Tecniche di Decomposizione per Programmazione Lineare Intera (Mista)

Domenico Salvagnin

2011-06-12

1 Introduzione

Dato un problema di programmazione lineare intera (mista), non è sempre possibile (o conveniente) risolvere tale problema risolvendo a scatola chiusa la sua formulazione più "naturale". Infatti tale formulazione potrebbe essere:

- troppo grande per poter essere risolta con un metodo esatto
- troppo debole per garantire la risoluzione in tempi accettabili

Una possibile soluzione a tali problemi consiste nel ricorrere a tecniche di decomposizione. L'impiego di tali tecniche è giustificato da

- i modelli decomposti sono di dimensione sensibilmente più piccola. Questo è ovviamente vantaggioso data la complessità esponenziale degli algoritmi MIP.
- la decomposizione permette di sfruttare la sparsità della matrice dei vincoli, ed in particolare una sua eventuale struttura "quasi" diagonale a blocchi. In generale, eventualmente dopo una permutazione delle righe e delle colonne del problema, la matrice A dei vincoli di un problema MIP ha una struttura *arrow-head* come in Figura 1(a), con una parte diagonale a blocchi e un insieme di vincoli/variabili "complicanti".
- la decomposizione permette di sfruttare una sottostruttura specifica del problema, per la quale esistono algoritmi ad-hoc efficienti (polinomiali o molto veloci in pratica, come ad esempio una struttura knapsack).

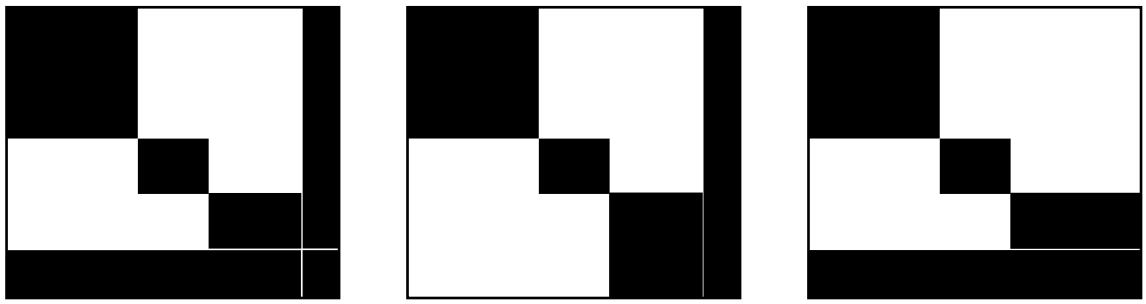
2 Decomposizione di Benders

La decomposizione di Benders si applica quando il problema si semplifica parecchio fissando un sottoinsieme di variabili (un caso particolare è la presenza di matrici di vincoli con struttura come in Figura 1(b)).

Vediamo l'applicazione di tale tecnica nel suo contesto più classico, cioè la risoluzione di problemi di programmazione lineare intera mista (tale classe di problemi rientra banalmente nella classe appena descritta, in quanto dopo aver fissato le variabili intere quello che resta è un semplice problema di programmazione lineare).

Sia dato un problema di programmazione lineare intera mista

$$\begin{aligned} \min\, & c^T x + d^T y \\ & Ax + By \geq b \\ & x \text{ intero} \\ & y \geq 0 \end{aligned}$$



(a) (b) (c)

Figure 1: Possibili strutture della matrice A.

che assumiamo avere ottimo finito o essere impossibile. Introduciamo una variabile continua libera η che tenga conto del contributo delle variabili continue y nella funzione obiettivo, ottenendo:

$$\begin{aligned} \min c^T x + \eta \\ Ax + By \geq b \\ \eta \geq d^T y \\ x \text{ intero} \\ y \geq 0 \end{aligned}$$

e consideriamo la proiezione della regione ammissibile di tale problema nello spazio delle variabili (x, η). Attraverso il lemma di Farkas è possibile dimostrare che una descrizione esplicita di tale proiezione è data da

$$\min c^T x + \eta \tag{2.1}$$
$$\eta \geq u_i^T (b - Ax) \quad \forall u_i \in U \tag{2.2}$$
$$v_i^T (b - Ax) \leq 0 \quad \forall v_i \in V \tag{2.3}$$
$$x \text{ intero} \tag{2.4}$$

dove U e V sono gli insiemi dei vertici e dei raggi estremi (rispettivamente) del seguente poliedro

$$P_D = \{\pi : \pi^T B \leq d^T, \pi \geq 0\}$$

Il sistema (2.1)-(2.4) è una riformulazione equivalente del sistema di partenza, quindi il valore delle rispettive soluzioni ottime coincide. I vincoli (2.2) prendono il nome di tagli di ottimalità, mentre i vincoli (2.3) prendono il nome di tagli di ammissibilità.

Purtroppo la riformulazione (2.1)-(2.4) è caratterizzata da un numero esponenziale di vincoli (uno per ogni vertice e raggio estremo di P_D) e quindi non è pensabile poterla scrivere esplicitamente per poi risolverla a scatola chiusa con un algoritmo branch-and-cut. Si usa invece un algoritmo iterativo dovuto a Benders in cui si considera inizialmente un sottoinsieme dei vincoli che definiscono il problema, e vincoli ulteriori sono aggiunti nel corso dell'algoritmo man mano che risultano violati.

In particolare, vengono eseguiti i seguenti passi:

Passo 1 Si considerino dei sottoinsiemi $U_r \subseteq U$ e $V_r \subseteq V$ (eventualmente all'inizio $U_r = V_r = \emptyset$). Risolvere il sistema (2.1)-(2.4) con i soli vincoli (2.2) associati a U_r e i soli vincoli (2.3) associati a V_r, ottenendo una soluzione (x^*, η^*). Tale problema viene detto *master* ed è a tutti gli effetti un rilassamento del problema originario.

Passo 2 Data la soluzione (x^*, η^*) del master corrente, risolvere il problema di separazione

$$\begin{aligned} \max \pi^T (b - Ax^*) \\ \pi \in P_D \end{aligned}$$

Tale problema viene detto problema *slave* ed è a tutti gli effetti un CGLP.

Passo 3 Se il problema di separazione è illimitato, sia $\overline{v}$ un raggio estremo che ne certifichi l'illimitatezza. Aggiornare

$$V_r = V_r \cup \{\overline{v}\}$$

e tornare al Passo 1.

Passo 4 Se il problema di separazione ammette ottimo finito, sia $\overline{u}$ un vertice ottimo. Se

$$\eta^* \geq \overline{u}^T (b - Ax^*)$$

allora la soluzione corrente è ottima (la soluzione (x^*, η^*) soddisfa implicitamente tutti i vincoli della riformulazione). Altrimenti aggiornare

$$U_r = U_r \cup \{\overline{u}\}$$

e tornare al Passo 1.

L'interpretazione dell'algoritmo di Benders è la seguente: al Passo 1 il problema master fornisce una assegnamento x^* per le variabili intere x e una stima η^* sul contributo delle variabili continue y nel miglior completamento di x^*. Al Passo 2 il problema di separazione "verifica" tale soluzione, controllando che effettivamente esista almeno un completamento ammissibile e che la stima sia corretta. Se il problema di separazione è illimitato (cioè $\{y : By \geq b - Ax^*, y \geq 0\} = \emptyset$), allora non esiste nessun completamento y ammissibile e viene ritornato un taglio di ammissibilità (violato dalla soluzione x^* corrente) da aggiungere al master (Passo 3). In caso contrario, se anche la stima η^* è corretta allora l'algoritmo termina. Altrimenti viene ritornato un taglio di ottimalità che corregge la stima (Passo 4).

Vale la pena fare alcune osservazioni:

- l'algoritmo di Benders non è altro che un algoritmo cutting-plane puro, in cui però il problema master *non* è un problema di programmazione lineare, ma un MIP a sua volta.
- i tagli (2.2)-(2.3), seppur derivati per una particolare soluzione (x^*, η^*), sono globalmente validi (grazie alla teoria della dualità).
- la funzione obiettivo del problema di separazione non fa altro ce massimizzare la violazione del taglio da generare.
- la decomposizione di Benders non è limitata ai problemi MIP, ma può essere applicata anche in contesti diversi, ad esempio a problemi di programmazione lineare in cui vi sia una struttura come in Figura 1(b), come ad esempio quelli che nascono nella programmazione stocastica.

3 Decomposizione di Dantzig-Wolfe

La decomposizione di Dantzig-Wolfe si applica quando il problema si semplifica parecchio eliminando un sottoinsieme di vincoli (un caso particolare è la presenza di matrici di vincoli con struttura come in Figura 1(c)).

Vediamo l'applicazione di tale tecnica nel suo contesto più semplice, cioè la risoluzione di problemi di programmazione lineare, ad esempio

$$\begin{aligned} \min c^T x \\ Dx = d \\ Ax = b \\ x \geq 0 \end{aligned}$$

Per ipotesi, la regione ammissibile del nostro problema si può descrivere come

$$X = \{x : Dx = d\} \cap \{x : Ax = b, x \geq 0\}$$

dove i vincoli $Dx = b$ sono considerati *complicanti*, mentre l'insieme $P_x = \{x : Ax = b, x \geq 0\}$ è un insieme *semplice*, cioè sul quale è possibile ottimizzare una arbitraria funzione obiettivo efficientemente. Assumiamo che l'insieme P_x sia limitato, cioè un politopo. Per il teorema di Minkowski, è possibile esprimere ogni vettore $x \in P_x$ come combinazione convessa di un numero finito di vertici $\{x_g\}$:

$$P_x = \{x : x = \sum_{g \in G} \lambda_g x_g \quad \sum_{g \in G} \lambda_g = 1 \quad \lambda_g \geq 0\}$$

È possibile sostituire tale espressione per x nel sistema originario, ottenendo

$$\min \sum_{g \in G} (c^T x_g) \lambda_g \tag{3.1}$$

$$\sum_{g \in G} (D x_g) \lambda_g = d \tag{3.2}$$

$$\sum_{g \in G} \lambda_g = 1 \tag{3.3}$$

$$\lambda_g \geq 0 \tag{3.4}$$

Come per la decomposizione di Benders, questa è una riformulazione esattamente equivalente al problema di partenza, seppur di dimensione esponenziale. In questo caso abbiamo infatti un numero esponenziale di variabili (una per ogni vertice di P_x). Ancora una volta la soluzione consiste in un approccio iterativo, in cui si parte con un sottoinsieme delle variabili ed ad ogni passo ne vengono aggiunte alcune di nuove.

Il criterio con cui una variabile viene considerata necessaria è il costo ridotto. Difatti, se tutte le variabili ignorate hanno costo ridotto non negativo, allora la soluzione corrente è comunque ottima.

In particolare, si ha il seguente schema:

Passo 1 Si consideri il sottoinsieme $G_r \subseteq G$ (eventualmente all'inizio $G_r = \emptyset$). Risolvere il sistema (3.1)-(3.4) con le sole variabili associate a G_r. Sia (π^*, α^*) una soluzione duale ottima di tale problema, dove i moltiplicatori π e α corrispondo ai vincoli (3.2) e (3.3).

Passo 2 Data la soluzione duale (π^*, α^*) del master corrente, risolvere il problema di pricing

$$\min (c - D^T \pi^*)^T x$$
$$x \in P_x$$

Sia z^* il valore della sua soluzione ottima $\overline{x}$.

Passo 3 Se $z^* \geq \alpha^*$ non esiste nessun vertice x_g di costo ridotto negativo e l'algoritmo termina. Altrimenti ($z^* < \alpha^*$), aggiornare

$$G_r = G_r \cup \{\overline{x}\}$$

e tornare al Passo 1.

Si noti che a differenza di quanto accade nella decomposizione di Benders, il problema master *non* costituisce un rilassamento del problema originario e quindi il valore della sua soluzione *non* è un lower bound, se non all'ultima iterazione in cui lo slave di pricing non trova nessuna colonna di costo ridotto negativo.

4 Metodi a generazione di colonne

La decomposizione di Dantzig-Wolfe della sezione precedente può essere estesa alla risoluzione di problemi di programmazione lineare intera, con opportuni aggiustamenti.

Consideriamo innanzitutto la definizione di P_x. Ora siamo interessati alle soluzioni *intere* contenute in P_x, cioè stiamo considerando l'insieme Z

$$Z = \{x : x \in P_x \quad x \in \mathbb{Z}^n\}$$

Abbiamo due metodi a disposizione:

convessificazione le soluzioni intere in Z sono ancora ottenute come combinazione convessa dei vertici di P_x, ma le variabili originarie x devo essere mantenute nel problema per poter imporre il vincolo di interezza.

$$Z = \{x : x = \sum_{g \in G} \lambda_g x_g \quad \sum_{g \in G} \lambda_g = 1 \quad \lambda_g \geq 0 \quad x \in \mathbb{Z}^n\}$$

In tal caso la riformulazione diventa

$$\begin{aligned} \min \sum_{g \in G} (c^T x_g) \lambda_g \\ \sum_{g \in G} (D x_g) \lambda_g = d \\ \sum_{g \in G} \lambda_g = 1 \\ x = \sum_{g \in G} \lambda_g x_g \\ \lambda_g \geq 0 \\ x \in \mathbb{Z}^n \end{aligned} \tag{4.1}$$

discretizzazione le soluzioni intere di P_x sono ottenute come combinazione convessa *a coefficienti interi* delle soluzioni di Z. Questo equivale a considerare tutte le soluzioni di Z e poi sceglierne una.

$$Z = \{x : x = \sum_{g \in H} \lambda_g x_g \quad \sum_{g \in H} \lambda_g = 1 \quad \lambda_g \in \{0, 1\}\}$$

In tal caso la riformulazione diventa

$$\begin{aligned} \min \sum_{g \in H} (c^T x_g) \lambda_g \\ \sum_{g \in H} (D x_g) \lambda_g = d \\ \sum_{g \in H} \lambda_g = 1 \\ \lambda_g \in \{0, 1\} \end{aligned}$$

Vale la pena osservare che:

- a seconda dell'approccio seguito cambia leggermente il problema di pricing. Infatti nel primo caso abbiamo una variabile per ogni vertice di P_x, mentre nel secondo abbiamo una variabile per ogni soluzione di Z. Nel secondo caso il problema di pricing è quindi un problema di programmazione lineare intera.

- se P_x ha solo vertici interi, cioè $P_x = conv(Z)$, il rilassamento continuo delle formulazioni ottenute con entrambi i metodi è uguale (entrambi i metodi permettono di descrivere $conv(Z)$).

È possibile risolvere il rilassamento continuo di tali formulazioni esattamente come descritto nella sezione precedente. In generale però la soluzione ottima di tale rilassamento *non* corrisponderà ad una soluzione intera (a seconda della formulazione, x o λ non sono intere). È pertanto necessaria una fase enumerativa, in cui si combina l'algoritmo branch-and-bound con la generazione di colonne: l'algoritmo risultante prende il nome di *branch-and-price*.

L'implementazione di un algoritmo branch-and-price richiede alcune accortezze per avere un metodo corretto ed efficiente. In particolare:

- il tipo di branching deve essere scelto con cura, in quanto deve essere efficace e soprattutto non deve distruggere la struttura del problema di pricing. A questo riguardo si noti come fare branching sulle variabili λ *non* abbia queste proprietà, in quanto il fissare $\lambda_g = 0$ è una scelta molto debole (scarta una sola soluzione/vertice) e complica notevolmente il problema di pricing (che deve essere modificato opportunamente per non ritornare come ottime le soluzioni "vietate"). L'alternativa è quella di effettuare branching sulle variabili originarie x del problema, scelta generalmente molto più efficace ma che richiede comunque uno studio ad-hoc del problema.

- come il master continuo al nodo radice *non* costituisce un rilassamento per il problema di partenza a meno che non venga portata a termine la generazione di colonne, lo stesso vale per gli altri nodi dell'albero decisionale. Pertanto è necessario portare sempre a termine l'algoritmo di Dantzig-Wolfe per avere un bound corretto ad un nodo e poter eventualmente applicare bounding.

Si noti infine che la tecnica di generazione di colonne può essere usata come una euristica primale per il problema di partenza. In particolare è possibile risolvere in una prima fase il rilassamento continuo del nodo radice all'ottimo (con l'algoritmo di Dantzig-Wolfe) e poi risolvere con un algoritmo branch-and-cut a scatola chiusa la riformulazione ottenuta con il solo insieme di colonne generate nella prima fase . L'algoritmo che ne risulta è euristico in quanto il sottoinsieme di colonne considerato è sufficiente per ottenere una soluzione ottima del solo rilassamento continuo, ma non necessariamente l'ottimo intero.