Progetto di Metodi e Modelli per l'Ottimizzazione Combinatoria

Alberto De Bortoli

19 dicembre 2010

Sommario

Progetto del corso *Metodi e Modelli per l'Ottimizzazione Combinatoria* tenuto dal Prof. Luigi De Giovanni durante il I trimestre dell'A.A. 2009/'10 presso l'Università degli Studi di Padova.

Modellazione del problema di *Cutting Stock bidimensionale*, implementazione del modello di programmazione lineare con il framework SCIP e di una euristica costruttiva *greedy* in linguaggio C++.

Indice

1 Premessa

Il problema affrontato nel presente documento è un'estensione del noto problema del taglio di tondini di ferro. Tale problema ha un'importanza storica ed è ben noto fin dal 1939 quando Leonid Kantorovich lo formalizzò.

Il problema consiste nell'ottimizzare la produzione di tondini ricavati da pezzi in stock, o in altri termini di soddisfare la richiesta di pezzi monodimensionali partendo da oggetti monodimensionali di dimensioni maggiori o uguali ai pezzi richiesti.

L'estensione di tale problema alle due dimensioni comporta il taglio di pezzi bidimensionali da piani bidimensionali, ovvero di soddisfare richieste di rettangoli partendo da pannelli di dimensioni standard.

Si illustrerà un metodo di generazione di colonne a partire dalla soluzione ricavata con il caso monodimensionale per fornire un modello per il problema bidimensionale.

Si illustrerà una euristica di tipo costruttivo per lo stesso problema e si mostreranno confronti tra il modello di PLI e il metodo euristico su differenti input.

2 Modellazione lineare

2.1 Problema monodimensionale

La soluzione al problema ad una dimensione è stata discussa durante il corso per introdurre la metodologia di generazione di colonne. Il problema prevede di partire con un insieme di pattern di taglio monopezzo (le colonne del simplesso) e tramite la generazione di colonne aggiungere nuovi pattern che migliorano il valore della funzione obiettivo. Segue il modello.

Insiemi

- I: insiemi dei pezzi;
- J: insieme dei pattern di taglio.

Parametri

- W: lunghezza del pezzo in stock;
- L_i: lunghezza del pezzi $i \in I$.
- R_i: numero di pezzi richiesti per il tipo $i \in I$;
- N_{ij}: numero di pezzi di tipo $i \in I$ nello schema di taglio $j \in J$.

Variabili decisionali

- x_j: numero di tondini in stock da tagliare secondo lo schema $j \in J$.

Modello

$$\min \sum_{j \in J} x_j$$

$$\begin{aligned} s.t. \quad \sum_{j \in J} N_{ij} x_j &\geq R_i && \forall i \in I \\ x_j &\in \mathbb{Z}_+ && \forall j \in J \end{aligned}$$

Il problema modellato è detto problema *master*.

Dì per sé tale problema è sufficiente per ottenere una soluzione ammissibile del problema, ma occorre introdurre una seconda modellazione per ottenere nuove colonne per il problema *master* che migliorino il valore della funzione obiettivo.

Il problema a questo punto individuare una variabile, e quindi una colonna della matrice dei vincoli, con costo ridotto negativo. Il numero di potenziali colonne è molto elevato e, inoltre, le colonne non sono note esplicitamente.

Il problema che verrà modellato per tale scopo si chiamerà *slave*. Per ottenere un nuovo pattern che migliori la soluzione bisogna risolvere un problema *slave* utilizzando le variabili duali del problema master, ovvero i costi ridotti. Vorremmo ricavare una colonna con costo ridotto negativo, e per giudicarne l'esistenza possiamo calcolare il minimo tra tutti i costi ridotti delle potenziali variabili del problema.

$$\begin{aligned} \min \quad & \overline{c} = 1 - u^T z \\ \text{s.t.} \quad & \text{z è una possibile colonna della matrice dei vincoli} \end{aligned}$$

Il nuovo pattern che vorremmo aggiungere al problema *master* corrisponde all'inserimento di una nuova variabile (nel *master*) con costo in funzione obiettivo pari a 1 (consuma un oggetto in stock), quindi ogni colonna della matrice dei vincoli del problema *master* corrisponde ad uno schema di taglio e ogni elemento della colonna indica quanti pezzi di un certo tipo sono contenuti nello stesso schema.

La funzione obiettivo del problema *slave* risulta

$$\min \quad \overline{c} = 1 - \sum_{i \in I} u_i z_i$$

che possiamo trasformare in funzione di massimo. La variabile trovata dal problema *slave* introduce un nuovo vincolo violato dalla soluzione corrente nel problema *master* duale. Si mostra il modello.

$$\max \sum_{i \in I} u_i z_i$$

$$\begin{aligned} s.t. \quad & \sum_{i \in I} L_i z_i \quad \leq \quad W \\ & z_i \quad \in \quad \mathbb{Z}_+ \qquad \forall i \in I \end{aligned}$$

Se l'ottimo ottenuto dallo slave è maggiore di 1, è opportuno aggiungere la colonna al problema master e far entrare in base la nuova variabile, risolvendo dunque un nuovo simplesso. Nuove variabili vanno aggiunte al *master* finché il problema *slave* genera dei pattern (variabili per il *master*) che, se usati, migliorano il valore della funzione obiettivo. Ad ogni aggiunta di una

variabile, il rilassamento del problema *master* è continuo (le variabili x_j, ovvero la soluzione, sono in $\mathbb{R}$). Quando non vi sono più colonne miglioranti da aggiungere al *master* e una soluzione ottima in $\mathbb{R}$ è stata trovata, il problema può essere risolto all'interezza con la tecnica del Branch & Bound.

2.2 Problema bidimensionale

Estendere il problema a due dimensioni prevede il taglio di oggetti bidimensionali da oggetti in stock anch'essi bidimensionali. Si può pensare ad un esempio pratico: un'azienda deve soddisfare delle richieste di fogli di forma rettangolare ricavandoli da piani di materiale in stock. Si vuole trovare un modo per ritagliare gli oggetti richiesti minimizzando il numero di oggetti in stock da utilizzare.

Il modello del problema monodimensionale viene riutilizzato anche nel caso bidimensionale. L'approccio esecutivo prevede la suddivisione del pezzo in stock in strisce e trattare ognuna di esse come nel problema monodimensionale, quindi come un problema di ottimizzazione di zaino. Anche qui è necessario partire da un insieme di pattern di taglio monopezzo ricavati calcolando il numero massimo di oggetti i-esimi ricavabili da un unico foglio.

Il taglio a strisce viene chiamato "taglio a ghigliottina" intendendo che una striscia viene ricavata tagliando da un lato all'altro il pezzo in stock secondo una linea retta. Oltre al taglio lungo la *width* del pezzo in stock, ulteriori tagli a ghigliottina lungo la *length* della striscia sono consentiti. Poiché si vuole permettere di ricavare pezzi di diverse *width* dalla stessa striscia, un'ulteriore fase di trimming è consentita, definendo il tipo di taglio col nome "taglio a ghigliottina a due fasi con trimming", se ne mostra un esempio in figura 1.

Con questo modus operandi non è consentito il riutilizzo della parte tratteggiata in figura, definita col termine 'sfrido', che potenzialmente potrebbe essere sufficiente per ricavare ulteriori pezzi in una striscia.

Per ogni altezza di pezzo da tagliare viene risolto un problema di cutting stock monodimensionale, il quale fornisce come soluzione un pattern di taglio.

In ogni striscia verranno considerati soltanto i pezzi di spessore (proprietà *width* dei pezzi) non superiore a quella della striscia, ciò si può ottenere considerando un vettore u^T costruito *ad hoc* come segue:

$$u_i^T = 0 \qquad \forall i \in I \quad \text{t.c.} \quad pezzo_i.width \leq striscia.width$$

Nel problema slave, la variabile z_i associata all'u_i con valore zero non comparirà tra le variabili in base e quindi il pezzo i-esimo non verrà considerato nel pattern creato. Lo slave risolve un problema di zaino massimizzando

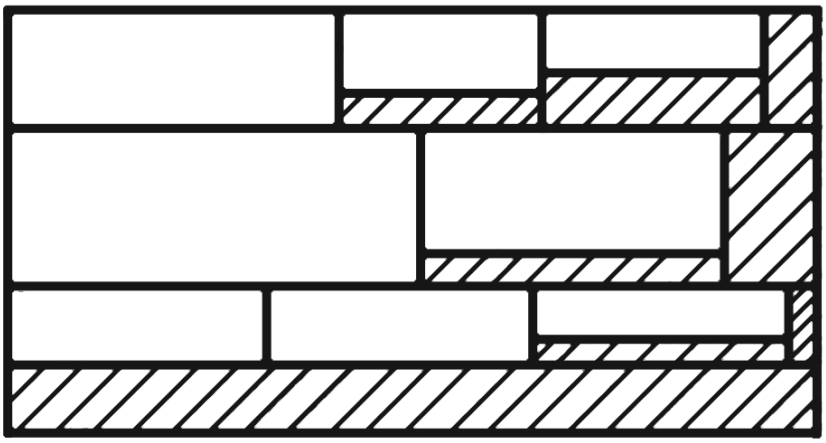

Figura 1: Esempio di taglio a ghigliottina a due fasi con trimming

l'utilizzo di una striscia, dove u_i è il vettore dei profitti. Si fa notare che a differenza del problema di zaino KP-01 dove le variabili possono assumere valori 0 o 1, il valore delle variabili in questo problema appartiene a $\mathbb{Z}_+$.

Non si fanno assunzioni di isotropia per il materiale tagliato, ovvero i pattern delle strisce dovranno rispettare l'orientamento dei pezzi. Ogni striscia conterrà la combinazione ottima dei soli pezzi con *width* non superiore alla *width* della striscia.

Per ogni spessore di striscia (spessore dei pezzi, $i \in \{1..|I|\}$) viene risolto il problema slave s_i con $i \in \{1..|I|\}$. Il vettore con i valori ottimi di $s_1...s_{|I|}$ servirà come vettore dei profitti per un ulteriore problema *slave* che opera sullo spessore (*width*) del pezzo in stock. Ogni s_i rappresenta il profitto che comporterebbe aggiungere tale pattern al taglio di un foglio (similmente al costo di un oggetto aggiunto allo zaino nel problema KP-01).

Il problema *slave* che opera sullo spessore del foglio fornisce un pattern di taglio secondo uno schema di strisce tali da massimizzare il numero di strisce ricavabili da un determinato foglio. L'ottimo di tale problema *slave* dice quanto migliora il valore della funzione obiettivo *master* tagliando un foglio in stock secondo un determinato pattern di strisce. Sarà opportuno aggiungere tale pattern di taglio al problema *master* se il valore ottimo dello *slave* i-esimo è maggiore di 1, per quanto detto nel caso del problema monodimensionale.

L'ottimo del problema master sarà il numero minimo di fogli da utilizzare[1] per soddisfare la richiesta. La soluzione x^* rappresenta quanti pattern di un

[1]per la modellazione data, che non è ottima per il *Bidimensional Cutting Stock Problem*

certo tipo (soluzioni dei problemi slave) sono utilizzati. Quanti pezzi sono stati tagliati in una soluzione x^* del *master* si ricava come segue:

$$pezzi_i = \sum_{j \in J} pattern_{ji} x_j \qquad \forall i \in I$$

Dove $pattern_{ji}$ è l'entrata indicante il pezzo i-esimo nel pattern trovato dallo slave j-esimo. Come nel caso monodimensionale, la soluzione ottima ottenuta aggiungendo tutti (o un numero limitato prefissato) di variabili non è intera. Vincolare l'interezza delle variabili si ottiene applicando l'algoritmo di Branch & Bound al problema finale ottenuto; ciò comporta la risoluzione di un simplesso in ogni nodo aperto dell'albero di Branch & Bound.

Estendendo il problema a più dimensioni rispetto al caso monodimensionale appare evidente che il problema master rimane invariato e ignaro del numero di dimensioni del problema di cutting stock; inoltre il problema slave (di zaino) viene utilizzato in più istanze e la soluzione finale è da intendersi come combinazione dei vari problemi risolti.

3 Metodo euristico *greedy*

3.1 Descrizione

L'euristica che si propone è di tipo costruttivo *greedy*. Il foglio in stock viene lavorato a strisce, tagliandolo secondo uno schema di taglio a "ghigliottina a due fasi con trimming", il ché di fatto rappresenta una soluzione allo stesso problema affrontato nella modellazione lineare. Anche in questo scenario si prevede che i pezzi richiesti non si possano tagliare isotropicamente (si sta risolvendo lo stesso problema del caso PL).

Il fatto di scegliere in maniera ottima localmente è dato dal fatto che si cerca sempre di inserire in una striscia il pezzo con *width* maggiore per minimizzare lo sfrido sulla parte inferiore della striscia e in caso non sia possibile, sceglie il pezzo con *width* massima tra quelli con *length* non maggiore dello spazio rimanente nella striscia. È necessario avere quindi un array ordinato decrescentemente per *width* dei pezzi, e a parità di *width*, per *length* decrescenti. I pezzi vanno tagliati in ordine dal più grande al più piccolo[2], fin quando le richieste non sono soddisfatte; si tagliano pezzi più piccoli solo per evitare di avere sfrido nella parte restante di striscia.

Si pensa sia migliore la valutazione dei pezzi secondo l'ordine prioritario descritto piuttosto di un ordinamento (ad esempio) decrescente in base all'area, in quanto pezzi con valori alti di *length* ma bassi di *width* potrebbero avere priorità rispetto a pezzi con width maggiore (i quali di fatto ridurrebbero lo sfrido nella parte bassa della striscia).

3.2 Pseudocodice

Si mostra la procedura iterativa in pseudocodice dell'euristica sviluppata.

$W \leftarrow SheetWidth$
$L \leftarrow SheetLength$
for $i = 1$ **to** $|I|$ **do**
  while $cut[i] < pieces[i].demand$ **do**
    if $piece[i].width \leq W$ **then**
      $W \leftarrow W - pieces[i].width$
      for $j = i$ **to** $|I|$ **do**
        while $cut[j] < pieces[j].demand$ **and** $pieces[j].length \leq L$ **do**
          $L \leftarrow L - pieces[j].length$;
          $cut[j] \leftarrow cut[j] + 1$

[2]prima per *width* decrescenti, poi per *length* decrescenti

```
            strip[j] ← strip[j] + 1
          end while
        end for
        sheet.push(strip)
        L ← SheetLength
        strip[j].clean();
      else
        for k = i to |I| do
          if piece[k].width ≤ W then
            W ← W − pieces[k].width
            for j = k to |I| do
              while cut[j] < pieces[j].demand and pieces[j].length ≤ L
              do
                L ← L − pieces[j].length;
                cut[j] ← cut[j] + 1
                strip[j] ← strip[j] + 1
              end while
            end for
            sheet.push(strip)
            L ← SheetLength
            strip[j].clean();
          end if
        end for
        solutions.push(sheet)
        sheet.clear()
      end if
    end while
  end for
```

L'array *pieces* contiene ordinatamente i pezzi da tagliare secondo l'ordine descritto nella precedente sezione, *cut* tiene conto di quanti pezzi di un certo tipo sono stati tagliati, *strip* memorizza il pattern di una striscia e *sheet* memorizza tutte le strisce ricavate in un foglio.

L'euristica permette di ottenere una soluzione molto buona del problema intero e talvolta migliore della soluzione intera data dalla soluzione del problema di PLI. Ciò è possibile perché applicando il Branch & Bound all'ultimo problema master generato, si risolve il problema originario limitato all'insieme dei soli pattern generati dalla soluzione dei sottoproblemi *slave*.

4 Prodotto sviluppato

Il progetto sviluppato consta di 3 parti:

1. Implementazione del modello di programmazione lineare in `C++` con il framework `SCIP`, con denominazione `bcsp_PLI`;
2. Implementazione di una euristica costruttiva *greedy* in `C++`, con denominazione `heuristic`;
3. Implementazione di un generatore di input per i programmi ai punti 1. e 2., con denominazione `generator`

4.1 Compilazione

Tutti i programmi sono stati sviluppati in ambiente Linux, la compilazione e l'esecuzione è quindi garantita in tale ambiente con le librerie `SCIP` e `SOPLEX` installate (si faccia riferimento alla sezione 4.3 per l'installazione) oltre al compilatore `g++` e il programma `make`. Per compilare i programmi è necessario eseguire da shell i seguenti 3 comandi dalla cartella `bcsp` fornita. È necessario modificare il valore della variabile `SCIPDIR` nel `Makefile` con il path relativo alla cartella di `SCIP`.

Per la compilazione del modello di programmazione lineare in `SCIP`:

```
make GMP=false READLINE=false ZIMPL=false
```

Per la compilazione dell'euristica:

```
make heuristic
```

Per la compilazione del generatore di input:

```
make generator
```

4.2 Utilizzo

A seguito della compilazione l'utilizzo avviene con i seguenti comandi dalla cartella `bcsp` fornita.

Per l'esecuzioni del modello di programmazione lineare in SCIP:

```
./bin/bcsp_PLI ./data/<file_input>.dat
```

Per l'esecuzione dell'euristica:

```
./bin/heuristic ./data/<file_input>.dat
```

Per l'esecuzione del generatore di input:

```
./bin/generator ./data/<output_file_input>.dat -i
```

Il parametro "`-i`" è opzionale e serve per forzare il contenuto del file in output ai soli valori interi.

4.3 Guida rapida all'installazione di SCIP

Per installare il sistema SCIP-SOPLEX utilizzato durante in corso, seguire le seguenti istruzioni. I termini di licenza e informazioni pi'u dettagliate sono reperibili su:

```
http://soplex.zib.be
http://scip.zib.be
```

Operazioni preliminari

Creare una directory dove scaricare e scompattare i file compressi. Si tenga presente che SCIP 'e un framework che permette (tra l'altro) la soluzione di problemi di programmazione lineare mista intera e che, per funzionare, ha bisogno di appoggiarsi su un solver per programmazione lineare. Nel nostro caso SCIP sar'a configurato per l'utilizzo del solver SOPLEX (che implementa il metodo del simplesso).

Installazione del solver SOPLEX

1. scaricare i sorgenti dalla pagina

   `http://soplex.zib.de/download.shtml`

   seguendo il link "SoPlex version 1.4.2: complete source code";

2. scompattare e compilare con make dalla directory soplex-1.4.2

   `make LPS=spx ZIMPL=false READLINE=false`

3. alla richiesta della directory del solver (prima domanda) inserire

   `soplex-1.4.2/src/../..`

4. alla richiesta della directory delle librerie del solver (seconda domanda) inserire

   `../../soplex-1.4.2/lib/libsoplex-1.4.2.linux.x86 64.gnu.opt.a`

5 Test e confronti

Si mostrano i risultati ottenuti dai programmi con metodo PLI e con metodo euristico sia in termini che di soluzione ottenuta. Si faccia riferimento ai seguenti file di input.

Input	**Vincoli**	**Dominio valori**
cut_small.dat	6	$\mathbb{Z}_+$
cut_gen20_f.dat	20	$\mathbb{R}_+$
cut_gen20_i.dat	20	$\mathbb{Z}_+$
cut_gen30_f.dat	30	$\mathbb{R}_+$
cut_gen30_i.dat	30	$\mathbb{Z}_+$
cut_gen40_i.dat	40	$\mathbb{Z}_+$
cut_gen50_i.dat	50	$\mathbb{Z}_+$

Di seguito si mostrano i risultati ottenuti con esecuzioni di `bcsp_PLI` impostando il numero massimo di iterazioni (numero massimo di pattern aggiunti) a 100.

Input	**Fogli usati**	**Tempo calcolo (sec)**
cut_small.dat	101	1
cut_gen20_f.dat	16	8
cut_gen20_i.dat	18	9
cut_gen30_f.dat	18	66
cut_gen30_i.dat	29	22
cut_gen40_i.dat	42	64
cut_gen50_i.dat	N/A	N/A (>3600)

Di seguito si mostrano i risultati ottenuti con esecuzioni di `heuristic`.

Input	**Fogli usati**	**Tempo calcolo (sec)**
cut_small.dat	100	0
cut_gen20_f.dat	15	0
cut_gen20_i.dat	17	0
cut_gen30_f.dat	19	0
cut_gen30_i.dat	33	0
cut_gen40_i.dat	40	0
cut_gen50_i.dat	32	0

La durata dell'esecuzione del metodo è espressa in secondi, in quanto `time.h` ha come granularità più fine tale unità di misura. Sarebbe in certa misura desiderabile una precisione decimale ma tali dati sono sufficienti ad evidenziare il notevole gap computazionale tra le esecuzioni dei due algoritmi.

Riferimenti bibliografici

[1] Gilmore P. C., R. E. Gomory (1961). A linear programming approach to the cutting-stock problem. Operations Research 9: 849-859

[2] Gilmore P. C., R. E. Gomory (1963). A linear programming approach to the cutting-stock problem - Part II. Operations Research 11: 863-888

[3] Gilmore P. C., R. E. Gomory (1963). Multistage cutting stock problems of two and more dimensions.A linear programming approach to the cutting-stock problem: 94-104