4]). La media geometrica è infatti molto più stabile in presenza di un elemento che sia qualche ordine di grandezza superiore alla media di tutti gli altri. Lo si può capire esaminando l'espressione delle due medie in funzione di un elemento dell'insieme:

$$\begin{aligned} M_a(x_j) &= \frac{1}{n}\left(\sum_{i\neq j} x_i\right) + \frac{1}{n}x_j \\ M_g(x_j) &= \left(\prod_{i\neq j} x_i\right)^{\frac{1}{n}} x_j^{\frac{1}{n}} \end{aligned}$$

e le derivate parziali rispetto allo stesso elemento:

$$\begin{aligned} \frac{\partial M_a(x_0, x_1, \ldots, x_n)}{\partial x_j} &= \frac{1}{n} \\ \frac{\partial M_g(x_0, x_1, \ldots, x_n)}{\partial x_j} &= \frac{1}{n} M_g \frac{1}{x_j} = \frac{1}{n}\left(\prod_{i\neq j} x_i\right)^{\frac{1}{n}} \frac{1}{x_j^{\frac{n-1}{n}}} \end{aligned}$$

4.3 Test eseguiti

Per problemi di tempo, solo alcune delle prove sono state eseguite su tutte le istanze di SAT e MAXSAT descritte nella sezione precedente. La maggior parte sono state effettuate su un testbed ridotto: metà della collezione SAT (le istanze dispari dell'ordinamento alfabetico). Se non diversamente specificato si intende che il testbed usato sia quello ridotto.

Per quanto riguarda le diverse configurazioni del separatore $0 - 1/2$, sono indicati volta per volta i valori assegnati ai vari parametri:

step: il separatore è applicato ogni "step" backtrack;

max_min_eff: il parametro con cui si regola la soglia di efficacia minima; si veda l'algoritmo 2;

max_scal_prod: la soglia di massimo parallelismo;

max_node_cuts: il numero di massimo di tagli ammessi, espresso come frazione del numero di righe della formulazione originale;

recomb: valore booleano, indica se sia attiva o meno la produzione di tagli di ordine superiore al primo.

4.3.1 Test preliminari

Questa prima serie di test sono stati condotti in primo luogo per avere un termine di paragone per i test successivi; in secondo luogo per verificare quanto siano determinanti i tagli di CPLEX per le prestazioni sui problemi SAT e MAXSAT.

Come spiegato nel capitolo 2, la presenza di una *cut callback* costringe CPLEX a disabilitare parte delle tecniche di preprocessing che gli consentono di migliorare la fomulazione matematica del rilassamento continuo. Per questo motivo tutte le misure di prestazioni saranno effettuate usando come punto di riferimento una configurazione di CPLEX con una *cut callback* che non fa nulla.

I risultati di queste prime prove sono piuttosto interessanti: indicano che CPLEX non dispone di tagli adatti a risolvere queste istanze. Anche l'azione del preprocessor è negativa.

CPLEX Default

```
>>>>>> ./cplex_def versus null_cbk <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          19
mean speedup:       0.89 (./cplex_def / null_cbk)

Instances solved by neither:
count:          16

Instances solved in ./cplex_def but not in null_cbk:
count:          4
mean time:      636 sec

Instances solved in null_cbk but not in ./cplex_def:
count:          0
```

CPLEX Default senza tagli

```
>>>>>> ./cplex_def_nocuts versus null_cbk <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          19
mean speedup:       1.10 (./cplex_def_nocuts / null_cbk)

Instances solved by neither:
count:          18

Instances solved in ./cplex_def_nocuts but not in null_cbk:
count:          3
mean time:      566 sec

Instances solved in null_cbk but not in ./cplex_def_nocuts:
count:          0
```

CPLEX Null Callback senza tagli

```
>>>>>> ./null_cbk_nocuts versus null_cbk <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          16
mean speedup:       1.42 (./null_cbk_nocuts / null_cbk)

Instances solved by neither:
count:          20

Instances solved in ./null_cbk_nocuts but not in null_cbk:
count:          1
mean time:      42 sec

Instances solved in null_cbk but not in ./null_cbk_nocuts:
count:          2
mean time:      15 sec
```

CPLEX Null Callback senza tagli rispetto al Default

```
>>>>>> ./null_cbk_nocuts versus cplex_def <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  18
mean speedup:           2.07 (./null_cbk_nocuts / cplex_def)

Instances solved by neither:
count:                  16

Instances solved in ./null_cbk_nocuts but not in cplex_def:
count:                  0

Instances solved in cplex_def but not in ./null_cbk_nocuts:
count:                  5
mean time:              140 sec
```

4.3.2 Taratura buona

Questi sono i risultati ottenuti con una buona taratura per i problemi SAT; in questo caso il testbed è quello completo.

Lo speedup medio è di 3.25, ma è importante anche notare che 23 problemi che non era possibile risolvere entro il limite di tempo con CPLEX, sono stati risolti in un tempo medio di 16 secondi. Altri 7, invece, risolti da CPLEX in una media di 360 secondi, non sono stati risolti entro il limite di tempo in presenza dei tagli $0 - 1/2$.

Più di metà delle istanze è scartata perchè risolte in meno di 20 secondi da entrambi i solver.

Parametri: step 4, max_min_eff 0.2, max_scal_prod 0.1, max_node_cuts 0.3, recomb.

```
>>>>>> step_4_ort_0.1_rec versus ilo <<<<<<

Ignored 129 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  30
mean speedup:           3.25 (step_4_ort_0.1_rec / ilo)

Instances solved by neither:
count:                  31

Instances solved in step_4_ort_0.1_rec but not in ilo:
count:                  23
mean time:              16 sec

Instances solved in ilo but not in step_4_ort_0.1_rec:
count:                  7
mean time:              360 sec
```

Questa taratura, purtroppo, non è altrettanto efficace sui problemi MAX-SAT:

```
>>>>>> ort_0.1_step_4_rec versus ilo <<<<<<

Ignored 80 instances
TIMEOVER value: 2000 sec
All means are geometric

Instances solved by both solvers:
count:                  67
mean speedup:           1.39 (ort_0.1_step_4_rec / ilo)

Instances solved by neither:
count:                  46
mean gap ratio:         0.927825

Instances solved in ort_0.1_step_4_rec but not in ilo:
count:                  22
mean time:              37 sec

Instances solved in ilo but not in ort_0.1_step_4_rec:
count:                  7
mean time:              802 sec
```

4.3.3 Taratura di partenza

Si tratta della configurazione ottenuta intuitivamente durante la fase di test del codice; costituisce il punto di partenza per le prove presentate sotto in cui si è cercato di ottimizzare la scelta dei parametri su un numero ristretto di istanze. Parametri: step 10, max_min_eff 0.2, max_scal_prod 0.3, max_node_cuts 0.3, recomb.

```
>>>>>> ort_0.3_recomb versus ilo <<<<<<

Ignored 129 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  32
mean speedup:           2.60 (ort_0.3_recomb / ilo)

Instances solved by neither:
count:                  28

Instances solved in ort_0.3_recomb but not in ilo:
count:                  26
mean time:              42 sec

Instances solved in ilo but not in ort_0.3_recomb:
count:                  5
mean time:              288 sec
```

Questa taratura è stata provata anche su tutte le istanze MAXSAT, sulle quali lo speedup è meno alto.

```
>>>>>> ort_0.3_recomb versus ilo <<<<<<

Ignored 82 instances
TIMEOVER value: 2000 sec
All means are geometric

Instances solved by both solvers:
count:                  65
mean speedup:           1.76 (ort_0.3_recomb / ilo)

Instances solved by neither:
count:                  45
mean gap ratio:         0.964953

Instances solved in ort_0.3_recomb but not in ilo:
count:                  23
mean time:              43 sec

Instances solved in ilo but not in ort_0.3_recomb:
count:                  7
mean time:              563 sec
```

4.3.4 Taratura peggiore

In questa configurazione i tagli sono aggiunti non appena è possibile generarli senza alcun tipo di selezione; l'unica forma di separazione "intelligente" è data dal fatto che il separatore è invocato dopo ogni backtrace.

Sono state fatte due prove: la prima con un limite massimo di quantità pari a cinque volte il numero delle righe della formulazione originale; la seconda con il limite massimo di 0.3 volte il numero di righe della formulazione originale.

I risultati confermano che, anche facendo scelte casuali, è meglio limitare parecchio il numero di tagli aggiunti alla formulazione.

```
>>>>>> tanti versus ../ilo <<<<<<

Ignored 61 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  19
mean speedup:           0.35 (naive_tanti / ../ilo)

Instances solved by neither:
count:                  14

Instances solved in tanti but not in ../ilo:
count:                  8
mean time:              224 sec

Instances solved in ../ilo but not in tanti:
count:                  5
mean time:              419 sec
```

```
>>>>>> pochi versus ../ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  18
mean speedup:           1.07 (naive_pochi / ../ilo)

Instances solved by neither:
count:                  17

Instances solved in pochi but not in ../ilo:
count:                  5
mean time:              300 sec

Instances solved in ../ilo but not in pochi:
count:                  1
mean time:              797 sec
```

Sulle istanze MAXSAT:

```
>>>>>> naive versus ilo <<<<<<

Ignored 62 instances
TIMEOVER value: 2000 sec
All means are geometric

Instances solved by both solvers:
count:                  59
mean speedup:           0.25 (naive_tanti / ilo)

Instances solved by neither:
count:                  58
mean gap ratio:         0.755302

Instances solved in naive but not in ilo:
count:                  10
mean time:              91 sec

Instances solved in ilo but not in naive:
count:                  33
mean time:              303 sec
```

```
>>>>>> naive_pochi versus ilo <<<<<<

Ignored 63 instances
TIMEOVER value: 2000 sec
All means are geometric

Instances solved by both solvers:
count:                  77
mean speedup:           0.50 (naive_pochi / ilo)

Instances solved by neither:
count:                  60
mean gap ratio:         0.948889

Instances solved in naive_pochi but not in ilo:
count:                  8
mean time:              88 sec

Instances solved in ilo but not in naive_pochi:
count:                  14
mean time:              438 sec
```

4.3.5 Campionamento sui parametri per una taratura sistematica

In questa seconda fase di test, partendo dalla taratura 4.3.3, si è cercato di ottimizzare sistematicamente la configurazione, un parametro alla volta. I risultati sono riassunti a pagina 55, nelle tabelle 4.1, 4.2 e 4.3.

Step di applicazione

Parametri fissi: max_min_eff 0.2, max_scal_prod 0.3, max_node_cuts 0.3, recomb.

In questo caso le medie degli speedup hanno un comportamento non correlato con i valori del parametro. Per le prove successive è stato usato come valore di step 4.

```
>>>>>> ./skip_1 versus ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:          16
mean speedup:       2.76 (./skip_1 / ilo)

Instances solved by neither:
count:          4

Instances solved in ./skip_1 but not in ilo:
count:          6
mean time:      22 sec

Instances solved in ilo but not in ./skip_1:
count:          4
mean time:      719 sec
```

```
>>>>>> ./skip_4 versus ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:          15
mean speedup:       3.80 (./skip_4 / ilo)

Instances solved by neither:
count:          8

Instances solved in ./skip_4 but not in ilo:
count:          9
mean time:      107 sec

Instances solved in ilo but not in ./skip_4:
count:          5
```

```
>>>>>> ./skip_5 versus ilo <<<<<<

Ignored 65 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:          17
mean speedup:       3.42 (./skip_5 / ilo)

Instances solved by neither:
count:          6

Instances solved in ./skip_5 but not in ilo:
count:          9
mean time:      112 sec

Instances solved in ilo but not in ./skip_5:
count:          4
mean time:      415 sec
mean time:      396 sec
```

```
>>>>>> ./skip_7 versus ilo <<<<<<

Ignored 61 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:          13
mean speedup:       2.35 (./skip_7 / ilo)

Instances solved by neither:
count:          8

Instances solved in ./skip_7 but not in ilo:
count:          9
mean time:      118 sec

Instances solved in ilo but not in ./skip_7:
count:          6
mean time:      480 sec
```

```
>>>>>> ./skip_10 versus ilo <<<<<<

Ignored 60 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:          16
mean speedup:       3.03 (./skip_10 / ilo)

Instances solved by neither:
count:          8

Instances solved in ./skip_10 but not in ilo:
count:          8
mean time:      85 sec

Instances solved in ilo but not in ./skip_10:
count:          2
mean time:      382 sec
```

```
>>>>>> ./skip_13 versus ilo <<<<<<

Ignored 65 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:          17
mean speedup:       2.37 (./skip_13 / ilo)

Instances solved by neither:
count:          7

Instances solved in ./skip_13 but not in ilo:
count:          9
mean time:      141 sec

Instances solved in ilo but not in ./skip_13:
count:          4
mean time:      710 sec
```

```
>>>>>> ./skip_16 versus ilo <<<<<<

Ignored 65 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:          15
mean speedup:       3.31 (./skip_16 / ilo)

Instances solved by neither:
count:          6

Instances solved in ./skip_16 but not in ilo:
count:          9
mean time:      119 sec

Instances solved in ilo but not in ./skip_16:
count:          6
mean time:      596 sec
```

Ortogonalità massima

Parametri fissi: step 4, max_min_eff 0.2, max_node_cuts 0.3, recomb.

Variando la massima ortogonalità consentita (condizione (3.11)) da 0.001 a 0.9, su metà delle istanze SAT (111) si sono ottenuti i risultati esposti in seguito.

Se ne deduce che conviene imporre una soglia stringente per la condizione di massimo parallelismo.

Il buon risultato ottenuto con i valori 0.9 e 1 è probabilmente dovuto al fatto che se non si pretende che i tagli siano perpendicolari fra loro, si spende meno tempo nel separatore.

```
>>>>>> ./ort_0.001 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          14
mean speedup:       3.71 (./ort_0.001 / ../ilo)

Instances solved by neither:
count:          16

Instances solved in ./ort_0.001 but not in ../ilo:
count:          9
mean time:      29 sec

Instances solved in ../ilo but not in ./ort_0.001:
count:          4
mean time:      357 sec
```

```
>>>>>> ./ort_0.1 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          13
mean speedup:       3.91 (./ort_0.1 / ../ilo)

Instances solved by neither:
count:          16

Instances solved in ./ort_0.1 but not in ../ilo:
count:          9
mean time:      29 sec

Instances solved in ../ilo but not in ./ort_0.1:
count:          5
mean time:      414 sec
```

```
>>>>>> ./ort_0.3 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          13
mean speedup:       3.14 (./ort_0.3 / ../ilo)

Instances solved by neither:
count:          13

Instances solved in ./ort_0.3 but not in ../ilo:
count:          12
mean time:      69 sec

Instances solved in ../ilo but not in ./ort_0.3:
count:          5
mean time:      444 sec
```

```
>>>>>> ./ort_0.5 versus ../ilo <<<<<<

Ignored 68 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          15
mean speedup:       3.05 (./ort_0.5 / ../ilo)

Instances solved by neither:
count:          15

Instances solved in ./ort_0.5 but not in ../ilo:
count:          10
mean time:      84 sec

Instances solved in ../ilo but not in ./ort_0.5:
count:          2
mean time:      994 sec
```

```
>>>>>> ./ort_0.7 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          16
mean speedup:       2.34 (./ort_0.7 / ../ilo)

Instances solved by neither:
count:          15

Instances solved in ./ort_0.7 but not in ../ilo:
count:          10
mean time:      38 sec

Instances solved in ../ilo but not in ./ort_0.7:
count:          2
mean time:      311 sec
```

```
>>>>>> ./ort_0.9 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          14
mean speedup:       3.16 (./ort_0.9 / ../ilo)

Instances solved by neither:
count:          18

Instances solved in ./ort_0.9 but not in ../ilo:
count:          7
mean time:      24 sec

Instances solved in ../ilo but not in ./ort_0.9:
count:          4
mean time:      544 sec
```

```
>>>>>> ./ort_1.0 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          14
mean speedup:       3.67 (./ort_1.0 / ../ilo)

Instances solved by neither:
count:          18

Instances solved in ./ort_1.0 but not in ../ilo:
count:          8
mean time:      63 sec

Instances solved in ../ilo but not in ./ort_1.0:
count:          4
mean time:      780 sec
```

Efficacia minima

Parametri fissi: step 4, max_scal_prod 0.1, max_node_cuts 0.3, recomb.

Come per la soglia di massimo parallelismo, anche quella di efficacia minima va tenuta alta.

```
>>>>>> ./mineff_0 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          15
mean speedup:       2.59 (./mineff_0 / ../ilo)

Instances solved by neither:
count:          18

Instances solved in ./mineff_0 but not in ../ilo:
count:          8
mean time:      50 sec

Instances solved in ../ilo but not in ./mineff_0:
count:          3
mean time:      386 sec
```

```
>>>>>> ./mineff_0.05 versus ../ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          14
mean speedup:       3.35 (./mineff_0.05 / ../ilo)

Instances solved by neither:
count:          15

Instances solved in ./mineff_0.05 but not in ../ilo:
count:          11
mean time:      19 sec

Instances solved in ../ilo but not in ./mineff_0.05:
count:          5
mean time:      419 sec
```

```
>>>>>> ./mineff_0.15 versus ../ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          14
mean speedup:       3.35 (./mineff_0.15 / ../ilo)

Instances solved by neither:
count:          15

Instances solved in ./mineff_0.15 but not in ../ilo:
count:          11
mean time:      33 sec

Instances solved in ../ilo but not in ./mineff_0.15:
count:          5
mean time:      414 sec
```

```
>>>>>> ./mineff_0.10 versus ../ilo <<<<<<

Ignored 65 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          14
mean speedup:       2.84 (./mineff_0.10 / ../ilo)

Instances solved by neither:
count:          14

Instances solved in ./mineff_0.10 but not in ../ilo:
count:          12
mean time:      61 sec

Instances solved in ../ilo but not in ./mineff_0.10:
count:          6
mean time:      462 sec
```

```
>>>>>> ./mineff_0.20 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          13
mean speedup:       3.92 (./mineff_0.20 / ../ilo)

Instances solved by neither:
count:          17

Instances solved in ./mineff_0.20 but not in ../ilo:
count:          9
mean time:      29 sec

Instances solved in ../ilo but not in ./mineff_0.20:
count:          5
mean time:      414 sec
```

```
>>>>>> ./mineff_0.25 versus ../ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:          13
mean speedup:       3.99 (./mineff_0.25 / ../ilo)

Instances solved by neither:
count:          15

Instances solved in ./mineff_0.25 but not in ../ilo:
count:          11
mean time:      51 sec

Instances solved in ../ilo but not in ./mineff_0.25:
count:          5
mean time:      414 sec
```

In breve

Nelle tabelle 4.1, 4.2 e 4.3 sono riassunti i risultati delle prove per l'ottimizzazione sistematica.

Lo speedup della configurazione migliore della tabella del parametro step (step 4, seconda riga) è diverso da quello della stessa configurazione nella tabella della prova sul parametro max_scal_prod (max_scal_prod 0.3, terza riga). Questo è dovuto al fatto che le due prove sono state eseguite su computer diversi.

La taratura migliore, per il testbed ridotto, è quindi quasi 4 volte più veloce di CPLEX in configurazione Null Callback. I parametri hanno i seguenti valori:

step: 4;

max_scal_prod: 0.1;

max_min_eff: 0.25;

recomb: attivo;

Tabella 4.1: Parametro step

			Non risolte da		
Valore	Num	Speedup	nessuno	$0-1/2$	cplex
1	16	2.76	4	4	6
4	15	3.80	8	5	9
5	17	3.42	6	4	9
7	13	2.35	8	6	9
10	16	3.03	8	2	8
13	17	2.37	7	4	9
16	15	3.31	6	6	9

Tabella 4.2: Parametro max_scal_prod

			Non risolte da		
Valore	Num	Speedup	nessuno	$0-1/2$	cplex
0.001	14	3.71	16	4	9
0.1	13	3.91	16	5	9
0.3	13	3.14	13	5	12
0.5	15	3.05	15	2	10
0.7	16	2.34	15	2	10
0.9	14	3.16	18	4	7
1.0	14	3.67	18	4	8

4.3.6 Contributo delle singole componenti del separatore

Con queste prove si vuol capire quanto sono utili da sole l'efficacia, l'ortogonalità e la ricombinazione dei tagli $0-1/2$ fra loro.

Una e una sola componenete disabilitata

A partire dalla configurazione ottenuta con l'ottimizzazione sistematica, si disattivano le funzionalità una sola per prova. I risultati sono riassunti nella tabella 4.4. Nella prova relativa alla disattivazione della condizione di ortogonalità si è ottenuto un risultato irregolare, dovuto al modo in cui le istanze hanno partecipato alle medie. Si veda il commento fatto al riassunto dettagliato.

Tabella 4.3: Parametro max_min_eff

			Non risolte da		
Valore	Num	Speedup	nessuno	$0 - 1/2$	cplex
0	15	2.59	18	3	8
0.05	14	3.35	15	5	11
0.10	14	2.84	14	6	12
0.15	14	3.35	15	5	11
0.20	13	3.92	17	5	9
0.25	13	3.99	15	5	11

Tabella 4.4: Una e una sola componente disabilitata

			Non risolte da		
Componente	Num	Speedup	nessuno	$0 - 1/2$	cplex
Riferimento	17	3.57	13	3	9
Efficacia	16	3.35	15	5	7
Step	18	2.92	11	2	11
Ricombinazione	17	2.36	14	3	8
Ortogonalità	15	6.94	14	5	8

Riferimento Parametri: step 4, max_min_eff 0.25, max_scal_prod 0.1, max_node_cuts 0.3, recomb.

```
>>>>>> base versus ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:                  17
mean speedup:           3.57 (base / ilo)

Instances solved by neither:
count:                  13


Instances solved in base but not in ilo:
count:                  9
mean time:              139 sec

Instances solved in ilo but not in base:
count:                  3
mean time:              952 sec
```

Ordinamento per efficacia

```
>>>>>> eff versus ilo <<<<<<

Ignored 65 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:                  16
mean speedup:           3.35 (eff / ilo)

Instances solved by neither:
count:                  15

Instances solved in eff but not in ilo:
count:                  7
mean time:              187 sec

Instances solved in ilo but not in eff:
count:                  5
mean time:              906 sec
```

Condizione di ortogonalità

```
>>>>>> ort versus ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:                  15
mean speedup:           6.94 (ort / ilo)

Instances solved by neither:
count:                  14

Instances solved in ort but not in ilo:
count:                  8
mean time:              301 sec

Instances solved in ilo but not in ort:
count:                  5
mean time:              835 sec
```

Lo speedup medio ottenuto da questa prova è dovuto a una infelice combinazione delle istanze a causa della selezione sulla base della soglia del tempo di soluzione. Se si confrontano la configurazione di riferimento e quella con la condizione di ortogonalità disattivata si ottiene il seguente riassunto, che conferma l'utilità della condizione di ortogonalità (si mettano a confronto anche i dati relativi alle istanze non risolte).

```
>>>>>> base versus ort <<<<<<

Ignored 74 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:                  14
mean speedup:           1.16 (base / ort)

Instances solved by neither:
count:                  14

Instances solved in base but not in ort:
count:                  4
mean time:              217 sec

Instances solved in ort but not in base:
count:                  1
mean time:              338 sec
```

Ricombinazione

```
>>>>>> rec versus ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 3600 sec
All means are geometric

Instances solved by both solvers:
count:                  17
mean speedup:           2.36 (rec / ilo)

Instances solved by neither:
count:                  14

Instances solved in rec but not in ilo:
count:                  8
mean time:              589 sec

Instances solved in ilo but not in rec:
count:                  3
mean time:              723 sec
```

Una e una sola componente abilitata

A partire da una configurazione di tipo tradizionale, si attivano le funzionalità una sola per prova. La tabella 4.5 contiene un riassunto dei risultati.

Tabella 4.5: Una e una sola componente attivata

			Non risolte da		
Componente	Num	Speedup	nessuno	$0 - 1/2$	cplex
Riferimento	18	1.02	21	1	5
Ricombinazione	13	2.00	21	5	5
Ortogonalità	19	2.24	11	2	11
Efficacia	14	2.65	19	3	7

Riferimento Parametri: step 4; tutti i criteri di selezione disattivati.

Lo speedup medio di questa configurazione rispetto a CPLEX con cut callback nulla e' praticamente 1.02. Per quanto riguarda le istanze difficili, a van-

taggio dei tagli $0 - 1/2$ vanno le 5 istanze risolte con i tagli rispetto alla sola risolta da CPLEX.

```
>>>>>> base versus ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  18
mean speedup:           1.02 (base / ilo)

Instances solved by neither:
count:                  21

Instances solved in base but not in ilo:
count:                  5
mean time:              380 sec

Instances solved in ilo but not in base:
count:                  1
mean time:              797 sec
```

Ricombinazione Parametri: step 4, recomb; i criteri di selezione disattivati.

La produzione di tagli di ordine superiore al primo è piuttosto efficace; lo speedup medio raddoppia, anche se il numero di istanze che vanno in time-over passa da 1 a 5.

```
>>>>>> rec versus ilo <<<<<<

Ignored 67 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  13
mean speedup:           2.00 (rec / ilo)

Instances solved by neither:
count:                  21

Instances solved in rec but not in ilo:
count:                  5
mean time:              911 sec

Instances solved in ilo but not in rec:
count:                  5
mean time:              444 sec
```

Condizione di ortogonalità Parametri: step 4, max_scal_prod 0.1; gli altri criteri di selezione disattivati.

L'aggiunta della condizione di non parallelismo porta a un incremento delle prestazioni ancora maggiore; inoltre, non ci sono istanze non risolte con i tagli che non lo siano anche senza.

```
>>>>>> ort versus ilo <<<<<<

Ignored 66 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  19
mean speedup:           2.24 (ort / ilo)

Instances solved by neither:
count:                  19

Instances solved in ort but not in ilo:
count:                  7
mean time:              217 sec

Instances solved in ilo but not in ort:
count:                  0
```

Ordinamento per efficacia Parametri: step 4, max_min_eff 0.25, gli altri criteri disattivati.

Lo speedup medio è ancora maggiore; purtroppo ci sono 3 istanze risolte senza tagli ma non con i tagli.

```
>>>>>> eff versus ilo <<<<<<

Ignored 68 instances
TIMEOVER value: 2400 sec
All means are geometric

Instances solved by both solvers:
count:                  14
mean speedup:           2.65 (eff / ilo)

Instances solved by neither:
count:                  19

Instances solved in eff but not in ilo:
count:                  7
mean time:              176 sec

Instances solved in ilo but not in eff:
count:                  3
mean time:              1017 sec
```

4.4 Test su altri tipi di problemi

Durante il lavoro di tesi si è provato ad applicare i tagli $0 - 1/2$ anche ad altri tipi di problemi, come alcune istanze della collezione Miplib3 o alcuni problemi di *Time-tabling*.

In generale si è verificato che se la formulazione ha un numero di variabili più elevato di quello dei vincoli, il separatore $0 - 1/2$ genera tagli con grande difficoltà.

Un'altra caratteristica che può abbattere l'efficacia del separatore è il fatto che i vincoli della formulazione abbiano molti coefficienti non nulli; per generare i tagli, infatti, la formulazione viene indebolita per fare in modo che la matrice dei coefficienti sia la matrice di incidenza di un grafo. In alternativa a questo metodo, il separatore scritto dal Professor Caprara è in grado di generare tagli usando un *tabù search*, ma neanche questa modalità si è dimostrata efficace.

Capitolo 5

GBF - General Branch&Cut Framework

Questo capitolo è stato scritto da me e da Gianni Codato. Come anticipato nella sezione 2.2, durante le nostre tesi di laurea abbiamo lavorato entrambi su algoritmi di Branch&Cut. Avendo notato che stavamo scrivendo programmi per soddisfare in buona parte le stesse necessità, abbiamo deciso di progettare e implementare una base di codice che potesse fare da punto di partenza per le future implementazioni di Branch&Cut realizzate nell'ambito dei progetti del gruppo di Ricerca Operativa. Abbiamo chiamato questo progetto GBF: General Branch&Cut Framework.

Seguono una breve analisi dei requisiti e una spiegazione di come i questi siano stati trasformati in classi compatibili con le API di CPLEX (chiamate Concert Technology). Nel capitolo 6 è riportata anche la documentazione delle classi prodotta a partire dai commenti del codice con il programma Doxygen.

5.1 Analisi dei requisiti

Sono di seguito presentate le funzionalità che abbiamo ritenuto indispensabili per un Branch&Cut da usare in ambito di ricerca. Il framework sarà costituito da un insieme di classi scritte in un linguaggio orientato agli oggetti; alcune saranno complete, anche se comunque estensibili; altre, come per esempio le classi che implementeranno tagli e pools, saranno delle classi base da cui derivare le implementazioni vere e proprie.

I requisiti fondamentali sono quindi proprio la flessibilità e la possibilità di estendere le funzionalità di base.

5.1.1 Gestione dell'ottimizzazione nel complesso

Parametri da riga di comando

- scegliere la taratura di un algoritmo per la soluzione di problemi NP richiede di effettuare moltissimi test a tappeto; per questo è indispensabile poter passare argomenti dalla riga di comando; in questo modo si minimizzano le ricompilazioni e si possono automatizzare i test usando un linguaggio di scripting ad alto livello.

Log-files, statistiche e profiling

- il framework deve prevedere delle strutture dati in cui raccogliere informazioni su cui poter fare analisi statistiche anche durante l'esecuzione stessa;
- le informazioni raccolte e stampate durante l'esecuzione del programma devono poter essere facilmente suddivise in vari files, in modo da poterle gestire in modo più efficace in fase di analisi dei risulati;
- in particolare è importante poter tenere traccia del tempo di calcolo speso in diverse sezioni del codice.

Preprocessing e gestione della formulazione

Spesso è necessario preprocessare la formulazione del problema per renderla più adatta alle tecniche che si stanno sperimentando; esempi di funzionalità implementate nelle nostre applicazioni:

- suddivisione delle variabili in intere e continue;
- manipolazione della funzione obiettivo.

Usabilità/raggiungibilità delle informazioni

È indispensabile che le informazioni di partenza (parametri da riga di comando e formulazione) e quelle raccolte durante l'esecuzione (informazioni statistiche) siano facilmente accessibili alle varie parti del codice. Inoltre, secondo noi, è importante evitare di appesantire troppo la sintassi delle chiamate a funzione con il passaggio di un numero inutilmente alto di parametri, che spesso sono informazioni di tipo globale.

Un software scritto in ambito di ricerca, infatti, viene modificato spesso, e questo si deve poter fare in modo agevole.

5.1.2 Piani di taglio e pool

Il framework deve prevedere l'implementazione simultanea di più separatori, ciascuno con il prorio pool di tagli, se necessario.

La funzionalità principale del pool è l'estrazione dalle sue strutture dati dei tagli violati da un punto frazionario. È poi molto importante che la ricerca dei tagli violati sia veloce.

5.2 Dai requisiti alle classi

Per implementare una base di codice di questo tipo è indispensabile l'uso di un linguaggio per la programmazione orientata agli oggetti.

Gli unici due linguaggi a oggetti con cui ci si può interfacciare a CPLEX sono C++ e Java: per questioni di prestazioni abbiamo scelto il primo.

Nel seguito della sezione si dà una descrizione di massima delle classi implementate per poter soddisfare i requisiti compatibilmente con la famiglia di classi Concert Technology, che costituiscono le API di CPLEX8.

5.2.1 Concert Technology

La libreria di classi che si usa per interfacciarsi a CPLEX si chiama Concert Technology; queste classi modellano da una parte il solver e tutte le sue funzionalità (fra cui le callbacks), dall'altra tutto ciò che compone un problema MIP: variabili, vincoli, funzione obiettivo. L'implementazione della libreria si basa sul paradigma *handle-implementation* allo scopo di semplificare la gestione dei riferimenti agli oggetti e per permettere di implementare anche funzionalità di *lazy-copy*. La libreria prevede inoltre un sistema di notifica automatico, per cui una modifica alla formulazione del problema viene automaticamente propagata a tutti gli oggetti a cui interessa.

Apparentemente, molte delle funzionalità necessarie al framework possono essere implementate direttamente sulle classi della Concert Technology; in realtà la Concert Technology modella classi molto generali, pensate sia per la programmazione lineare che per la programmazione vincolata; sono molto comode, per esempio, per la descrizione dell'istanza da risolvere, ma sono troppo lente se usate in modo intenso – ad esempio per l'implementazione dei tagli e dei pool.

Quindi le classi della Concert Technology sono state usate semplicemente per lo scambio di dati con CPLEX, oltre che per il meccanismo delle callbacks; tutte le elaborazioni di espressioni lineari che abbiamo implementato operano invece su strutture dati più snelle scritte da noi.

I nomi delle classi Concert Technology seguono il pattern `Ilo<Nome>`; un `IloRange` è un vincolo lineare, un `IloNumVar` è una variabile numerica, `IloCplex` è il solver.

La classe `IloCplex` ha numerose sottoclassi, fra cui le callbacks, che consentono di prendere il controllo del Branch&Cut: `IloCplex::CutCallback` è la callback dalla quale si possono inserire tagli dopo la soluzione del lineare del nodo corrente; `IloCplex::SolveCallback` è quella da cui si può pilotare la soluzione del lineare; `BranchCallback` è quella in cui vengono creati i figli del nodo appena risolto , e così via. Per implementare una callback si deriva la classe originale e si inserisce il codice che va eseguito nel metodo `main()`. Si tenga presente che ogni classe callback è istanziata una e una sola volta.

5.2.2 Gestione dell'ottimizzazione

Parametri da riga di comando Per la gestione dei parametri è stata implementata la struct `Arguments`, della quale viene creata una sola istanza. In pratica si tratta di un semplice insieme di variabili che contengono i valori passa-

ti dalla riga di comando; per il parsing vero e proprio è stato usato `argp_parse`, parte di `libc`.

Log-files La gestione dei log-files è attuata per mezzo della classe `OutputLogs`. Di questa classe è definita una sola istanza, che contiene un `ostream` per ogni file di log.

Statistiche Le informazioni raccolte durante l'ottimizzazione sono memorizzate in una istanza della classe `Statistics`; l'unico metodo di questa classe serve a stamparne il contenuto in una forma comoda da leggere.

Profiling La classe `Timers` implementa un insieme di cronometri che possono essere fermati e fatti ripartire, a seconda della sezione di codice da cui si entra o si esce.

Preprocessing e gestione della formulazione Questa parte dei requisiti è soddisfatta dalla classe `Problem`, che contiene le istanze delle classi Concert Technology che modellano il MIP. Qui sono implementate anche le funzioni che preprocessano la fomulazione per l'utilizzo da parte di separatori etc.

Molto importante è l'indicizzazione delle variabili, necessaria a non perdere la corrispondenza fra le strutture Concert Technology e le nostre strutture di tipo C.

Usabilità Per ciascuna delle classi elencate fino a qui è allocata una sola istanza, della quale è disponibile un puntatore globale. In questo modo si minimizzano i parametri da passare ai vari metodi e si evita di duplicare informazioni nelle varie classi.

Anche la variabile `env`, che rappresenta l'ambiente CPLEX in esecuzione e che va passata ad ogni costruttore di oggetti Concert Technology è globale.

5.2.3 Piani di taglio e pool

L'implementazione dei piani di taglio e del pool è la parte più complessa del framework, almeno per quanto riguarda la gerarchia delle classi.

La forma più generale di taglio è data dalla classe virtuale `BaseCut`; le funzionalità previste sono quelle di poter interrogare il taglio per sapere se è violato da un punto frazionario e di poterlo convertire in un IloRange.

Figlia di questa è la classe template `SimpleCut`: implementa un vincolo lineare in cui il tipo dei coefficienti è il parametro del template. La struttura dati è costituita da un vettore di indici e uno di coefficienti, in modo da poter calcolare violazioni, slack, prodotti esterni etc nel modo più veloce possibile. I tagli $0 - 1/2$, per esempio, sono stati implementati derivando una classe da `SimpleCut`.

La forma più generale di pool è data dalla classe virtuale `Pool`: i suoi due metodi consentono di aggiungere un `BaseCut` al pool e di estrarre dal pool un `IloRangeArray` di tagli violati da un punto frazionario.

La presenza di queste interfacce standard consente di usare facilmente più separatori all'interno del Branch&Cut. A chi implementa i separatori è lasciata

la massima libertà, mentre chi li usa non deve preoccuparsi di come sono stati implementati.

Il codice dei separatori va poi chiamato dal metodo `main` della classe `CutCBI`, che naturalmente discende da `CutCallbackI`.

5.2.4 Altre classi a corredo

Per l'effettivo funzionamento del Branch&Cut è stato necessario implementare altre classi.

La classe `DatiNodo` rappresenta un insieme di informazioni che vengono associate a ogni nodo dell'albero di Branch&Cut: la sua profondità, un intero che lo identifica univocamente etc. CPLEX, infatti, prevede di poter associare a ciascun nodo un puntatore a un oggetto di tipo `IloCPlex::NodeData`, in modo da poter associare al nodo tutte le informazioni che possono essere utili alla gestione del Branch&Cut. Le istanze di `DatiNodo` sono assegnate ai nodi al momento della loro creazione da parte dell'istanza di `BrancCBI`, che è una `BranchCallbackI`; possono poi essere raggiunte da tutte le altre callback: per esempio, la `NodeCBI` si occupa di scegliere il prossimo nodo da risolvere.

Nel prossimo capitolo è riportata la documentazione delle classi generata a partire dai commenti dei sorgenti con il programma Doxygen: vi si trovano le descrizioni delle classi e dei loro metodi e alcuni diagrammi ad albero che aiutano a comprendere le relazioni di ereditarietà e specializzazione fra le classi.

Capitolo 6

Classi di GBF

Questo capitolo contiene parte della documentazione prodotta a partire dai sorgenti con il sistema Doxygen. Si tratta di un software che estrae i commenti dal codice C/C++ e li impagina per la stampa o per la pubblicazione su web. La sintassi da usare per i commenti è molto simile a quella del sistema Javadoc che implementa le stesse funzionalità per il linguaggio Java.

Per una introduzione alla struttura del framework si veda il capitolo precedente.

6.1 Arguments Struct Reference

Variables related to command line parameters.

`#include <arguments.h>`

Public Types

- enum **node_sel_t** { **cplex**, **deeper**, **upper** }

  Next-node selection policy.

Public Methods

- ostream & **print** (ostream &os) const

  Prints the value of each parameter.

- **Arguments** (int argc, char *argv[])

Static Public Methods

- error_t **parse_opt** (int key, char *arg, struct argp_state *state)

  Command line parser from libc.

Public Attributes

- string **input_file**

  Usually a .lp file.

- string **output_dir**

  Directory to put log files.

- bool **cplex_only**

  Enable/disable callbacks.

- **node_sel_t node_sel**

  Next node selection strategy.

- IloCplex::MIPEmphasisType **MIPEmphasis**

  See Cplex docs.

- int **MIPInterval**

  See Cplex docs.

- int **MIPDisplay**

  See Cplex docs.

- bool **cplex_cuts**

  Wether to disable or not Cplex's cutting plane families.

- bool **cplex_heur**

  Whether to disable or not cplex's heuristic.

- int **time_limit**

  A time limit in seconds.

- int **sol_limit**

  Maximum number of int sol to be found.

- bool **null_callback**

  Use an empty cut callback.

- int **max_depth**

  Don't cut if deeper than this (1000).

- int **max_call**

  Max number of calls in the same node (excluded root) (1).

- int **max_cuts**

  Max number of cuts generated by the C separator (1000).

- int **root_max_call**

  Max number of calls at root node (5).

- int **step**

  Use the callback every "step" nodes (10).

- bool **recomb**

  Try to generate new cuts recombinig already generated cuts (false).

- bool **check**

  The problem is mixed integer (true).

- double **max_scal_prod**

  Max allowed normalized scalar product between two cuts (0.6).

- double **max_min_eff**

  Upper bound for minimal efficacy (0.2).

- double **max_obj_parall**

  Max allowed normalized scalar product between cuts and obj func (0.2).

- double **max_node_cuts**

  Max number of cuts per node, normalized to the number of constraints (0.3).

- bool **fake_obj**

  Build a fake obj func for sat problems (false).

Static Public Attributes

- char * **doc**

  General info about the application.

- char * **args_doc** = "INPUT_FILE"

  A description of the command line syntax.

- argp_option **options** []

  Every argp_option describes a parameter.

- argp **argp**

  The main argument passed to arp_parse.

Detailed Description

Variables related to command line parameters.

The management of command line parameters is based on Argp, an interface for parsing unix-style argument vectors, part of libc. In case of sintax errors a usage message is displayed.

There is one and only global instance of this struct during execution.

Please read libc's documentation to understand how to add variables to manage parameters needed by your application.

Watch out: the names of command line parameters and the names of their variables are not always the same: use "./exe –usage" to get a list of accepted parameters.

Member Enumeration Documentation

enum Arguments::node_sel_t

Next-node selection policy.

You can force the program to always process the deepest or the least deep among branch tree active nodes. This is implemented by means of class **NodeCBI** (p. 100).

Enumeration values:

cplex Let cplex choose (default).

deeper Always the deepest.

upper Always the least deep.

Member Function Documentation

error_t Arguments::parse_opt (int *key*, char * *arg*, struct argp_state * *state*) [static]

Command line parser from libc.

Fills in the fields, according to descriptions provided in struct argp_option options[].

Member of struct argp argp.

ostream & Arguments::print (ostream & *os*) const

Prints the value of each parameter.

We use this to print a final report about the optimization

Member Data Documentation

struct argp Arguments::argp `[static]`

The main argument passed to arp_parse.
Needed to pass doc, args_doc, options[] and parse_opt to argp_parse()

char * Arguments::args_doc = "INPUT_FILE" `[static]`

A description of the command line syntax.
Member of struct argp argp.

bool Arguments::cplex_cuts

Wether to disable or not Cplex's cutting plane families.
true = cuts are enabled (default);
false = cuts are disabled.

bool Arguments::cplex_heur

Whether to disable or not cplex's heuristic.
true = enabled;
false = disabled.

bool Arguments::cplex_only

Enable/disable callbacks.
true = run with std cplex (no callbacks);
false = otherwise (default).

char * Arguments::doc `[static]`

Initial value:

```
"Branch&Cut Framework for MIP solvers using Cplex8.0"
              "\v"
              "..."
```

General info about the application.

Member of struct argp argp.

double Arguments::max_min_eff

Upper bound for minimal efficacy (0.2).

If negative, cuts' efficacy is ignored.

bool Arguments::null_callback

Use an empty cut callback.

Using a cut callback forces cplex to avoid applying irreversible transformations to the model before branch & cut. These transformations can make a huge difference in optimization time; so, if you are testing a cutting plane family set this to true.

Default is false.

struct argp_option Arguments::options[] `[static]`

Every argp_option describes a parameter.

Member of struct argp argp.

int Arguments::sol_limit

Maximum number of int sol to be found.

Typically used for sat problems with a fake objective function

int Arguments::time_limit

A time limit in seconds.

A negative value means no limits (default)

The documentation for this struct was generated from the following files:

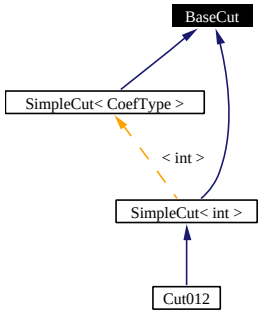
- **arguments.cpp**
- **arguments.h**

6.2 BaseCut Class Reference

Very general, abstract base class for cuts.

`#include <cut.h>`

Inheritance diagram for BaseCut:



Public Methods

- virtual ~**BaseCut** ()

  Destructor.

- virtual BaseCut & **operator=** (BaseCut const &c)

  Assignement operator.

- virtual IloRange **toIloRange** ()=0

  Conversion to IloRange.

- virtual bool **isViolated** (IloNumArray sol)=0

  Is this cut violated by solution sol?

- virtual void **print** (ostream &out)

  Used by the output operator.

Detailed Description

Very general, abstract base class for cuts.

At this level, we assume that a cut:

- is able to convert itself to the IloRange format
- can tell wether it is violated or not by a given solution.

We did not make any assumption on the implementation to let you choose the most efficient one for your application.

Member Function Documentation

virtual bool BaseCut::isViolated (IloNumArray *sol*) [pure virtual]

Is this cut violated by solution sol?

Parameters:

sol The solution is in IloNumArray format, which is the format used by CPLEX to return solutions.

Implemented in **SimpleCut< CoefType >** (p. 115), and **SimpleCut< int >** (p. 115).

virtual IloRange BaseCut::toIloRange () [pure virtual]

Conversion to IloRange.

We suggest you to implement a cache mechanism; it is easy, since IloRanges are handles.

Implemented in **SimpleCut< CoefType >** (p. 116), and **SimpleCut< int >** (p. 116).

The documentation for this class was generated from the following file:

- **cut.h**

6.3 BranchCBI Class Reference

A BranchCallbackI.

`#include <callbacks.h>`

Public Methods

- void **main** ()
- IloCplex::CallbackI * **duplicateCallback** () const

Detailed Description

A BranchCallbackI.

This callback lets you:

- override Cplex's branching algorithm;
- build a **DatiNodo** (p. 94) object and stick it to its node (more details in class **DatiNodo** (p. 94) documentation).

The documentation for this class was generated from the following files:

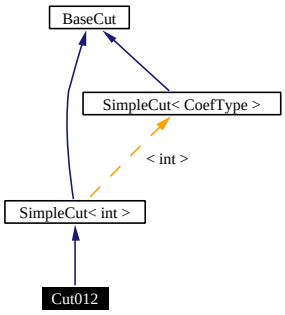
- **callbacks.h**
- **callbacks.cpp**

6.4 Cut012 Class Reference

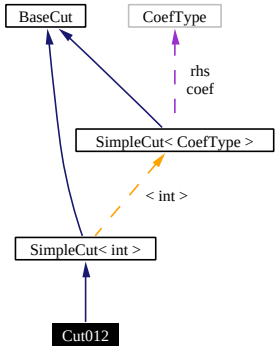
0-1/2 cuts.

`#include <cut012.h>`

Inheritance diagram for Cut012:



Collaboration diagram for Cut012:



Public Methods

- **Cut012** (int _cnt, int *_ind, int *_coef, Sense_t _sense, int _rhs, double *xstar=0)
- **Cut012** (Cut012 const &a)
- ~**Cut012** ()
- Cut012 & **operator=** (Cut012 const &a)
- void **calcEff** (IloNumArray xstar)

Updates cut's efficacy with respect to xstar.

- void **calcEff** (double const *xstar)

  Updates cut's efficacy with respect to xstar.

- double **operator** * (const Cut012 &b)

  Returns the scalar product of two cuts.

Static Public Methods

- void **setObj** ()

  Fills member static double objec[].

Public Attributes

- double **eff**

  Euclidean distance between the plane and xstar.

- double **inc**

  Normalized scalar product of the plane's generator and the obj func.

- double **nor**

  Euclidean norm of the plane's generator.

Static Public Attributes

- int **tot**

  Total number of cuts (used to name cuts).

- double * **object**

  Coefficients of the obj func.

- double **normaobj**

  Norm of the obj func's vector.

Protected Methods

- void **calcInc** ()

  Calculates how much this cut is parallel to the obj func's vector.

- void **calcNor** ()

  Calculates the norm of the plane's generator.

Detailed Description

0-1/2 cuts.

This class adds to **SimpleCut** (p. 112) the notions of efficacy (i.e. euclidean distance from the fractional point) and inclination (i.e. normalized scalar product between plane's generator and objective function). There is also an operator to compute the normalized scalar product of two cuts, i.e. the cosinus of the angle defined by their generators, or their mutual parallelism.

The documentation for this class was generated from the following files:

- **cut012.h**
- **cut012.cpp**

6.5 CutCBI Class Reference

A CutCallbackI.

`#include <cutcb.h>`

Collaboration diagram for CutCBI:

Public Methods

- IloCplex::CallbackI * **duplicateCallback** () const
- virtual void **main** ()

  The entry point of the whole cut callback.

- void **main012** ()

  The entry point of the 0-1/2 cuts part of the callback.

- **CutCBI** ()

Protected Methods

- void **resizeUpRows012** (int num_rows)

  Minimizes the memory used for the C representation of the problem.

- void **resizeUpNZs012** (int num_nzs)

  Doubles the amount of memory allocated for the C formulation (non-zeros).

- void **addRangeArray012** (IloRangeArray &con)

  Adds an IloRangeArray to the C formulation passed to sep_012_cut.

- int **generate012** ()

  Generation and translation of new cuts.

- int **select012** ()

  Extracts the best cuts from the pool.

- bool **checkRange012** (IloRange &a)

  Checks for the integrality of an IloRange's coefficients and variables.

Protected Attributes

- IloNumArray **sol**

  The current solution.

- **Pool** ∗ **pool**

  A pool.

- int **mr**

  Number of rows in the ILP matrix.

- int **mc**

  Number of columns in the ILP matrix.

- int **mnz**

  Number of nonzero's in the ILP matrix.

- int ∗ **mtbeg**

  Starting position of each row in arrays mtind and mtval.

- int ∗ **mtcnt**

  Number of entries of each row in arrays mtind and mtval.

- int ∗ **mtind**

  Column indices of the nonzero entries of the ILP matrix.

- int * **mtval**

  Values of the nonzero entries of the ILP matrix.

- int * **vlb**

  Lower bounds on the variables.

- int * **vub**

  Upper bounds on the variables.

- int * **mrhs**

  Right hand sides of the constraints.

- char * **msense**

  Senses of the constraints: 'L', 'G' or 'E'.

- double * **xstar**

  Current optimal solution of the LP relaxation.

- short int **aggressive**

  Flag specifying whether as many cuts as possible are required on output (TRUE) or not (FALSE).

- int **cnum**

  Number of violated 0-1/2 cuts identified by the procedure.

- int **cnzcnt**

  Overall number of nonzero's in the cuts.

- int * **cbeg**

  Starting position of each cut in arrays cind and cval.

- int * **ccnt**

  Number of entries of each cut in arrays cind and cval.

- int * **cind**

  Column indices of the nonzero entries of the cuts.

- int * **cval**

  Values of the nonzero entries of the cuts.

- int * **crhs**

  Right hand sides of the cuts.

- char * **csense**

  Senses of the cuts: 'L', 'G' or 'E'.

- int **num_int_vars_012**

  Number of integer variables.

- IloNumVarArray **int_vars_012**

  A handle pointing to the vector in class **Problem** (p. 109).

- int **depth_012**

  The depth of the current node.

- int **row_capacity_012**

  Amount of space allocated for the model representation to be passed to sep_-012_cut.

- int **nz_capacity_012**

  Amount of space allocated for the model representation to be passed to sep_-012_cut.

- int **last_node_012**

  NodeId at last call.

- int **added_cuts_012**

  Number of new cuts added douring last call.

- int **glob_cuts_012**

  Num of cuts added globally.

- int **total_node_cuts_012**

  Total number of cuts applied to the current node.

- int **conta_call_012**

  The number of times the callback has been called on the current node.

- int **this_node_012**

  NodeId at this call.

Detailed Description

A CutCallbackI.

To make the code reusable – think about implementing a cut callback which adds two different cut families – we chose to put all the code outside method **main**() (p. 88).

So far, we merged different cut callbacks just cutting and pasting source codes. A more mature implementation should be based upon multiple inheritance.

Member Function Documentation

int CutCBI::generate012 () [protected]

Generation and translation of new cuts.

Calls sep_012_cut, converts generated cuts in **Cut012** (p. 85) objects and adds them to the pool.

void CutCBI::main012 ()

The entry point of the 0-1/2 cuts part of the callback.

This function relies on all other ∗012 members, and is called by member main.

void CutCBI::resizeUpNZs012 (int *num_nzs*) `[protected]`

Doubles the amount of memory allocated for the C formulation (non-zeros).

Called before adding an IloRangeArray of cuts to the C formulation.

void CutCBI::resizeUpRows012 (int *num_rows*) `[protected]`

Minimizes the memory used for the C representation of the problem.

This function is called before adding an IloRangeArray of cuts to the C formulation (rows).

int CutCBI::select012 () `[protected]`

Extracts the best cuts from the pool.

Asks the pool (of type **Pool012** (p. 106)) to retrieve the best collection of cuts to separate current solution. See **Pool012::getViolatedCuts** (p. 106), where the actual selection is done.

The documentation for this class was generated from the following files:

- **cutcb.h**
- **cutcb.cpp**

6.6 DatiNodo Class Reference

Node informations: size, depth etc.

`#include <nodedata.h>`

Public Methods

- **DatiNodo** ()
- ~**DatiNodo** ()
- **DatiNodo** (DatiNodo &padre)

  This is not a copy constructor.

Public Attributes

- int **node_cons**

  The number of constraints in the node.

- int **node_id**

  The node identifier.

- int **depth**

  The node's depth in the branch tree.

- IloExpr **lhs**

  An IloExpression representing the branch sequence that built the node.

- IloInt **rhs**

  The right hand side of the cut representing the branch sequence.

- int **node_cuts_012**

  The number of 0-1/2 cuts locally added down to this node.

Detailed Description

Node informations: size, depth etc.

An object of this class is bound to every node in the branch callback; it stores mainly depth and size informations; there is also an IloExpression which can be used to record the branch sequence that leaded to this node.

Constructor & Destructor Documentation

DatiNodo::DatiNodo (DatiNodo & *padre*)

This is not a copy constructor.

This constructor is used to initialize the object related to a node starting from its father's one.

The documentation for this class was generated from the following files:

- **nodedata.h**
- **nodedata.cpp**

6.7 HeurCBI Class Reference

A HeuristicCallbackI.

`#include <callbacks.h>`

Public Methods

- void **main** ()
- IloCplex::CallbackI * **duplicateCallback** () const

Detailed Description

A HeuristicCallbackI.

Here you can implement your heuristic.

The documentation for this class was generated from the following files:

- **callbacks.h**
- **callbacks.cpp**

6.8 IncumCBI Class Reference

An IncumbentCallbackI.

`#include <callbacks.h>`

Public Methods

- void **main** ()
- IloCplex::CallbackI * **duplicateCallback** () const

Detailed Description

An IncumbentCallbackI.

This one is called each time an integer feasible solution is found; you can accept or reject it, according to some problem specific (non-linear) criteria.

The documentation for this class was generated from the following files:

- **callbacks.h**
- **callbacks.cpp**

6.9 std::less< Cut012 * > Struct Template Reference

Specialization of the binary operator less.

`#include <cut012.h>`

Public Methods

- bool **operator()** (const **Cut012** *a, const **Cut012** *b) const

Detailed Description

template<> struct std::less< Cut012 * >

Specialization of the binary operator less.

Please, note that actually this is a "more".

The documentation for this struct was generated from the following file:

- **cut012.h**

6.10 MIPCBI Class Reference

A MIPCallbackI.

`#include <callbacks.h>`

Public Methods

- void **main** ()
- IloCplex::CallbackI * **duplicateCallback** () const

Detailed Description

A MIPCallbackI.

We do not really know how to use this one.

The documentation for this class was generated from the following files:

- **callbacks.h**
- **callbacks.cpp**

6.11 NodeCBI Class Reference

A NodeCallbackI.

`#include <callbacks.h>`

Public Methods

- void **main** ()
- IloCplex::CallbackI * **duplicateCallback** () const

Detailed Description

A NodeCallbackI.

This is were the choice for the next node takes place.

In general, this class gives you access to the queue of pending nodes; we use it also to get a measure of how much of the tree has not yet been explored.

The documentation for this class was generated from the following files:

- **callbacks.h**
- **callbacks.cpp**

6.12 OutputLogs Struct Reference

A struct of ofstreams.

`#include <output.h>`

Public Methods

- **OutputLogs** ()

  The constructor.

- ~**OutputLogs** ()
- void **flushAll** ()

  Flushes all the files at once.

Public Attributes

- ofstream **solver**

  The usual node log.

- ofstream **warnings**

  Informations about recoverable problems.

- ofstream **errors**

  Not recoverable problems.

- ofstream **solutions**

  Solutions found douring the optimization.

- ofstream **perNode**

  Per node infos.

- ofstream **perCallback**

  Per callback infos.

Detailed Description

A struct of ofstreams.

These ofstreams are provided to send log infos to several different files

Constructor & Destructor Documentation

ILOSTLBEGIN OutputLogs::OutputLogs ()

The constructor.

Opens the files and appends a header to each one, in order to separate subsequent runs.

The output directory is read from global **Arguments** (p. 76) ∗ args.

perNode and perCallback are initialized accordingly to the syntax of the plotting program used.

Member Function Documentation

void OutputLogs::flushAll ()

Flushes all the files at once.

You should flush all the files no more often than once per node; to write a newline use

```
<< \n
```

and not

```
<< endl
```

Member Data Documentation

ofstream OutputLogs::perCallback

Per callback infos.

We output to this stream gnuplot or scilab formatted data

ofstream OutputLogs::perNode

Per node infos.

We output to this stream gnuplot or scilab formatted data

The documentation for this struct was generated from the following files:

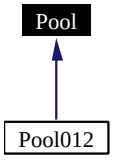
- **output.h**
- **output.cpp**

6.13 Pool Class Reference

Base class for pools.

`#include <pool.h>`

Inheritance diagram for Pool:



Public Methods

- virtual IloRangeArray **getViolatedCuts** (IloNumArray sol)=0

  Returns an IloRangeArray of cuts violated by solution sol.

- virtual void **addCut** (**BaseCut** *cut)=0

  Intserts **BaseCut** (p. 82) *cut into the pool.*

- virtual ~**Pool** ()

  The destructor (empty).

Detailed Description

Base class for pools.

This virtual class provides an interface for the most general pool: there are only two virtual methods, representing the actions of pushing cuts into the pool and getting the cuts violated by a solution. Added cuts must be BaseCuts.

Member Function Documentation

virtual void Pool::addCut (BaseCut * *cut*) `[pure virtual]`

Intserts **BaseCut** (p. 82) cut into the pool.

The argument is a pointer; in this way efficiency is improved.

Implemented in **Pool012** (p. 107).

virtual IloRangeArray Pool::getViolatedCuts (IloNumArray *sol*) `[pure virtual]`

Returns an IloRangeArray of cuts violated by solution sol.

The types of the parameters are Concert Technology classes: the assumption is you obtained the solution via CutCallback::getValue() and are just going to add the cuts calling CutCallbackI::add() or CutCallbackI::addLocal() methods.

Implemented in **Pool012** (p. 106).

The documentation for this class was generated from the following files:

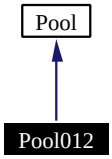
- **pool.h**
- **pool.cpp**

6.14 Pool012 Class Reference

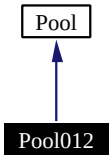
A pool of 0-1/2 cuts.

`#include <pool012.h>`

Inheritance diagram for Pool012:



Collaboration diagram for Pool012:



Public Methods

- **Pool012** (int _maxpool)
- void **addCut** (**BaseCut** *cut)

  Intserts **BaseCut** (p. 82) *cut into the pool.*

- IloRangeArray **getViolatedCuts** (IloNumArray sol)

  Implements the virtual method, to be consistent with the interface: calls getViolatedCuts(sol, cuts.size()).

- IloRangeArray **getViolatedCuts** (IloNumArray sol, int max_cuts)

  Overloads the virtual method.

- ~**Pool012** ()

  Destructor.

Protected Attributes

- int **maxpool**
- vector< **Cut012** * > **cuts**

  The actual data structure.

Detailed Description

A pool of 0-1/2 cuts.

This pool is size limited; geometric cut properties are taken into account when violated cuts are extracted.

Constructor & Destructor Documentation

Pool012::~Pool012 ()

Destructor.

Explicitly deletes each cut, since they are stored as pointers.

Member Function Documentation

void Pool012::addCut (BaseCut * *cut*) [virtual]

Intserts **BaseCut** (p. 82) cut into the pool.

The argument is a pointer; in this way efficiency is improved.

Implements **Pool** (p. 105).

IloRangeArray Pool012::getViolatedCuts (IloNumArray *sol*, int *max_cuts*)

Overloads the virtual method.

Here is implemented all the selection of violated cuts according to their geometric properties: geom depth, mutual parallelism, parallelism with the obj func.

If the pool is too large, cuts in excess are deleted after the pool has been sorted (see **Arguments::max_node_cuts** (p. 78)).

Parameters:

max_cuts maximum number of cuts to be returned in the IloRangeArray, not to oversize the formulation.

Member Data Documentation

vector<Cut012 *> Pool012::cuts `[protected]`

The actual data structure.

We store cuts as pointers to have the fastest sorting.

The documentation for this class was generated from the following files:

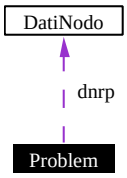
- **pool012.h**
- **pool012.cpp**

6.15 Problem Class Reference

Encapsulates and initializes the objects that describe the model.

`#include <problem.h>`

Collaboration diagram for Problem:



Public Methods

- **Problem** ()

Public Attributes

- IloModel **mod**

  The model to be optimized.

- IloCplex **alg**

  The algorithm to be used.

- IloObjective **obj**

  The objective function.

- IloNumVarArray **vars**

  The handles of the variables.

- IloNumVarArray **int_vars**

  The handles of the integer variables.

- IloNumVarArray **cont_vars**

The handles of the continuous variables.

- IloRangeArray **cons**

  The constraints of the model.

- **DatiNodo** $*$ **dnrp**

  The **VarData** (p. 126) *instance related to the root node.*

Protected Methods

- void **processVars** ()

  Initializes int_vars and cont_vars and constructs **VarData** (p. 126) *objects.*

- void **processCons** ()

  Clears up cons; checks that all constraints have one and only bound.

- void **fakeObjSat** ()

  Builds a not-null obj func for SAT problems.

Detailed Description

Encapsulates and initializes the objects that describe the model.

Loads the model from **Arguments::input_file** (p. 76), processes vars separating integer from continuous ones and provides a method to build a fake obj func for sat problems.

prob (see **globals.h**) is a global pointer to the one and only instance of this class.

Member Function Documentation

void Problem::fakeObjSat () [protected]

Builds a not-null obj func for SAT problems.

The obj func is the sum of (a subset of) cons' linear expressions which has a minimum support of one fifth of vars' size.

Member Data Documentation

DatiNodo* Problem::dnrp

The **VarData** (p. 126) instance related to the root node.

It is stored here since there is no branch callback or similar to initialize the root node.

The documentation for this class was generated from the following files:

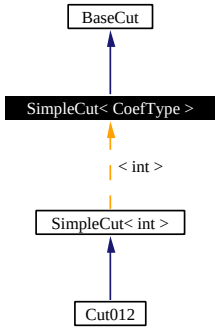
- **problem.h**
- **problem.cpp**

6.16 SimpleCut< CoefType > Class Template Reference

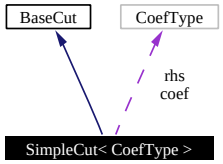
Linear cut template.

`#include <cut.h>`

Inheritance diagram for SimpleCut:



Collaboration diagram for SimpleCut< CoefType >:



Public Types

- enum **Sense_t** { **lessEq**, **equal**, **greaterEq** }

  The sense of the inequality.

Public Methods

- **SimpleCut** (int _cnt, int *_ind, CoefType *_coef, **Sense_t** _sense, CoefType _rhs)
- **SimpleCut** (SimpleCut const &c)

This class has value semantics.

- virtual ~**SimpleCut** ()
- virtual SimpleCut & **operator=** (const SimpleCut &a)

  This class has value semantics.

- virtual IloRange **toIloRange** ()

  Conversion to IloRange.

- virtual bool **isViolated** (IloNumArray sol)

  Is this cut violated by solution sol?

- virtual void **print** (ostream &out)

  Cuts are printed in ".lp" format.

Static Public Methods

- void **setEpsFeas** (IloNum _eps_feas)

  Sets the _eps_feas tolerance.

- void **setVars** (IloNumVarArray _vars)

  Initializes _vars.

Protected Attributes

- int **cnt**

  The number of non-zero coefficients in the lhs.

- int * **ind**

  The array of the indexes of the variables.

- CoefType * **coef**

  The array of the values of the coefficients.

- **Sense_t sense**

  The sense of the inequality.

- CoefType **rhs**

  The right hand side.

- int **cut_id**

  Every SimpleCut is identified by this integer index.

- IloRange **cache**

  The IloRange representation of the cut.

Static Protected Attributes

- int **num_cuts** = 0

  The total number of SimpleCut objects constructed so far.

- IloNum **eps_feas** = 0.0

  The feasibility tolerance.

- IloNumVarArray **vars** = 0

  The array of variables, needed to convert the cut to IloRange.

Detailed Description

template<typename CoefType> class SimpleCut< CoefType >

Linear cut template.

This template models a linear constraint; the type used for the coefficients is a parameter, not to slow down applications based on cuts with integer coefficients. Since the heavily object oriented implementation of IloRanges is too slow to be used in a pool based separator, this one is based on a more traditional array of

non-zero coefficients matched by an array of column coefficients, both allocated in dynamic memory.

If you are going to derive from this class, pay attention to memory management.

Constructor & Destructor Documentation

template<typename CoefType> SimpleCut< CoefType >::SimpleCut (SimpleCut< CoefType > const & *c*)

This class has value semantics.

Please, note that IloRange has reference semantics instead.

Member Function Documentation

template<typename CoefType> bool SimpleCut< CoefType >::isViolated (IloNumArray *sol*) `[virtual]`

Is this cut violated by solution sol?

The check is based on a tolerance value stored in member eps_feas.

Implements **BaseCut** (p. 83).

template<typename CoefType> SimpleCut< CoefType > & SimpleCut< CoefType >::operator= (const SimpleCut< CoefType > & *a*) `[virtual]`

This class has value semantics.

Please, note that IloRange has reference semantics, instead.

template<typename CoefType> void SimpleCut< CoefType >::print (ostream & *out*) `[virtual]`

Cuts are printed in ".lp" format.

We always use this format to import models in Cplex.

Reimplemented from **BaseCut** (p. 82).

template<typename CoefType> IloRange SimpleCut< CoefType >::toIloRange () `[virtual]`

Conversion to IloRange.

We suggest you to implement a cache mechanism; it is easy, since IloRanges are handles.

Implements **BaseCut** (p. 83).

Member Data Documentation

template<typename CoefType> IloRange SimpleCut< CoefType >::cache `[protected]`

The IloRange representation of the cut.

It is inizialized during the first execution of **toIloRange**() (p. 116), and deleted when the object is destructed. In this way we build it if and only if it is usefull, and only once.

The documentation for this class was generated from the following file:

- **cut.h**

6.17 SolveCBI Class Reference

A SolveCallbackI.

`#include <callbacks.h>`

Public Methods

- void **main** ()
- IloCplex::CallbackI * **duplicateCallback** () const

  < *Updates the moving average of the estimated number of nodes in the tree.*

Detailed Description

A SolveCallbackI.

From this callback you can control the lp optimization.

The documentation for this class was generated from the following files:

- **callbacks.h**
- **callbacks.cpp**

6.18 Statistics Struct Reference

A data structure of infos collected during execution.

`#include <statistics.h>`

Public Methods

- **Statistics** ()

  Assignes Initial values to vars.

- ostream & **print** (ostream &os) const

  Prints a complete report.

- ostream & **shortPrint012** (ostream &os) const

  Prints most important infos in one line.

Public Attributes

- int **num_vars**

  The number of vars.

- int **num_int_vars**

  The number of integers vars.

- int **num_cont_vars**

  The number of continuous vars.

- int **num_cons**

  The number of constraints.

- int **initial_cuts**

  The number of cons read from file.

- int **cplex_added_cuts**

The number of cuts added by Cplex MIP solver.

- int **added_cuts**

  The number of cuts added by our callbacks.

- int **pool_access**

  The number of pool queries.

- int **pool_hits**

  The number of successfull pool queries.

- int **int_sol**

  The number of integer solutions found.

- int **cb_calls**

  The number of callback calls.

- int **num_nodes**

  The number of nodes visited.

- int **num_nodes_left**

  The number of nodes left.

- double **obj**

  The value of the objective function.

- double **comb**

  The fraction of the branch tree left to explore.

- double **best_bound**

  The best bound's value.

- double **best_int_obj**

  The cost of the best integer solution.

- int **cb_calls_012**

  The number of 0-1/2 cut callback calls.

- int **pool_access_012**

  The number of 0-1/2 cut cb pool queries.

- int **pool_hits_012**

  The number of 0-1/2 cut cb pool hits.

- int **added_cuts_glob_012**

  The number of 0-1/2 cuts added globally.

- int **added_cuts_loc_012**

  The number of 0-1/2 cuts added locally.

Detailed Description

A data structure of infos collected during execution.

The documentation for this struct was generated from the following files:

- **statistics.h**
- **statistics.cpp**

6.19 Timers Class Reference

This class implements profiling utilities.

`#include <timers.h>`

Public Types

- enum **codeSection** { **pool_search**, **solve**, **cut_cb**, **solve_cb**, **branch_cb**, **node_cb**, **num_sections** }

  Has a member for each code section to monitor.

Public Methods

- **Timers** ()

  The constructor; asserts initial conditions.

- void **start** (**codeSection** _start)

  Starts the specified timer.

- void **stop** (**codeSection** _stop)

  Stops the specified timer.

- void **stopStart** (**codeSection** _stop, **codeSection** _start)
- double **getTime** (**Timers::codeSection** clock)

  Gets the amount of time spent in the specified codeSection.

- double **getBeCTime** ()

  Gets the time spent doing branch & cut.

- ostream & **print** (ostream &) const

  Prints a profile of the process up to this point.

Static Public Methods

- double **totalTime** ()

  Gets the total cpu time of the process.

Static Public Attributes

- char const ∗const **names** [ ]

  Names needed to print profile informations.

Protected Attributes

- double **acc_times** [num_sections]
- clock_t **last_start** [num_sections]

  Documented in file timer.cpp.

Detailed Description

This class implements profiling utilities.

As the name suggests, an object of this class is a set of clocks: a global one and one for each section of code you want to monitor.

Clocks must be manually started and stopped when the program flow leaves a code section to enter another one.

To add a clock, you have to add its name both to member enum codeSection and to member string array names.

For future developmens: maybe this class' design could be more object oriented.

It is not based on class IloTimer, which seems to be broken.

Member Enumeration Documentation

enum Timers::codeSection

Has a member for each code section to monitor.

Add your members before the last one (num_sections), since it is used to count the number of sections.

Enumeration values:

pool_search Time spent looking for violated cuts in the pool.

solve Time spent outside callbacks.

cut_cb Time spent in CutCallbackI's function methods.

solve_cb Time spent solving lp relaxations.

branch_cb Time spent in BranchCallbackI's function methods.

node_cb Time spent in NodeCallbackI's function methods.

num_sections This is not a timer!! It serves only as a counter.

Add your code sections before this one, and remember to add a proper element to string array **Timers::names** (p. 124).

Member Function Documentation

double Timers::getTime (Timers::codeSection *clock*)

Gets the amount of time spent in the specified codeSection.

You can query active timers without consequences.

ostream & Timers::print (ostream &) const

Prints a profile of the process up to this point.

It can be called at any moment, without worrying: no clock is stopped.

void Timers::start (codeSection *_start*)

Starts the specified timer.

See method **Timers::stop** (p. 124)

void Timers::stop (codeSection *_stop*)

Stops the specified timer.

When a timer is stopped, the elapsed time, in seconds, is accumulated in a double.

void Timers::stopStart (codeSection *_stop*, codeSection *_start*) `[inline]`

Parameters:

_start == stop(_stop); start(_start). This is very usefull if you are profiling a program made of a main code section periodically interrupted by other sections.

double Timers::totalTime () `[static]`

Gets the total cpu time of the process.

Implemented by means of libc's getrusage(). This assures that time spent building the model, preprocessing it,etc is taken into account.

Member Data Documentation

char const *const Timers::names `[static]`

Initial value:

```
{
        "Pool_Search",
        "Solve",
        "CutCB",
        "SolveCB",
        "BranchCB",
        "NodeCB",
}
```

Names needed to print profile informations.

Please note that names and code sections are matched if and only if this array and enum codeSection are ordered the same way.

The documentation for this class was generated from the following files:

- **timers.h**
- **timers.cpp**

6.20 VarData Class Reference

An instance of this data structure is bound to each variable.

`#include <problem.h>`

Public Methods

- **VarData** (int _index, int _int_index, int _cont_index=-1)

Public Attributes

- int **index**

  The variable's index in **Problem::vars** (p. 109).

- int **int_index**

  The variable's index in **Problem::int_vars** (p. 109)*; -1 if the variable is continue.*

- int **cont_index**

  The variable's index in **Problem::cont_vars** (p. 109)*; -1 if the variable is integer.*

Detailed Description

An instance of this data structure is bound to each variable.

There are only index informations so far, to deal with C oriented linear expressions.

The documentation for this class was generated from the following file:

- **problem.h**

Conclusione

Ciò che si è esposto e suo significato Questo lavoro di tesi ha previsto lo studio della teoria dei tagli $0-1/2$ e del funzionamento delle librerie C++ Ilog CPLEX. Sulla base di queste e di una implementazione C del separatore per i tagli $0-1/2$ è stato implementato un algoritmo Branch&Cut per verificare l'utilità dei tagli $0-1/2$.

I tagli $0-1/2$ si sono dimostrati efficaci per le istanze di problemi SAT, a patto di aggiungerne un numero limitato alla formulazione e di selezionarli sulla base di caratteristiche geometriche come profondità e parallelismo.

L'implementazione finale dell'algoritmo si basa su GBF, una struttura di classi C++ sviluppata in collaborazione con il collega Gianni Codato per semplificare e standardizzare lo sviluppo di algoritmi Branch&Cut in ambito di ricerca.

Analisi comparativa dei risultati I risultati ottenuti sono sicuramente interessanti; dimostrano che i tagli $0-1/2$ sono utili per una classe di problemi importanti e che in generale un algoritmo Branch&Cut può essere migliorato curando la selezione dei tagli aggiunti alla formulazione.

Limitazioni cui sono soggetti i risultati presentati La taratura trovata per i vari parametri del sparatore non si può ritenere definitiva: i valori sono stati trovati in modo empirico su una collezione di problemi limitata.

Possibili applicazioni I criteri di selezione di tagli qui presentati possono essere applicati a tutti gli algoritmi Branch&Cut.

I tagli $0-1/2$, invece, si sono dimostrati efficaci solo per istanze IP in cui la matrice dei vincoli ha coefficienti in $\{0,1\}$. Per quanto riguarda i problemi MIP, il separatore $0-1/2$ può essere usato sulla parte intera del modello, quella che si ottiene tenendo solo le variabili intere e i vincoli il cui supporto è contenuto nell'insieme delle variabili intere.

Bibliografia

[1] M. Fischetti, "Lezioni di Ricerca Operativa", Edizioni Libreria Progetto Padova, 1995.

[2] A. Caprara, "Algoritmi basati su rilassamenti di programmazione lineare per problemi di ottimizzazione combinatoria", Tesi di Dottorato di Ricerca in Ingegneria dei Sistemi, Bologna, Febbraio 1996

[3] DIMACS Challenge Web Page: `http://mat.gsia.cmu.edu/challenge.html`

[4] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, R. Wunderling, "MIP: Theory and practice – closing the gap", `http://www.ilog.com/products/optimization/tech/research/mip.pdf`

[5] E. Balas, S. Ceria, G. Cornuéiols, "Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework", Management Science, Vol. 42, No. 9, September 1996