UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Naturali, Fisiche e Matematiche
Corso di Laurea Magistrale in Informatica

Corso di Complementi di Ricerca Operativa

Professor Marco Trubian

Relazione sul progetto di Column Generation per il problema dei P-Centri in linguaggio OPL

Rossini Nicole Alicia
Matricola 772006
a.a. 2009/2010

Indice generale

1. Introduzione

Lo scopo di questa relazione è evidenziare le funzionalità di base del linguaggio IMB ILOG OPL ed del suo linguaggio di scripting IBM ILOG Script for OPL, analizzando il codice da me sviluppato per la tecnica di Column Generation applicato al problema dei P-Centri. Per tanto non mi soffermerò sulla parte di modellazione.

2. IBM ILOG OPL

IBM ILOG OPL è il linguaggio sviluppato da IBM per la descrizione dei modelli.
Ogni modello è salvato in un file *mod* separato. Ad ogni modello può essere associato un file contenente i dati dell'istanza e questo file deve avere lo stesso nome del modello a cui si riferisce.
La struttura del file *mod* è composta da: dichiarazione dei dati, dichiarazione delle variabili di decisione, funzione obbiettivo e vincoli; inoltre può contenere dei blocchi di script in IBM ILOG Script for OPL.

2.1 Dati

I dati possono essere inizializzati internamente al modello oppure esternamente in un file *dat*.
La dichiarazione

```
int nNodi = ...;
```

significa che il numero di nodi del grafo verrà caricato dal file esterno.
La dichiarazione di array e matrici è la stessa:

```
int Distanze[Nodes][Nodes]= ...;
```

Nel file *dat* non bisogna specificare alcuna parola chiave, si riporta solo il nome della variabile a cui devono essere assegnati i dati e si chiude l'assegnamento con un punto e virgola.

```
nNodi = 6;
Distanze = [
    [0 6 2 5 1 5 ]
    [6 0 6 1 5 1 ]
    [2 6 0 5 1 5 ]
    [5 1 5 0 4 2 ]
    [1 5 1 4 0 4 ]
    [5 1 5 2 4 0 ]
];
```

Gli array richiedono parentesi quadre anche nel file dati.
Per creare l'insieme di nodi, dichiaro internamente un *range*, da *1* a *nNodi*:

```
range Nodes = 1..nNodi;
```

Inoltre, predispongo il modello ad accettare nuove colonne dallo script in fase di esecuzione. OPL non consente l'aggiunta di dati agli array, l'operazione è consentita solo per oggetti di tipo IloTupleSet. Per prima cosa quindi dichiaro una tupla composta da un intero per memorizzare il costo della colonna e un array di interi per contenere la colonna:

```
tuple coveringSet{
     int cost;
     int column[Nodes];
};
```

e dichiaro successivamente il mio oggetto di tipo IloTupleSet:

```
{coveringSet} A = ...;
```

con inizializzazione esterna al modello. Questo perché i dati verranno aggiunti all'oggetto *masterData* di tipo IloDataElement, che verrà passato al modello ad ogni iterazione. Dal punto di vista del modello si tratta quindi di dati provenienti dall'esterno.

Ho provato anche le dichiarazioni:

```
{coveringSet} A;
{coveringSet} A = {};
```

ma queste non consentivano l'aggiunta di dati dallo script. Il messaggio d'errore riportava che l'elemento *A* era già stato settato in precedenza.
Dal momento che esiste una dichiarazione esterna, nel file dati deve necessariamente essere presente la dichiarazione:

```
A = { };
```

Le parentesi graffe indicano gli insiemi. Se volessi inizializzare manualmente il TupleSet dovrei utilizzare le parentesi <> per indicare la tupla:

```
A = {
     <0 [0 0 0 0 1 0]>
     <1 [0 1 0 1 0 1]>
};
```

Sia nelle tuple che negli array è possibile usare la virgola come separatore tra i vari elementi, ma non è strettamente necessario.

2.2 Variabili

Dichiaro un array di variabili:

```
dvar int x[A] in 0..1;
```

una per ogni elemento del TupleSet. Le variabili del modello hanno come parola chiave *dvar*, che sta per decision variables, mentre le variabili del linguaggio di scripting sono dichiarate come *var*. Le *dvar* inoltre sono tipizzate, in questo caso le ho dichiarate *int in 0..1* (accetta anche la parola chiave *binary*), mentre le variabili dello script non lo sono.

2.3 Funzione obbiettivo

La funzione obbiettivo in OPL, a differenza di AMPL, non ha un nome.

```
minimize sum(i in A) i.cost * x[i];
```

l'indice *i* indica la colonna di *A* e, per accedere al campo *cost*, si usa *i* come riferimento alla colonna, non come indice di elemento in *A.*

2.4 Vincoli

I vincoli devono essere racchiusi tutti in un unico blocco:

```
subject to {
     //vincoli
};
```

Ad ogni vincolo associo un nome in modo da poter accedere successivamente ai valori delle variabili duali associate ai vincoli:

```
pCentri: sum(j in A) x[j] == P;
```

Da notare la differenza tra l'operatore = di assegnamento e l'operatore == di confronto.

2.5 Blocco execute

I blocchi execute servono per eseguire codice scritto in ILOG Script all'interno dello scope del modello.
I blocchi execute posti alla fine del modello appartengono alla fase di post-processing. Possono essere anche utilizzati nella fase di pre-processing, ponendoli prima del modello, per esempio per inizializzare i dati.
Per poter avere più blocchi execute all'interno dello stesso modello è necessario assegnare un nome diverso ad ognuno.

I blocchi vengono eseguiti in ordine di dichiarazione.
In questo caso è necessario un blocco execute per poter accedere ai valori delle variabili duali, a cui non si può avere accesso dallo scritp di flow control, il quale, in questo caso, è rappresentato dall'unico metodo main.

```
execute FillDuals {
   for(var i in Nodes) {
         Duali[i] = setCovering[i].dual;
   }
   DualeP = pCentri.dual;
   writeln("MSTR Duali: setCovering ",Duali," e PCentri ",DualeP);
   writeln("MSTR Soluzione: ",x);
}
```

I valori delle variabili associate ai vincoli sono proprietà dell'oggetto IloConstraint, a cui si accede con la sintassi *nomeVincolo.proprietà*. Rendo disponibili questi valori allo script di flow control salvandole in variabili precedentemente dichiarate del modello.

2.6 Gli altri modelli

Per il problema di pricing ed il problema di inizializzazione creo altri due file con i rispettivi modelli.
Per poter passare i dati del master problem agli altri modelli, è necessario che questi abbiano tutti i dati dichiarati esterni al modello.

3 IBM ILOG Script for OPL

IBM ILOG Script for OPL è il linguaggio di scripting utilizzato nelle fasi di pre-processing, post-processing e flow control. È un linguaggio ad oggetti basato su JavaScript. Il nome di tutti gli oggetti ha la forma: *IloNomeOggetto.* Per scorrevolezza di lettura ometterò il prefisso *Ilo* quando l'oggetto sarà già stato introdotto.
Il flow control in OPL si colloca nel metodo *main* all'interno del file del modello principale.

3.1 Soglia

All'inizio dello script metto una variabile per il valore di soglia rispetto a cui testerò il valore della funzione obbiettivo del problema di pricing, in modo che sia facilmente individuabile e modificabile:

```
var RC_EPS = 1.0e-2;
```

3.2 Preparazione oggetti

Per prima cosa si genera il modello:

```
thisOplModel.generate();
```

questo causa l'inizializzazione delle variabili e ne rende disponibili i valori, sia quelle inizializzare internamente tramite espressioni, sia quelle che prendono i dati dal file *dat*.
Una volta generato, estraggo la definizione del modello:

```
var masterDef = thisOplModel.modelDefinition;
```

salvo l'istanza di Cplex:

```
var masterCplex = cplex;
```

ed estraggo l'oggetto IloDataElement, che contiene tutti i dati esterni del modello:

```
var masterData = thisOplModel.dataElements;
```

Questi elementi serviranno ad ogni iterazione per instanziare un nuovo modello da risolvere.

Le stesse variabili create per il master problem servono sia per il modello di inizializzazione, sia per il modello di pricing. Diversamente dal modello del master problem però, la definizione dei modelli si trova in file esterni. Quindi, per prima cosa, creo un oggetto IloOplModelSource, che prende come parametro del costruttore il nome del file del modello da caricare:

```
var subSource = new IloOplModelSource("Pricing.mod");
```

Dal file sorgente estraggo la definizione del modello e la salvo in un oggetto:

```
var subDef = new IloOplModelDefinition(subSource);
```

Creo anche un oggetto IloDataElement vuoto, che verrà riempito successivamente con i dati presi dal master problem:

```
var subData = new IloOplDataElements();
```

Infine, creo un'istanza di Cplex per il problema di pricing:

```
var subCplex = new IloCplex();
```

Lo stesso faccio per il problema di inizializzazione:

```
var initSource = new IloOplModelSource("Init.mod");
var initDef = new IloOplModelDefinition(initSource);
var initData = new IloOplDataElements();
var initCplex = new IloCplex();
```

3.3 Inizializzazione

Preparati gli oggetti, inizio a costruire una soluzione di partenza per il problema.
La prima soluzione consiste nel dividere i nodi esattamente in *P* insiemi, prendendo i nodi nell'ordine dato. Questa inizializzazione può portare a costruire soluzioni non ammissibili quando il grafo non è completo.
La seconda soluzione consiste nel prendere come colonne di partenza le righe della matrice di adiacenza del grafo. Questa soluzione è inutile nel caso di grafi completi, perché darebbe origine a *P* insiemi, tutti contenenti l'insieme completo dei nodi del grafo.
Quindi, la seconda soluzione viene utilizzata solo se la prima non ha prodotto un numero sufficiente di colonne.
All'interno di ogni sottoinsieme così costruito è poi necessario individuare quale sia il miglior centro che minimizzi la massima distanza da ogni altro nodo appartenente al sottoinsieme. Di questo si occupa il modello di inizializzazione.
Quindi per ogni colonna generata si risolve il problema *Init*.
Per poterlo fare, bisogna preparare i dati da fornire al modello: inizio a riempire l'oggetto *initData* con i dati necessari presi dal problema master:

```
initData.nNodi = masterData.nNodi;
initData.Distanze = masterData.Distanze;
initData.Archi = masterData.Archi;
```

initData, prima di questi assegnamenti, è un oggetto vuoto. La chiamata:

```
initData.nNodi
```

crea automaticamente una variabile di nome *nNodi* all'interno del DataElement.
Inoltre, è necessario inizializzare un array di interi destinato ad ospitare la colonna che verrà creata:

```
initData.Colonna = thisOplModel.intArray;
```

In questo caso ho creato tra i dati del problema un array vuoto di interi appositamente per questo scopo, poiché non avevo altri array di interi monodimensionali a mia disposizione. L'alternativa sarebbe stato creare un array tramite il costruttore *new Array()* messo a disposizione da ILOG Script for OPL. Ma questo crea array di variabili non tipizzate che non possono essere accettate dal modello, dal momento che attende degli interi in input; bisognerebbe quindi operare un cast.
Un'altra cosa da notare è che l'array *intArray* era stato inizializzato internamente al modello, quindi, per richiamarlo, è necessario accedervi dall'oggetto *thisOplModel*, invece che dal

DataElement *masterData*, il quale contiene solo i dati esterni al modello.
Una volta che la colonna è inizializzata, posso procedere ed assegnare valori numerici che verranno automaticamente interpretati come interi.
Inizio un ciclo *while* per dividere i nodi in *P* colonne.
La sintassi e gli operatori di ILOG Script for OPL sono gli stessi di JavaScript. Le funzioni matematiche appartengono al linguaggio OPL; sono rese disponibili in OPL Script come metodi statici dell'oggetto *Opl*:

```
var resto=masterData.nNodi % masterData.P;
var setSize=Opl.floor(masterData.nNodi / masterData.P);
```

Da notare come i cicli *for* necessitino, diversamente da quanto accade in OPL per i costrutti *foreach*, di variabili dichiarate:

```
for(var j in initData.Colonna){}
```

Diversamente da Java però, lo scope della variabile *j* non è limitato al solo ciclo *for*, ma è esteso all'intero metodo *main*: quando verrà riutilizzata in cicli seguenti, non dovrà più essere dichiarata.
Un altro commento necessario è sui valori che assume *j* durante il ciclo. In questo caso l'elemento *Colonna* è dichiarato su un range di interi, quindi *j* assume valori interi. Ma se avessimo deciso di rappresentare i nomi dei nodi tramite stringhe, allora *j* avrebbe assunto come valori stringhe di caratteri. Per questo ho utilizzato una variabile *index* da incrementare ad ogni passo del ciclo, in modo da avere sempre a disposizione un intero su cui operare.
Nel ciclo assegno i valori *1* o *0* direttamente ai dati di input del problema di inizializzazione:

```
if(index>=min & index<max ){
      initData.Colonna[j] = 1;
}
else{
      initData.Colonna[j] = 0;
}
```

All'uscita dal ciclo in *initData.Colonna* ho la prima colonna costruita. Quindi devo passare i dati al modello e chiamare il solutore per individuare il miglior centro e, soprattutto, il costo della colonna, cioè la massima distanza tra il miglior centro e gli altri nodi.
Creo l'ultimo oggetto, il modello vero e proprio che lega la definizione del modello all'istanza del solutore che voglio utilizzare:

```
var initOpl = new IloOplModel(initDef, initCplex);
```

Aggiungo i dati al modello appena creato:

```
initOpl.addDataSource(initData);
```

e genero il modello:

```
initOpl.generate();
```

A questo punto posso ordinare al solutore di risolvere l'istanza del problema chiamando:

```
initCplex.solve()
```

La chiamata restituisce *true* se la risoluzione ha avuto successo, quindi la utilizzo in un blocco *if..else*:

```
if( initCplex.solve()){
      // Operazioni in caso di esito positivo
  }else{
      writeln("INIT: Nessuna soluzione accettabile");
}
```

In questo caso mosto l'eventuale insuccesso, senza fermare la procedura, perché non preclude il possibile esito positivo del metodo. Più avanti il controllo sarà necessario per interrompere il column generation quando non ci saranno soluzioni ammissibili.
In caso di esito positivo devo aggiungere una tupla *<costo,colonna>* al TupleSet *A.*Lo faccio aggiungendo la tupla all'interno del DataElement *masterData*:

```
masterData.A.addOnly(initOpl.maxCosto.solutionValue,initData.Colonna);
```

Per aggiungere una tupla ad un TupleSet esistono due metodi: *add()* e *addOnly()*. Entrambi prendono come parametri i valori da assegnare ai campi della tupla. La differenza tra i due consiste solo nel valore di ritorno: il primo restituisce la tupla aggiunta, il secondo no; *addOnly()* quindi risulta essere un metodo più veloce, da preferire ad *add()* nel caso non si sia interessati ad avere l'oggetto IloTuple che il metodo costruisce per poter operare l'aggiunta al TupleSet.

Il costo della colonna è il valore della funzione obbiettivo, che, in questo caso, è rappresentata dalla sola variabile di decisione *maxCosto*.
Il valore finale della variabile si ottiene chiamando la proprietà *solutionValue* della variabile, la quale, a sua volta, è una proprietà del modello:

```
initOpl.maxCosto.solutionValue
```

Il valore della funzione obbiettivo si ottiene invece dall'istanza di Cplex:

```
initCplex.getObjValue();
```

Le due cose, in questo caso, si equivalgono.

La colonna adesso appartiene ai dati del problema master. Incremento un contatore per sapere quante colonne ho aggiunto al TupleSet, dal momento che l'oggetto IloTupleSet non fornisce metodi per ottenere il numero di tuple appartenenti all'insieme.

Prima della fine del ciclo è necessario terminare l'oggetto IloOplModel *initOpl*:

```
initOpl.end();
```

Il metodo *end()* rilascia la memoria allocata ed elimina l'oggetto dalla memoria. Alla variabile *initOpl* verrà assegnato un nuovo oggetto nell'iterazione successiva.

3.4 Ciclo di generazione

Preparo un boolean *equals*, che utilizzerò per controllare il ciclo di generazione, una variabile *currColumn* per memorizzare la colonna generata al passo *k e* una variabile *lastColumn* per la colonna del passo *k-1.* Il boolean equals indica se *lastColumn* è identica a *currColumn.* Questo non accade mai se il valore di soglia *RC_EPS* del column generation è ben tarato, ma in fase di progettazione il valore può non essere ottimale e il metodo può entrare in loop infinito. Per sicurezza, quindi, fermo il ciclo in caso di generazione consecutiva di colonne identiche.

```
while ( !equals ) {
     lastColumn=currColumn;
     /* Solve MASTER */
     /* Solve PRICING */
          currColumn=subOpl.y.solutionValue;
     /* Aggiungi colonna */

     equals=true;
     for(  i in subOpl.Nodes ){
         equals=equals && (currColumn[i]==lastColumn[i]);
     }
}
```

All'inizio del ciclo salvo il valore della colonna generata nell'iterazione precedente in *lastColumn*. Successivamente risolvo il problema master per ottenere il valore delle variabili duali. Passo questi valori in input al problema di pricing, lo risolvo e ottengo la nuova colonna da aggiungere ai dati del master problem che salvo in *currColumn*. Aggiungo la colonna ai dati del problema master per l'iterazione successiva. Alla fine controllo che *currColumn* sia diversa da *lastColumn*.

3.4.1 Risoluzione del problema master

All'interno del ciclo, per ogni iterazione è necessario creare l'oggetto IloOplModel per poter risolvere il problema, aggiungere i dati e generare il modello:

```
var masterOpl = new IloOplModel(masterDef, masterCplex);
masterOpl.addDataSource(masterData);
masterOpl.generate();
```

Inoltre, è necessario rilassare le condizioni di interezza delle variabili. Per farlo si chiama il metodo:

```
masterOpl.convertAllIntVars();
```

Questo metodo non agisce sulla definizione del modello, ma sull'oggetto IloOplModel, che è un'istanza del modello, quindi ad ogni iterazione l'oggetto verrà creato con i vincoli d'interezza delle variabili attivi e dovranno essere rilassati esplicitamente.
Chiamo il solutore e controllo il proseguimento del metodo nel caso non esistano o meno soluzioni ammissibili:

```
if ( masterCplex.solve() ) {
  masterOpl.postProcess();
}
else {
  writeln("No solution!");
  masterOpl.end();
  break;
}
```

Nel caso in cui esista una soluzione indico esplicitamente di passare alla fase di post-processing: viene quindi eseguito il blocco execute *FillDuals* che copia i valori delle variabili duali in array accessibili a questa porzione di codice.
Nel caso in cui, invece, non esista una soluzione rilascio la memoria allocata all'oggetto ed interropo il ciclo.

3.4.2 Risoluzione del problema pricing

La risoluzione del problema di pricing è analoga.
Passo i dati dal problema master al DataElement *subData*:

```
subData.nNodi=masterOpl.nNodi;
subData.Distanze=masterOpl.Distanze;
subData.Archi=masterOpl.Archi;
subData.dualCost=masterOpl.Duali;
subData.dualAlpha=masterOpl.pCentri.dual;
```

Creo l'istanza del modello a cui passo i dati e genero l'istanza:

```
var subOpl = new IloOplModel(subDef, subCplex);
subOpl.addDataSource(subData);
subOpl.generate();
```

Chiamo il solutore:

```
if ( subCplex.solve() ) {
  currColumn=subOpl.y.solutionValue;
}
else {
  writeln("SUB No solution!");
  subOpl.end();
  masterOpl.end();
  break;
}
```

se esiste la soluzione salvo la colonna in *currColumn*, altrimenti chiamo il metodo *end()* sugli oggetti in memoria ed interrompo l'esecuzione.

3.4.3 Controllo per la terminazione

Se il valore della funzione obbiettivo del problema di pricing è maggiore di una piccola quantità negativa (condizione di non negatività rilassata) interrompo il ciclo di generazione:

```
if (subCplex.getObjValue() >= -RC_EPS) {
      subOpl.end();
      masterOpl.end();
      break;
}
```

3.4.4 Preparazione dell'iterazione successiva

Se il ciclo non viene arrestato, preparo l'iterazione successiva aggiungendo la colonna appena trovata al DataElement del problema master:

```
masterData.A.addOnly(subOpl.maxDist.solutionValue,currColumn);
```

In questo caso il costo della colonna è rappresentato dalla variabile di decisione *maxDist*, che non coincide col valore della funzione obbiettivo.

Alla fine rilascio la memoria degli oggetti IloOplModel:

```
subOpl.end();
masterOpl.end();
```

3.5 Soluzione intera finale

All'uscita del ciclo tutte le colonne necessarie alla soluzione del problema sono presenti nel DataElement, ma non è ancora stata generata la soluzione ottima.

Per l'ultima volta instanzio l'oggetto *masterOpl:*

```
masterOpl = new IloOplModel(masterDef,masterCplex);
masterOpl.addDataSource(masterData);
masterOpl.generate();
```

ma, diversamente da quanto fatto all'interno del ciclo di generazione, non rilasso le condizioni di interezza delle variabili.

Chiamo il solver e stampo a video la soluzione:

```
if ( masterCplex.solve() ) {
    writeln();
    writeln("OBJECTIVE: ",masterCplex.getObjValue());
    for( i in  masterData.A) {
        writeln("Col : ", i," selected: ",masterOpl.x[i].solutionValue);
    }
    writeln("Dimensione A : ", sizeA);
}else{
    writeln("Impossibile generare soluzione intera!");
}
```

Infine, rilascio la memoria dell'oggetto:

```
masterOpl.end();
```