Appunti sulla Sintassi e sui Comandi di AMPL Plus v1.6

a cura di
R. Bruni , G. Fasano , G. Liuzzi*

AMPL è un *linguaggio di modellazione* per la programmazione matematica. Serve ad esprimere un problema di ottimizzazione in una forma che sia comprensibile da un generico solutore. È un *linguaggio algebrico*, cioè contiene diverse primitive per esprimere la notazione matematica normalmente utilizzata nello scrivere problemi di ottimizzazione quali sommatorie, funzioni matematiche elementari, ecc.
Vediamo ora alcune nozioni elementari per poter scrivere i primi modelli di programmazione matematica. Ciascuna istruzione di AMPL deve terminare con un '`;`'. Questo vuol dire che, nello scrivere una istruzione, possiamo inserire tra le parole chiave del linguaggio quanti spazi e ritorni a capo vogliamo senza per questo generare errori. Spesso questa libertà di scrittura viene sfruttata per indentare il file dei comandi in modo da renderne più agevole la lettura. Quindi, benché i due seguenti frammenti di codice in AMPL

```
var x1; var x2; minimize obiettivo: x1+x2; subject to vincolo1: x1 >= 0;
subject to vincolo2: x2 >=0;subject to vincolo3: x1 <= 10;
subject to vincolo4: x2 <= 10; s.t. vincolo3: x1-x2 <= 0;
```

e

```
var x1;
var x2;

minimize obiettivo: x1+x2;

subject to vincolo1: x1 >= 0;
subject to vincolo2: x2 >=0;
subject to vincolo3: x1 <= 10;
subject to vincolo4: x2 <= 10;
s.t.       vincolo3: x1-x2 <= 0;
```

abbiano sintatticamente lo stesso significato, il secondo formato è certamente più leggibile del primo.
Sebbene AMPL consenta di scrivere in un unico file (con estensione `.mod`) un problema e farlo quindi risolvere al solutore, concettualmente è sempre meglio tenere ben separati il file di "modello" (`.mod`), in cui è descritta la struttura logica del modello del problema in esame, dal file dei "dati" (`.dat`), in cui invece sono scritti i valori numerici del problema stesso. Per uno stesso modello, i dati possono essere contenuti in uno o più file `.dat`. In questo modo, mantenendo fisicamente separati il modello dai suoi dati, è possibile cambiare i dati modificando solo il file relativo (`.dat`) senza quindi il pericolo di introdurre errori nel modello (`.mod`). Si noti che il file di modello ha obbligatoriamente estensione `.mod`, quello di dati obbligatoriamente estensione `.dat`.

*bruni@dis.uniroma1.it, fasano@dis.uniroma1.it, liuzzi@dis.uniroma1.it

È possibile inserire in un file `.mod` o `.dat` delle righe di commento; a tal fine queste devono necessariamente essere precedute dal simbolo `#`.

1 Gli Insiemi in AMPL

Gli *insiemi* sono strutture di dati molto utilizzate in AMPL. È anzitutto necessario avere ben presente la distinzione tra *dichiarazione* di un insieme (nel file `.mod`)

> con la quale semplicemente si comunica ad AMPL che un certo nome scelto da noi rappresenta un generico insieme non meglio specificato

e *definizione* di un insieme (in genere nel file `.dat`)

> con la quale invece si assegnano all'insieme precedentemente dichiarato, i suoi elementi.

La dichiarazione di un insieme (o di un altro oggetto) deve sempre precedere la sua definizione. Quello che si fa di solito è mettere tutte le dichiarazioni nel file `.mod` del modello e le relative definizioni nel file dei dati con estensione `.dat`.
Per dichiarare un insieme dobbiamo usare la parola chiave `set` seguita dal nome dell'insieme. Nell'assegnare nomi agli insiemi, così come a tutte le altre entità del modello che vedremo più avanti, bisogna tenere presente il fatto che AMPL è *case sensitive* cioè distingue le lettere maiuscole dalle minuscole. Quindi, le istruzioni:

```
set CARTONI;
set Cartoni;
```

dichiarano `CARTONI` e `Cartoni` come insiemi distinti di elementi non meglio specificati. Una volta dichiarato un insieme, è possibile assegnargli degli elementi. L'istruzione AMPL per questo assegnamento è la seguente (nel file `.dat`)

```
set CARTONI := pippo topolino paperino pluto paperone;
```

Dati due insiemi `A` e `B` è possibile, in AMPL, eseguire su di essi alcune operazioni elementari quali

Operazione	Significato
`A union B`	insieme degli elementi che sono in A oppure in B
`A inter B`	insieme degli elementi che sono in A e B
`A diff B`	insieme degli elementi che sono in A e non in B
`A symdiff B`	insieme degli elementi che sono in A oppure in B ma non in entrambi
`card(A)`	restituisce il numero di elementi di `A`

quindi se `A` e `B` fossero definiti nel `.dat` come

```
set A := 1 3 5 7 9 11;
set B := 9 11 13 15 17;
```

avremmo

Operazione	Risultato
`A union B`	1 3 5 7 9 11 13 15 17
`A inter B`	9 11
`A diff B`	1 3 5 7
`A symdiff B`	1 3 5 7 13 15 17
`card(A)`	6

Per dichiarare un insieme come sottoinsieme di un altro insieme precedentemente dichiarato, si usa la parola chiave `within` come in questo esempio

```
set A ;
set B within A;
```

ove `B` è dichiarato come sottoinsieme di `A`. Ovviamente nel definire `A` e `B` dovremo essere coerenti con la dichiarazione precedente.
Sebbene, matematicamente parlando, un insieme è una collezione di elementi non ordinati, AMPL offre la possibilità di definire anche insiemi *ordinati* di elementi. La dichiarazione di un insieme ordinato è simile al caso non ordinato ma seguita, questa volta, dalla parola chiave `ordered`. Pertanto

```
set NUMERI ordered;
```

dichiara NUMERI come insieme *ordinato* di elementi non meglio specificati. L'istruzione AMPL che assegna valori ad un insieme ordinato è esattamente identica a quella del caso non ordinato, solo che questa volta l'ordine con cui scriviamo gli elementi definisce il loro ordinamento all'interno dell'insieme. Così per esempio

```
set NUMERI := 3 7 100 2 8;
```

definisce l'insieme `NUMERI` come insieme dei numeri `3, 7, 100, 2, 8` e l'elemento `100` in questo ordinamento precede sia `2` che `8`. Dato un insieme ordinato, è possibile eseguire su di esso alcune operazioni particolari, cioè non definibili su insiemi non ordinati. Esse sono:

Operazione	Risultato
`first(NUMERI)`	primo elemento di `NUMERI`
`last(NUMERI)`	ultimo elemento di `NUMERI`
`next(t,NUMERI,n)`	n-esimo elemento in `NUMERI` dopo l'elemento `t`[1]
`prev(t,NUMERI,n)`	n-esimo elemento in `NUMERI` prima dell'elemento `t`
`next(t,NUMERI)`	uguale a `next(t,NUMERI,1)`
`prev(t,NUMERI)`	uguale a `prev(t,NUMERI,1)`
`ord(t,NUMERI)`	posizione di `t` nell'insieme `NUMERI`
`ord0(t,NUMERI)`	come sopra ma restituisce 0 se `t` non sta in `NUMERI`
`member(j,NUMERI)`	elemento di `NUMERI` in j-esima posizione

Quindi se l'insieme `NUMERI` è definito come sopra allora

`set NUMERI := 3 7 100 2 8;`	
Operazione	Risultato
`first(NUMERI)`	3
`last(NUMERI)`	8
`next(7,NUMERI,2)`	2
`prev(8,NUMERI,3)`	7
`next(100,NUMERI)`	2
`prev(100,NUMERI)`	7
`ord(2,NUMERI)`	4
`ord0(17,NUMERI)`	0
`ord0(7,NUMERI)`	2
`member(3,NUMERI)`	100

Supponiamo ora che `A` e `B` siano dichiarati come

[1] t deve essere un elemento di `NUMERI`.

```
set A;
set B ordered;
```

valgono le seguenti considerazioni

Dichiarazione	Ordinato?
`B diff A`	SI
`B union A`	NO
`A diff B`	NO
`A symdiff B`	NO

Ovviamente, come è logico aspettarsi, alcuni insiemi, come le progressioni aritmetiche (p.es. i numeri pari da 1 a 100), sono dotati di un ordinamento predefinito. Il più semplice insieme ordinato è quello dei numeri interi compresi tra due valori. Questo insieme non necessita di alcuna dichiarazione e si indica con la notazione

```
1..N;
```

ove `N` è un numero intero. Mediante la parola chiave `by` è possibile, opzionalmente, specificare la distanza tra due interi consecutivi nell'insieme. Per esempio

```
1..100 by 5;
```

indica l'insieme di numeri `1,6,11,16,21,26,... 91,96`. Ovviamente `1..100` e `1..100 by 1` indicano esattamente lo stesso insieme di numeri.
Sebbene non sia molto comune, è possibile in AMPL dichiarare collezioni di insiemi indicizzati su altri insiemi. Così, per esempio

```
set INDEX;
set INS{INDEX};
```

dichiara `INS` come un array di insiemi, tanti quanti sono gli elementi dell'insieme `INDEX`. Per riferirsi ad un ben preciso insieme dell'array `INS` si usa la seguente notazione

```
INS[i];
```

che individua l' i-esimo insieme dell'array di insiemi `INS`. Come vedremo questa notazione è molto comune nella dichiarazione di *parametri* nel file di modello.

1.1 Insiemi a più dimensioni

In AMPL è possibile specificare la dimensione di un insieme contestualmente alla sua dichiarazione, semplicemente facendo seguire alla dichiarazione dell'insieme la parola chiave `dimen` seguita da un numero indicante la dimensione desiderata. Così, per esempio

```
set TUPLE dimen 3;
```

dichiara l'insieme `TUPLE` come insieme di triplette ordinate. Ciascun suo elemento sarà quindi costituito da triple di valori (ancora non specificati). Una possibile definizione dell'insieme `TUPLE` potrà essere perciò la seguente

```
set TUPLE := (1,7,1) (7,1,1) (10,5,3) (pippo,paperino,pluto);
```

Notiamo che, essendo ciascuna tripla *ordinata*, i due elementi `(1,7,1)` e `(7,1,1)` sono considerati distinti. Un modo alternativo per ottenere un insieme di dimensione maggiore di uno è quello di usare insiemi gia dichiarati e quindi usare la parola chiave `cross`, come nel seguente esempio

```
set INS1;
set INS2;
set INS3;
set TUPLE := INS1 cross INS2 cross INS3;
```

che dichiara `TUPLE` come prodotto cartesiano dei tre insiemi `INS1, INS2, INS3`. Così, se i tre insiemi fossero definiti da

```
set INS1 := A B C;
set INS2 := 1 2 3;
set INS3 := X Y Z;
```

l'insieme `TUPLE` sarebbe formato dalle triple: `(A,1,X) (A,1,Y) (A,1,Z) (A,2,X) (A,2,Y) (A,2,Z) (A,3,X) (A,3,Y) (A,3,Z) ...` Ovviamente è anche possibile seguire il procedimento inverso, ovvero ottenere da un insieme a tre dimensioni i tre insiemi contenenti ciascuno la corrispondente componente di una tripla. Si usa a tale scopo la parola chiave `setof` come nel seguente esempio

```
set TUPLE dimen 3;
set INS1 := setof{(i,j,h) in TUPLE} i;
set INS2 := setof{(i,j,h) in TUPLE} j;
set INS3 := setof{(i,j,h) in TUPLE} h;
```

2 I Parametri in AMPL

I *parametri* sono i dati che compaiono nel modello AMPL che descrive il problema di ottimizzazione. Bisogna stare molto attenti a non confondere i parametri con le *variabili* del problema. Sebbene, infatti, sia possibile ed anzi molto frequente, modificare i valori di uno o più parametri, modificando in questo modo il problema che si sta trattando, una volta avviato il processo risolutivo ovvero una volta invocato un solutore per il problema, il valore dei parametri rimane *costante*. Al contrario invece, è possibile assegnare alle variabili dei valori *iniziali*, ma il loro valore viene modificato dal solutore che, anzi, in generale restituirà un valore finale ottimo diverso dall'eventuale valore iniziale assegnato.
Un parametro può essere utilizzato solo dopo la sua dichiarazione. Per dichiarare un parametro, si usa la parola chiave `param`, seguita dal nome del parametro. La più semplice dichiarazione di parametro è:

```
param T;
```

È possibile dichiarare vettori e matrici di parametri con un'unica istruzione in cui si dichiara, oltre al nome del parametro, anche l'insieme entro cui varia l'indice, o gli indici, delle sue componenti. Ovviamente, a meno che non si usi un insieme predefinito di quelli visti pocanzi, bisognerà usare un insieme gia dichiarato. Così le istruzioni

```
set C;
param costi{C};
```

dichiarano `C` come insieme e `costi` come vettore di parametri indicizzati sull'insieme `C`. Quindi, per maggiore chiarezza, il parametro `costi` avrà tante componenti quanti sono gli elementi dell'insieme `C`. Analogamente,

```
param N integer;
param costi{1..N};
```

definiscono un parametro intero `N` ed un vettore di parametri `costi` con tante componenti quanti sono i numeri interi da 1 ad `N`. È anche possibile far comparire nella dichiarazione di un parametro, non un insieme ma due o più.

```
set VAR;
set VINC;
parm a{VINC,VAR};
```

Quanto sopra ha l'effetto di dichiarare due insiemi `VAR` e `VINC` (i cui elementi non sono ancora stati specificati), ed un parametro *bidimensionale* `a` con elementi identificati da coppie di valori, il primo appartenente all'insieme `VINC` ad il secondo all'insieme `VAR`.
Ovunque occorra usare una specifica componente del parametro `costi` oppure `a` si dovra usare una notazione del tipo

```
...costi[10]...
....a[i,j]...
```

in cui 10 deve essere compreso tra 1 ed `N`, mentre `i` e `j` devono essere elementi rispettivamente di `VINC` e `VAR`.
Contestualmente alla dichiarazione di un parametro è possibile specificarne alcune restrizioni, per cui

```
param T >1 integer;
```

dichiara il parametro `T` come un numero intero maggiore di 1.

2.1 Assegnazione di Valori ai Parametri

Come si è detto, i valori dei parametri sono tipicamente assegnati nel file `.dat` (ed ovviamente non possono più essere cambiati) sempre attraverso la parola chiave `param`, seguita dal nome del parametro (come dichiarato nel file di modello), dal simbolo `:=` e da un valore.

```
param T:= 1;
param C:= 20;
```

Per assegnare valori ad un parametro monodimensionale (vettore), occorre specificare le coppie indice valore. Quindi, avendo dichiarato nel file di modello

```
set indice;
param vettore{indice};
```

nel file dei dati un assegnamento ammissibile per l'insieme `indice` ed il parametro `vettore` potrebbe essere il seguente

```
set indice := A B;
param: vettore := A 1  B 3;
```

Per aumentare la leggibilità del file dei dati, conviene indentare l'istruzione `param` in modo da ottenere

```
param: vettore := A 1
                  B 3;
```

Nelle due precedenti istruzioni il simbolo `:` che segue la parola chiave `param` è, in realtà, opzionale cioè potremmo non metterlo. Vedremo, tuttavia, che ci sono casi in cui è obbligatorio mettere i ":" e casi in cui, invece, è obbligatorio *non* mettere i due punti. Per esempio, un caso in cui l'uso dei due punti (":") dopo la parola `param` è obbligatorio, si ha quando vogliamo assegnare valori a due o più vettori di parametri monodimensionali e indicizzati sullo stesso insieme. Supponiamo per esempio che nel file del modello siano presenti le seguenti istruzioni

```
set indice;
param vett1{indice};
param vett2{indice};
```

Per assegnare valori ai due vettori di parametri AMPL ci offre la possibilità di usare una sola istruzione, e precisamente:

```
set indice := A B;
param: vett1 vett2 :=
    A    1     4
    B    3     7   ;
```

Qui il simbolo ":" è obbligatorio perchè serve per avvertire AMPL del fatto che stiamo per definire non un vettore ma due (o più) contemporaneamente.
Per un parametro bidimensionale, il discorso è solo leggermente più complicato. Infatti, avendo dichiarato `matrice` come

```
set dim1;
set dim2;
param matrice{dim1,dim2};
```

l'istruzione standard per assegnare valori a `matrice` sarebbe la seguente:

```
set dim1 := X Y Z;
set dim2 := A B C;
.
.
param:  matrice := X A 1   X B 2   X C 3
                   Y A 3   Y B 1   Y C 2
                   Z A 7   Z B 5   Z C 4;
```

che prevede di specificare tutte le componenti mediante indicazione del primo indice, secondo indice e valore. Facciamo notare che anche questa volta il simbolo ":" è opzionale. Questo metodo presenta degli ovvi svantaggi, non ultimo quello di dover ripetere molte volte gli stessi indici. Per questo motivo è prevista anche la più concisa notazione seguente

```
param matrice: A  B  C :=
             X 1  2  3
             Y 3  1  2
             Z 7  5  4;
```

In pratica, è come se la matrice venisse inserita per colonne. Cerchiamo di essere un po' più chiari. Il comando precedente, dal simbolo ":" in poi è esattamente uguale all'assegnazione di valori a tre parametri fittizi aventi il nome delle colonne `A`, `B` e `C` ed indicizzati sullo stesso insieme `dim1`. Per questo motivo possiamo dire che tutto va come se stessimo assegnando valori alla matrice *per colonne*. Ovviamente, in questo caso, dopo la parola `param` abbiamo dovuto specificare il nome del parametro `matrice`, in modo tale da far capire ad AMPL che stiamo assegnando valori ad un parametro a due dimensioni.
Se una matrice è molto larga e poco alta, e quindi non agevolmente visibile nella schermata, conviene scrivere al suo posto la matrice *trasposta*, facendo seguire al nome della matrice la parola chiave `(tr)`, ottenendo

```
param matrice(tr): X  Y  Z :=
                 A 1  3  7
                 B 2  1  5
                 C 3  2  4;
```

Per finire, descriviamo brevemente il caso in cui si debbano assegnare dei valori ad un parametro con tre o più dimensioni. Per non appesantire troppo la scrittura, supponiamo di avere un parametro `pippo` a tre dimensioni cioè:

```
set DIM1;
set DIM2;
set DIM3;
param pippo{DIM1,DIM2,DIM3};
```

e supponiamo che i tre insiemi siano definiti nel file dei dati come

```
set DIM1 := X Y Z;
set DIM2 := A B C;
set DIM3 := A1 B2 C3;
```

Oltre al metodo tradizionale di assegnare valori al parametro, ovvero, come visto nel caso di matrici a due dimensioni

```
param: pippo := X A A1 7   X A B2 8   X A C3 9
                X B A1 6   X B B2 7   X B C3 8
                  ...        ...        ...
                Z C A1 1   Z C B2 2   Z C C3 3;
# il due punti e' opzionale
```

anche in questo caso è prevista una notazione più concisa. In pratica quello che si fa è "*affettare*" il parametro in matrici di dimensione due e quindi assegnare valori in maniera molto simile a quanto visto prima, ovvero

```
param pippo:=
    [X,*,*]:  A1  B2  C3 :=
         A    10  20  30
         B    15  25  35
         C    40  50  60

    [Y,*,*]:  A1  B2  C3 :=
         A    17  23  29
         B    10  20  30
         C    10  29  35

    [Z,*,*]:  A1  B2  C3 :=
         A     1   2   3
         B     7   5   9
         C    10  10  10 ;
```

Notiamo che nell'istruzione precedente è assolutamente vietato l'uso dei due punti (":") dopo la parola `param`. Inoltre, coerentemente con quanto detto, ovvero che stiamo assegnando valori a delle sottomatrici di dimensione due del parametro a tre dimensioni, la notazione

```
[X,*,*]:  A1  B2  C3 :=
     A    10  20  30
     B    15  25  35
     C    40  50  60
```

è, chiaramente, molto simile a quella vista per assegnare valori ad un parametro a due dimensioni.

Abbiamo visto che è possibile definire un parametro con tante componenti quanti sono gli elementi di un insieme, semplicemente facendo seguire al nome del parametro il nome dell'insieme racchiuso tra parentesi graffe. Nel caso in cui si vuole specificare un parametro con tante componenti quante sono le coppie di elementi del prodotto cartesiano di due insiemi, con scrittura analoga, si mettono tra parentesi graffe i nomi dei due insiemi (o più), separati dalla virgola

```
param pippo{dim1,dim2};
```

Più in generale possiamo sostituire {`dim1,dim2`} con altre cossiddette *espressioni di indicizzazione*. Qui sotto ne riportiamo alcune:

```
{A}                    # tutti gli elementi di A
{A,B}                  # tutte le coppie di elementi uno
                       # di A e uno di B
{i in A, j in B}       # come sopra
{i in A, B}            # come sopra
{A, j in B}            # come sopra
{i in A: p[i]>0}       # tutti gli elementi di A tali che
                       # p[i] > 0
{i in A, i in B}       # tutte le coppie di elementi uno
                       # di A e uno di B purche uguali
```

Le precedenti espressioni di indicizzazione vengono usate tanto nella dichiarazione dei parametri, quanto in quella degli insiemi, delle variabili e dei vincoli. Inoltre tali espressioni sono usate per definire sommatorie e produttorie aritmetiche.

3 Dichiarazione delle Variabili

A differenza dei parametri, le variabili sono dei simboli il cui valore numerico deve essere calcolato dal solutore; cionondimeno possiamo indicare nel file dei dati, dei valori *iniziali* per le variabili. Quando parleremo di modelli e problemi non lineari, vedremo anche come sia importante poter assegnare alle variabili dei valori iniziali da cui far partire il solutore. Le variabili rappresentano le grandezze incognite che vogliamo conoscere risolvendo il problema di ottimizzazione. Si dichiarano con la parola chiave `var` seguita dal nome della variabile. Tutte le variabili devono essere dichiarate prima di poter essere utilizzate. La più semplice dichiarazione di variabile è del tipo

```
var x;
```

Le variabili possono essere indicizzate ed è possibile porre dei bound su di esse. L'istruzione

```
var  x{p in VAR} >=0, <=LIMIT;
```

indica, per esempio, un vettore di variabili `x` con indice che varia nell'insieme `VAR` (precedentemente dichiarato), che devono essere tutte $>= 0$ e $<=$ del parametro `LIMIT` (anche esso precedentemente dichiarato).
È possibile, anche se inusuale, specificare restrizioni di uguaglianza sulle variabili.

```
var Buy{j in FOOD} = f_min[j];
```

questo vuol dire che siamo interessati solo alle soluzioni in cui le variabili hanno il valore dei corrispondenti parametri `f_min`.
Come detto, è possibile specificare dei valori iniziali sulle variabili. L'istruzione

```
var Buy{j in FOOD} := f_min[j];
```

inizializza le variabili Buy ai valori dei corrispondenti parametri f_min. Tuttavia, in questo caso, il solutore, pur partendo da questi valori iniziali, può modificarli ed anzi, sperabilmente li modificherà migliorando in questo modo il valore della funzione obiettivo. Si noti che il significato delle due istruzioni appena viste, seppure qpparentemente simile, è completamente diverso. Mentre infatti nel primo caso (con il segno =) si vincolano le variabili ad essere uguali ai valori di f_min, nel secondo caso il valore iniziale può essere cambiato.
Se una variabile è vincolata ad assumere solo valori interi, nella sua dichiarazione bisognerà usare la parola chiave `integer` come in questo esempio:

```
var x{INS} integer;
```

che dichiara un vettore di variabili (tante quante sono le componenti dell'insieme INS) tutte a valori *interi*. Se invece, una variabile è vincolata ad assumere solo i valori 0 o 1, nella sua dichiarazione dovremo usare la parola chiave `binary`. L'istruzione

```
var y binary;
```

dichiara la variabile y che può assumere solo valore 0 od 1. L'uso di questo tipo di variabili è molto frequente soprattutto per esprimere le condizioni logiche presenti nel modello.

4 Funzione obiettivo

È ciò che vogliamo massimizzare o minimizzare nel problema di ottimizzazione. AMPL non genera nessun errore se vengono specificati due obiettivi, ma semplicemente considera come funzione obiettivo il primo dichiarato. La funzione obiettivo è dichiarata con la parola chiave `maximize` o `minimize`, seguita obbligatoriamente da un qualsiasi nome, dai due punti, e da una espressione in cui possono comparire solo gli insiemi, i parametri e le variabili già definiti. La più semplice dichiarazione di obiettivo è

```
minimize obiettivo: x;
```

Vuol dire che vogliamo che il solutore trovi un valore per le variabili tale da rendere minimo il valore di x. Nel caso invece

```
minimize total_cost: sum{j in VAR} cost[j]*x[j];
```

si intende minimizzare la somma dei prodotti dei costi per le rispettive variabili. Di regola, nell'obiettivo compaiono sempre le variabili, i.e. la funzione obiettivo non è costante.

5 Vincoli

Sono delle specifiche che dobbiamo soddisfare nel problema di ottimizzazione. Sono dichiarati con la parola chiave `subject to`. Questa parola chiave è opzionale, dato che ogni dichiarazione che non comincia con una parola chiave viene considerata un vincolo, e può anche essere abbreviata come `subj to` o `s.t.`. I vincoli devono avere un nome, seguito dai due punti e una espressione in cui possono comparire solo gli insiemi, i parametri e le variabili già definiti. Deve ovviamente comparire un operatore di relazione (<, <=, >, >=, =) ed ogni vincolo termina al solito con ';'. La più semplice dichiarazione di vincolo è

```
subject to vinc: x<=4;
```

nella quale vogliamo che il solutore trovi una soluzione in cui la variabile `x` valga al più 4 (questo tipo di vincolo poteva anche essere specificato come restrizione nella dichiarazione della variabile `x`).
I vincoli, come le altre entità in AMPL, possono essere indicizzati. Quindi, le istruzioni seguenti

```
set VINC;
set VAR;
param a{VINC,VAR};
param b{VINC};

var x{VAR};
.
.
s. t. limiti{s in VINC}: sum{i in VAR} a[s,i]*x[i] <= b[s];
```

dichiarano:
- `a` come parametro a due dimensioni (matrice) con indice di riga che varia nell'insieme `VINC` e indice di colonna che varia in `VAR`;
- `b` come parametro ad una dimensione con indice che varia nell'insieme `VINC`
- `limiti` come un *vettore* di vincoli (tanti quante sono le componenti dell'insieme `VINC`).

Se l'insieme `VINC` è formato dai quattro elementi `A, B, C, D`, l'istruzione

```
s. t. limiti{s in VINC}: sum{i in VAR} a[s,i]*x[i] <= b[s];
```

equivale ai quattro vincoli

```
s. t. limite_a: sum{I in VAR} a['A',i]*x[i] <= b['A'];
s. t. limite_b: sum{I in VAR} a['B',i]*x[i] <= b['B'];
s. t. limite_c: sum{I in VAR} a['C',i]*x[i] <= b['C'];
s. t. limite_d: sum{I in VAR} a['D',i]*x[i] <= b['D'];
```

Quindi, è un modo di scrivere i vincoli in forma compatta. D'altra parte però, questo è l'unico modo possibile di scrivere i vincoli se non conosciamo quali sono gli elementi dell'insieme `VINC`. Ricordiamo a questo proposito che nel file del modello, dove bisogna scrivere i vincoli, non sono ancora noti gli elementi che compongono l'insieme `VINC`.
Con gli strumenti sintattici visti è possibile esprimere direttamente una grande varietà di modelli.

6 Espressioni Aritmetiche in AMPL

Le funzioni aritmetiche che è possibile utilizzare sono:

Significato	indicato con
Valore assoluto di x	`abs(x)`
Arcoseno(x)	`acos(x)`
Arcoseno iperbolico(x)	`acosh(x)`
Arcocoseno(x)	`asin(x)`
Arcocoseno iperbolico(x)	`asinh(x)`
Arcotangente iperbolico(x)	`atanh(x)`
Seno(x)	`sin(x)`
Coseno(x)	`cos(x)`
Tangente(x)	`tan(x)`
Tangente iperbolica(x)	`tanh(x)`
Parte intera inferiore di x	`ceil(x)`
Parte intera superiore di x	`floor(x)`
Logaritmo naturale loge x	`log(x)`
Logaritmo decimale log10 x	`log10(x)`
Esponenziale e^x	`exp(x)`
Radice quadrata di x	`sqrt(x)`
Minimo tra 2 o più numeri (x, y, z, ?)	`min(x,y,z,?)`
Massimo tra 2 o più numeri (x, y, z, ?)	`max(x,y,z,?)`

Gli operatori che è possibile utilizzare, in ordine di precedenza decrescente, sono:

Significato	Tipo	indicato con	o anche con
Potenza	Aritmetico	^	**
Numero negativo	Aritmetico	-	
Somma	Aritmetico	+	
Sottrazione	Aritmetico	-	
Moltiplicazione	Aritmetico	*	
Divisione	Aritmetico	/	
Divisione intera	Aritmetico	div	
Modulo	Aritmetico	mod	
Differenza non negativa: max(a-b,0)	Aritmetico	less	
Sommatoria	Aritmetico	sum	
Produttoria	Aritmetico	prod	
Minimo	Aritmetico	min	
Massimo	Aritmetico	max	
Unione di insiemi	Insiemistica	union	
Intersezione di insiemi	Insiemistica	inter	
Differenza tra insiemi	Insiemistica	diff	
Differenza simmetrica tra insiemi	Insiemistica	symdiff	
Prodotto cartesiano tra insiemi	Insiemistica	cross	
Appartenenza ad un insieme	Insiemistica	in	
Non appartenenza ad un insieme	Insiemistica	not in	
Maggiore, maggiore o uguale	Aritmetico	>, >=	
Minore, minore o uguale	Aritmetico	<, <=	
Uguale	Aritmetico	=	==
Diverso	Aritmetico	<>	!=
Negazione logica	Logico	not	!
And logico	Logico	and	&&
Quantificatore esistenziale logico	Logico	exists	
Quantificatore universale logico	Logico	forall	
Or logico	Logico	or	||
If then else	Aritmetico	if then else	

7 Espressioni Logiche

È possibile utilizzare alcune istruzioni logiche mediante le quali attribuire risultati del tipo `VERO`/`FALSO` ad alcune grandezze. Un primo insieme di tali istruzioni è già stato menzionato in precedenza ed è dato dagli operatori relazionali:

Operatore	Significato
=	uguale a
<>	diverso da
<	minore di
>	maggiore di
<=	minore o uguale a
>=	maggiore o uguale a

Quindi per esempio le istruzioni:

```
set ORE:= 1..24;
```

```
param mezzogiorno {ORE} = 12;
```

verificano se il parametro `mezzogiorno` vale 12, se ciò non accade (FALSE), il compilatore segnala un errore.
Le istruzioni `in` e `within` costituiscono altri operatori che come descritto in precedenza sono utilizzati per gli insiemi, rispettivamente per definire l'appartenenza di un elemento ad un insieme come in :

```
mezzogiorno in ORE
```

oppure per definire sottoinsiemi di elementi come in:

```
POMERIDIANE within ORE
```

in cui `POMERIDIANE` è un sottoinsieme di `ORE`.
Altre istruzioni logiche di uso comune sono indicate nella seguente tabella:

Operatore	Significato
and	(`a` *and* `b`) è TRUE se `a` e `b` sono entrambe TRUE
or	(`a` *or* `b`) è TRUE se almeno uno tra `a` e `b` è TRUE
not	(*not* `a`) è TRUE se `a` è FALSE (e viceversa)

Per esempio se il parametro `mezzogiorno` è pari a 12, allora

```
(mezzogiorno > 13) or (mezzogiorno < 15)
```

restituisce TRUE (è cioè verificata la seconda relazione). Invece il seguente vincolo

```
s.t. vinc {i in ORE and i=mezzogiorno}: x[i] = 1000;
```

provvede ad assegnare alla sola componente `mezzogiorno` del vettore `x`, il valore 1000. Va ricordato che l'operatore `not` ha precedenza rispetto all'operatore `and` e questi a sua volta ha precedenza rispetto ad `or`. Ciò implica che l'espressione:

```
(not(i in PAPEROPOLI)) and pluto or paperino
```

è differente da:

```
(not(i in PAPEROPOLI)) and (pluto or paperino)
```

Altri due operatori logici di particolare utilità sono `exist` e `forall`, che generalizzano gli operatori `or` e `and` rispettivamente; infatti l'espressione:

```
exists {i in ORIGINI} spedite[i] > 5
```

è TRUE se *almeno una* componente del vettore di parametri `spedite`, è maggiore di 5; invece l'espressione:

```
forall {i in ORIGINI} spedite[i] > 5
```

è TRUE se *tutte* le componenti del vettore `spedite` sono maggiori di 5.
Notiamo che nella definizione di un'espressione logica non possono essere presenti variabili perchè esse non hanno un valore fisso e determinato.

Uso delle espressioni logiche

A parte gli usi già incontrati, il principale impiego delle espressioni logiche è nella definizione di insiemi (anche di quelli usati per definire sommatorie e insiemi di vincoli). Vediamo alcuni esempi:

```
set GRANDI:= {i in PRODUTTORI: capacita[i] >= 20};
```

definisce `GRANDI` come il sottoinsieme di `PRODUTTORI` che hanno una capacità non inferiore a 20 (ovviamente si suppone che `capacita` sia un vettore di parametri definito sull'insieme `PRODUTTORI`). Notiamo la sintassi del comando: ciò che viene dopo i due punti è una specifica. La notazione $\boxed{:}$ corrisponde all'uso classico dei ":" come abbreviazione di "tale che...". Più formalmente il comando precedente definisce l'insieme `GRANDI` come l'insieme degli "`i`" in `PRODUTTORI` tali che `capacita[i] >= 20` è TRUE. L'espressione dopo ":" può essere una qualunque espressione logica, con la sola restrizione che deve far riferimento a parametri e grandezze precedentemente definite. Riportiamo di seguito alcuni esempi di espressioni valide:

```
set GRANDI:= {i in PRODUTTORI: capacita[i] >= sum {j in PRODUTTORI}
                                   capacita[j]/card(PRODUTTORI)};

set GRANDI:= {i in (PRODUTTORI inter NORD): exists {j in FORMAGGI}
                                             costo[i,j] >= 1000};

set GRANDI:= {i in PRODUTTORI : exists {j in FORMAGGI} costo[i,j] >=
                                            1000 and (i in NORD) };

set GRANDI:= {i in (PRODUTTORI inter NORD): (forall {j in FORMAGGI}
        costo[i,j] >= 1000) and (capacita[i] >= sum {j in PRODUTTORI}
                                    capacita[j]/card(PRODUTTORI))};
```

Ovviamente abbiamo supposto che `capacita, costo, PRODUTTORI, FORMAGGI` siano stati definiti in precedenza. Per inciso, notiamo che la seconda e la terza espressione sono equivalenti e definiscono lo stesso insieme. Una volta definito l'insieme `GRANDI`, questi può essere usato in una sommatoria o nella definizione di un insieme. Per esempio i seguenti due comandi sono equivalenti:

```
s.t. vincolo_grandi: sum {i in GRANDI} x[i] <= 124;

s.t. vincolo_grandi: sum {i in PRODUTTORI: capacita[i] >= 20}
                                                 x[i] <= 124;
```

Nel primo caso stiamo supponendo che l'insieme `GRANDI` sia stato definito in precedenza, nel secondo invece non serve questa assunzione. Analogamente sono equivalenti le seguenti due istruzioni:

```
s.t. vincolo_grandi {i in GRANDI}:  y[i] <= 124;

s.t. vincolo_grandi{i in PRODUTTORI: capacita[i] >= 20}: y[i] <= 124;
```

Un ulteriore uso delle espressioni logiche è con l'istruzione `check`:

`check:` espressione logica;

Ovunque compaia questa istruzione AMPL calcola l'espressione logica dopo ":". Se è TRUE allora procede, altrimenti segnala un errore e si arresta. Vediamo ora due esempi dell'istruzione `check`:

```
check: sum {i in PRODUTTORI} capacita[i] >= 500;

check:  (sum {i in PRODUTTORI} capacita[i] >= 500) or
        (sum {i in PRODUTTORI, j in FORMAGGI} costi[i,j] <= 1200);
```

Sostanzialmente il comando `check` serve a verificare che i dati comunicati al solutore rispettino le specifiche richieste. In questo senso il comando `check` estende la possibilità di effettuare controlli sui dati, rispetto ai semplici controlli possibili all'interno delle dichiarazioni `param` e `set` (es. `param a` > 0 o `set A within B`). Infine come ultimo esempio di uso di espressioni logiche, esaminiamo il comando `if-then-else`. Questo comando permette di scegliere una tra due espressioni, sulla base del valore (TRUE o FALSE) di un'espressione logica. A titolo di esempio si consideri allora il seguente frammento di codice:

```
set PERIODI ordered;
set PRODOTTI;


param scorte1{PRODOTTI};
param scorte2{PRODOTTI};

var x{PRODOTTI,PERIODI};
var y{PRODOTTI,PERIODI};

s.t. vincolo {p in PRODOTTI, t in PERIODI}:
                x[p,t] + ( if     t=first(PERIODI)
                           then   scorte1[p]
                           else   scorte2[p]) = y[p,t];
```

allora quando l'indice `t` è proprio il primo elemento dell'insieme `PERIODI`, alla variabile `x[p,t]` viene sommato `scorte1[p]`, altrimenti viene sommato `scorte2[p]`.

8 Modelli non lineari

Ogni qual volta la funzione obiettivo e/o i vincoli del problema di ottimizzazione presentano delle nonlinearità il problema è un problema di ottimizzazione non lineare. Tipicamente, per poter risolvere problemi non lineari, dobbiamo usare un solutore diverso da quello del caso lineare. Nella versione per studenti di AMPL Plus 1.6 è presente MINOS v5.5 come solutore nonlineare. Tuttavia, di default AMPL usa come solutore CPLEX che è un solutore di problemi lineari. Quindi quando vogliamo risolvere un problema non lineare, come prima cosa dobbiamo selezionare (nel menu `solver` di AMPL Plus) l'opzione MINOS.
A differenza del caso lineare, in cui CPLEX è sostanzialmente sempre in grado di determinare una soluzione del problema o concludere che il problema è illimitato o inammissibile (cfr. Teorema fondamentale della PL), nel caso nonlineare, a meno che il problema non sia estremamente semplice, nessun solutore è in grado di garantire il fatto di trovare sempre una soluzione. Oltre a questo, la soluzione di un problema nonlineare è fortemente dipendente, oltre che dal solutore, anche dal punto iniziale da cui si fa partire il processo di soluzione stesso.
In AMPL, la parola chiave per assegnare dei valori iniziali alle variabili è `let`. L'istruzione

```
let x := 25;
```

assegna alla variabile `x` il valore iniziale 25. L'istruzione `let` serve per assegnare valori iniziali nel file dei dati `.dat`. È anche possibile assegnare un valore iniziale ad una variabile contestualmente alla sua dichiarazione nel file `.mod`, semplicemente facendo seguire alla dichiarazione della variabile il simbolo `:=` seguito dal valore iniziale desiderato. Ad esempio

```
param valIniz;
var x1 := 25;
var x2 := valIniz;
```

assegnano ad `x1` e `x2` rispettivamente `25` ed il valore del parametro `valIniz`.
Dal momento che la complessità di un problema nonlineare dipende in parte dal numero di variabili del problema, è sempre raccomandabile eliminare dal problema tutte quelle variabili che in maniera semplice dipendono da altre variabili. AMPL offre la possibilità, in maniera automatica, di ricercare ed eliminare dal modello tutte le variabili "inutili". Per fare questo occorre dare, nella finestra dei comandi `commands`, il seguente comando AMPL

```
option substout 1;
```

A differenza di un solutore lineare, un solutore nonlineare ha bisogno di essere opportunamente aggiustato per risolvere efficientemente un problema nonlineare. Le opzioni di default infatti sono sufficienti ad affrontare problemi non troppo complicati ma non appena il problema si complica un po', diventa necessario poter cambiare le opzioni di funzionamento del solutore. Il modo in cui in AMPL si interviene sulle opzioni del solutore è mediante delle istruzioni `option`. Alla parola chiave `option` bisogna fare seguire una parola chiave dipendente dal solutore e che ne indica le opzioni cioè, nel caso di `MINOS` `minos_options`. Infine bisogna specificare, racchiuse tra singoli apici, le opzioni che si vuole modificare seguite da "=" e il valore desiderato.

```
option minos_options '<generica_opzione>=<valore>';
```

Di seguito riportiamo un elenco delle principali opzioni del solutore `MINOS`.

Opzione	Valore	Significato
`Completion`	`partial (default)`	i sottoproblemi sono risolti parzialmente
	`full`	i sottoproblemi sono risolti completamente
`Hessian_dimension`	`r (default 50)`	dimensione dell'Hessiano
`Major_iterations`	`i (default 50)`	numero max di iter. esterne
`Minor_iterations`	`i (default 40)`	numero max di iter. interne
`Superbasics_limit`	`i (default 50)`	numero max di variabili superbasiche

Quindi per esempio se volessimo impostare a 100 (anziché 50) la dimensione dell'Hessiano, dovremmo scrivere la seguente istruzione

```
option minos_options 'Hessian_dimension=100';
```

9 I problemi su reti

Finora sono stati formulati mediante il linguaggio di modellazione AMPL, problemi di programmazione lineare piuttosto generali. È possibile tuttavia mostrare con semplici esempi, che alcuni problemi di programmazione lineare possono essere descritti, in maniera del tutto naturale, mediante formulazioni che potremmo definire "strutturate". Ciò costituisce un elemento aggiuntivo per l'utente che deve risolvere la formulazione in quanto è possibile:

(1) osservare e risolvere la formulazione come un qualsiasi problema di programmmazione lineare (PL);

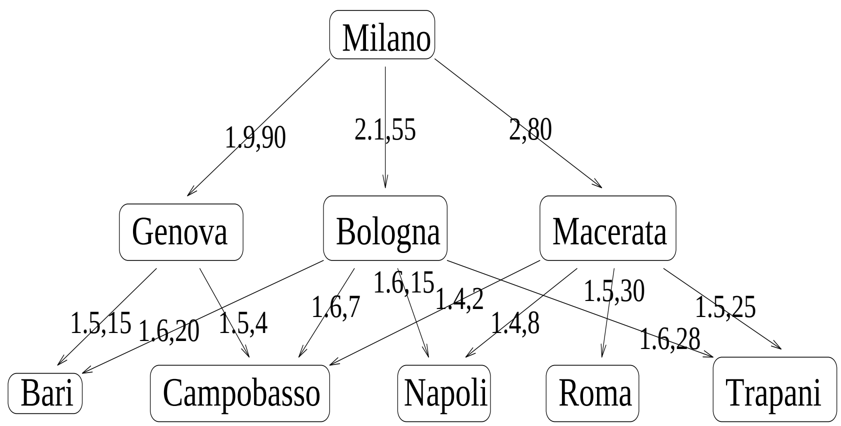
(2) scorgere nella formulazione una “struttura” particolare e risolverlo tenendo conto di questa ulteriore informazione.

Vedremo ora come alcuni problemi “su reti” che appartengono alla categoria (2) possono essere opportunamente formulati.

Problema di trasporti

Si immagini per esempio (cfr. figura sotto) di dover trasportare 80 tonnellate di una determinata merce, da uno stabilimento di produzione (Milano) ai negozi per la libera vendita (Bari, Campobasso, Napoli, Roma, Trapani), passando attraverso 3 magazzini di grossisti (Genova, Bologna, Macerata). La quantità di merce da trasportare da Milano ai negozi è così ripartita (in tonnellate):

	Bari	Campobasso	Napoli	Roma	Trapani
`quantità`	25	8	16	12	19



Non tutte le città sono collegate (cfr. figura), inoltre per ciascun “arco” (strada) che connette 2 città viene indicato: il costo unitario di trasporto ed il quantitativo massimo trasportabile. Si chiede di minimizzare i costi totali di trasporto da Milano alle 5 città ove sono ubicati i negozi. Una possibile formulazione (file `.mod` e `.dat`) potrebbe essere la seguente:

file `network1.mod`

```
set CITTA;   # comprende tutte le citta’ nella figura.
set PERCORSI within {CITTA, CITTA};   # definiamo un sottoinsieme
                                      # delle coppie di citta’
param offerta_domanda {CITTA};   # la componente i-sima di questo
                                 # array di parametri e’ positiva se
                                 # la citta’ i-sima e’ Milano, e’
                                 # nulla se la la citta’ i-sima e’
                                 # Genova, Bologna o Macerata, e’
                                 # infine negativa per le rimanenti
                                 # componenti.
param costi {PERCORSI} >=0;    # indica il costo unitario associato
```

```
                                # a ciascuno dei possibili collega-
                                # menti tra citta'.
param merce_trasportabile {PERCORSI} >=0;   # indica il max quanti-
                                             # tativo di merce tra-
                                             # sportabile per ogni
                                             # collegamento.

var x {(i,j) in PERCORSI} >=0, <= merce_trasportabile[i,j];
                                  # e' il quantitativo di merce che
                                  # alla fine transita su ciascuno
                                  # dei collegamenti possibili.
minimize costi_complessivi:
        sum {(i,j) in PERCORSI} costi[i,j] * x[i,j];

s.t. equilibrio {i in CITTA}:
        sum {(k,i) in PERCORSI} x[k,i] + offerta_domanda[i] =
        sum {(i,j) in PERCORSI} x[i,j];
              # per ogni citta' vi e' un equilibrio tra i quantitativi
              # di merce "entrante" ed "uscente".
```

mentre per il file `network1.dat` si ha l'espressione:

file `network1.dat`

```
set CITTA := Milano
             Genova  Bologna  Macerata
             Bari  Campobasso  Napoli  Roma  Trapani  ;

set PERCORSI := Milano Genova  Milano Bologna  Milano Macerata
                Genova Bari  Genova Campobasso
                Bologna Bari  Bologna Campobasso Bologna Napoli
                Bologna Trapani
                Macerata Campobasso  Macerata Napoli  Macerata Roma
                Macerata Trapani     ;

param      offerta_domanda :=
  Milano        80
  Genova         0
  Bologna        0
  Macerata       0
  Bari            -25
  Campobasso       -8
  Napoli          -16
  Roma            -12
  Trapani         -19      ;

param:            costi       merce_trasportabile :=
    Milano Genova        1.9   90
    Milano Bologna       2.1   55
    Milano Macerata      2     80
    Genova Bari          1.5   15
    Genova Campobasso    1.5    4
    Bologna Bari         1.6   20
```

```
Bologna Campobasso    1.6    7
Bologna Napoli        1.6   15
Bologna Trapani       1.6   28
Macerata Campobasso   1.4    2
Macerata Napoli       1.4    8
Macerata Roma         1.5   30
Macerata Trapani      1.5   25                ;
```

Si noti che i vincoli nel file `network1.mod` rappresentano relazioni di "equilibrio" per ciascuna città , nel senso che la somma dei flussi (frecce in figura) di merci entranti e quelli uscenti devono equivalersi. Quindi la componente i-sima del vettore `offerta_domanda` sarà:

- *positiva* per città da cui le merci escono esclusivamente (Milano);
- *nulla* per le città con grossisti (città di transito Genova, Bologna, Macerata);
- *negativa* per le rimanenti città .

Il modello così come è stato sviluppato, ha lo svantaggio di risolvere il problema senza mostrare esplicitamente una differenziazione di ruoli tra le varie città . Viceversa è possibile evidenziare la struttura "grafica" del problema, identificando ciascuna città come **nodo** e ciascun collegamento tra coppie di città come **arco**. AMPL è in grado di mostrare tale struttura mediante l'introduzione esplicita dei costrutti `node` e `arc`, sostituendoli rispettivamente a quelli di `subject to` e `var`. In pratica a ciascun vincolo di equilibrio viene sostituita una dichiarazione `node` ed a ciascuna variabile si sostituisce un "arco". Quest'ultimo (cfr. figura) viene sostanzialmente identificato dai due nodi estremi da cui diparte ed arriva. Nel modello presentato allora, introducendo queste nuove dichiarazioni, il file `.mod` subisce una modifica mentre rimane inalterato il file `.dat`; il nuovo file `.mod` è dato dal seguente (nel quale si sono omessi i commenti):

file `network2.mod`

```
set CITTA;
set PERCORSI within {CITTA, CITTA};

param offerta_domanda {CITTA};
param costi {PERCORSI} >=0;
param merce_trasportabile {PERCORSI} >=0;

minimize costi_complessivi;

node NODO {i in CITTA}: net_out = offerta_domanda[i];

arc x {(i,j) in PERCORSI} >=0, <= merce_trasportabile[i,j],
      from NODO[i], to NODO[j], obj costi_complessivi costi[i,j];
```

che, relativamente alla sezione di dichiarazione insiemi e parametri, è formalmente identico al file `network1.mod`. Rimarchiamo inoltre il fatto che nel file `network2.mod` la dichiarazione delle variabili `x` è successiva tanto alla dichiarazione della funzione obiettivo quanto a quella dei vincoli; ciò si deve essenzialmente alla necessità di poter utilizzare in generale gli oggetti, solo dopo everli precedentemente definiti.
Per chiarire meglio le differenze con il file `network1.mod`, si osservi che nel file `network2.mod`, della funzione obiettivo è rimasto sostanzialmente il nome, identico a quello definito nel file `network1.mod`. La sua espressione è invece specificata due righe dopo. Nella istruzione successiva vengono dichiarati i nodi della rete; in essa possiamo distinguere le seguenti parti:

- inizia con la parola chiave `node` per indicare che si stanno introducendo nodi di una rete;
- viene inserito il nome del vettore di nodi `NODO`;
- si assegna alla parola chiave `net_out` la differenza tra la quantità di merce entrante e quella di merce uscente dal nodo: questo comunica ad AMPL se per ogni nodo questa differenza è nulla (nodo di transizione Genova, Bologna, Macerata), oppure è un nodo di origine o destinazione (Milano, Bari, Campobasso, Napoli, Roma, Trapani).

Infine nelle ultime due righe del file `network2.mod` vengono introdotte le variabili del problema, quindi queste istruzioni sostituiscono completamente la dichiarazione `var` nel file `network1.mod`; per esse possiamo dire quanto segue:

- cominciano con la parola chiave `arc` per indicare che la variabile `x[i,j]` è associata all'arco `[i,j]`, essendo `i,j` due città dell'insieme `CITTA`;
- poi segue il vincolo bilatero `>=0, <= merce_trasportabile[i,j]` sulla variabile `x[i,j]` (tale vincolo bilatero è in generale opzionale);
- poi viene comunicato ad AMPL chi sono gli estremi dell'arco `[i,j]`, introducendoli con le parole chiave `from` e `to`, separando le informazioni con una virgola;
- infine la sintassi `obj costi_complessivi costi[i,j]` comunica ad AMPL che nella funzione obiettivo `costi_complessivi` l'arco `[i,j]` contribuisce con il termine di costo `costi[i,j]*x[i,j]`, implicitamente definito con la sola quantità `costi[i,j]`.

Concludiamo la sezione aggiungendo che la variabile predefinita `net_out` serve a comunicare ad AMPL se nel nodo vi è o meno equilibrio; essa può essere equivalentemente sostituita con la variabile `net_in`, cambiando però di segno alla differenza successiva.

Contents

References

[1] R. Fourer, D.M. Gay, and B.W. Kernighan, *AMPL a modeling language for mathematical programming*, boyd & fraser publishing company, Massachusetts, 1993