# 4

# *Il linguaggio di modellizzazione algebrica AMPL*

Come ampiamente discusso, l'*approccio modellistico* rappresenta un potente "strumento" per la soluzione di un problema di decisione. In particolare, rappresentare un problema attraverso un modello di Programmazione Matematica permette di utilizzare i numerosi solutori disponibili che sono in grado di risolvere efficientemente problemi anche di dimensioni molto elevate. Tuttavia pur disponendo di efficienti algoritmi di soluzione, sarà necessario trasferire al solutore il problema rappresentato dal modello di Programmazione Matematica. Ovvero è necessario implementare il modello in una qualche forma riconoscibile dai vari solutori. Gli strumenti che svolgono questa funzione vengono chiamati *generatori algebrici di modelli* e, ormai da diversi anni, ne sono stati messi a punto alcuni che hanno avuto una vasta diffusione. Uno di questi è AMPL (acronimo di *A Mathematical Programming Language*) ed è, ad oggi, uno dei più largamente utilizzati per la formulazione di problemi di Programmazione Lineare, Programmazione Lineare Intera, Programmazione Non Lineare e mista. Esso ha la funzione di interfaccia tra il modello algebrico ed il solutore numerico e rappresenta un potente linguaggio di modellazione algebrica, ovvero un linguaggio che contiene diverse primitive per esprimere le notazioni normalmente utilizzate per formulare un problema di Programmazione Matematica: sommatorie, funzioni matematiche elementari, operazioni tra insiemi, etc. Utilizza un'architettura molto avanzata, fornendo un'ottima flessibilità d'uso; infatti, la naturale notazione algebrica utilizzata permette di implementare modelli anche complessi e di dimensioni molto elevate in una forma assai concisa e facilmente comprensibile. È di fatto un linguaggio ad alto livello e non fa uso di particolari strutture dati perché utilizza solamente file di testo sia per l'implementazione del modello sia per la memorizzazione dei dati. In aggiunta, per affermare l'esigenza già ricordata di rendere un modello in-

dipendente dai dati, permette di tenere distinta la struttura del modello dai dati numerici che sono memorizzati separatamente. Inoltre, permette di richiamare al suo interno diversi tra i più efficienti solutori per problemi di Programmazione Matematica di cui attualmente si dispone. È disponibile per i sistemi operativi Windows, Linux e Mc OS X e tutte le informazioni sono reperibili sul sito `http://www.ampl.com`. Sono disponibili la AMPL IDE version e la AMPL Command Line version. Nel seguito, si farà esplicito riferimento a sistemi Windows e alla versione "command line". Gli studenti possono scaricare la Demo version. Tale versione è limitata nel numero delle variabili e dei vincoli. È anche disponibile una versione di prova completa senza limitazioni valida 30 giorni (AMPL Free 30-day trials).

Il testo di riferimento è:

> Robert Fourer, David M. Gay, Brian W. Kernighan. AMPL *A Modeling Language For Mathematical Programming*, Second Edition, Duxbury Thomson, 2003

disponibile sul sito di AMPL al link `http://ampl.com/resources/the-ampl-book`.

Allo stesso link si possono trovare i file di tutti gli esempi riportati nel libro. Sul sito è inoltre disponibile ulteriore materiale riguardante AMPL e i solutori supportati.

## 4.1 INSTALLAZIONE E AVVIO DI AMPL

Per quanto riguarda la versione Windows (64 bit) a riga di comando, andrà scaricato il file `ampl.mswin64.zip` che è un archivio contenente tutti i file necessari. Una volta estratto l'archivio in una directory, saranno create due directory (MODELS e TABLES) che contengono numerosi utili esempi e dei file eseguibili. In particolare il file `ampl.exe` è l'eseguibile di AMPL che chiamato dal prompt di comandi avvierà il programma. Gli altri file eseguibili `baron.exe`, `conopt.exe`, `cplex.exe`, `gurobi.exe`, `ilogcp.exe`, `knitro.exe`, `minos.exe` e `snopt.exe` sono i solutori disponibili. Ciascuno di essi risolve alcune classe di problemi di Programmazione Matematica. Nel seguito riporteremo nel dettaglio le tipologie di problemi risolte dai solutori più comunemente utilizzati.
Si tratta di una versione a linea di comando e quindi deve essere utilizzata dalla finestra del *prompt di comandi*. Quindi aprire tale finestra, portarsi nella directory dove è stato estratto il file `ampl.mswin64.zip` (oppure aggiungere questa directory nel `PATH` di Windows) ed avviare il programma digitando il comando `ampl`. Apparirà così il prompt dei comandi di AMPL:

```
ampl:
```

A questo punto si è nell'ambiente AMPL, ovvero possono essere digitati i comandi di AMPL. Alternativamente, per avviare AMPL si può cliccare due volte su sw.exe (*scrolling-window utility*), digitando poi `ampl` sul prompt che appare.

## 4.2 UN PRIMO ESEMPIO

In linea generale (anche se assai poco pratico) un modello potrebbe essere inserito dal prompt di comandi di AMPL, ovvero digitando, ad esempio, sulla riga di comando i seguenti comandi in successione:

```
ampl: var x1;
ampl: var x2;
ampl: maximize funzione_obiettivo: x1 + x2;
ampl: subject to vincolo1: x1 + x2 <= 1;
ampl: subject to vincolo2: x1 - x2 <= 2;
ampl: subject to x1_non_neg: x1 >= 0;
ampl: subject to x2_non_neg: x2 >= 0;
```

Vedremo più avanti come sia possibile scrivere tali comandi in un file, ma per il momento soffermiamoci sull'analisi dei comandi AMPL utilizzati nell'esempio e sulle parole chiave del linguaggio. Inanzitutto osserviamo che

- ogni comando termina con un "`;`"
- sono presenti due variabili ($x_1$ e $x_2$) e queste sono dichiarate con i comandi `var x1;` e `var x2;`
- la funzione obiettivo è introdotta dalla parola chiave `maximize` in quanto trattasi di un problema di massimizzazione (per problemi di minimizzazione la parola chiave sarà `minimize`). Tale parola chiave è seguita da una etichetta proposta dall'utente (nel caso dell'esempio è `funzione_obiettivo`) seguita da "`:`" che introducono all'espressione della funzione obiettivo
- i vincoli sono elencati di seguito: ciascun vincolo è introdotto dalla parola chiave `subject to` (che può anche esser abbreviata in `s.t.`), seguita dall'etichetta che si vuole dare al vincolo e da "`:`" dopo il quale è riportata l'espressione del vincolo.

Questo semplice esempio ci ha permesso di introdurre la struttura tipica di un modello nel linguaggio AMPL, struttura che ricalca fedelmente quella standard di un modello di Programmazione Matematica come descritto in precedenza (in

particolare, si veda lo schema riportato alla pagina 18). Ovvero, dopo aver introdotto le *variabili di decisione*, deve essere formalizzata la *funzione obiettivo* e i *vincoli*.
Ora che abbiamo il modello implementato in AMPL, naturalmente viene spontaneo domandarsi come può essere determinata la soluzione del problema di ottimizzazione rappresentato dal modello (che nel caso dell'esempio è un problema di Programmazione Lineare). Come già detto, insieme ad AMPL sono disponibili alcuni solutori. In particolare, nella directory dove risiede AMPL sono presenti i file eseguibili di solutori per problemi di Programmazione Lineare, come CPLEX e GUROBI. Ora vogliamo introdurre il comando per selezionare il solutore da utilizzare, ovvero

`option solver` *nome_solutore*;

Quindi digitando:

```
ampl: option solver cplex;
```

stiamo comunicando ad AMPL di voler utilizzare il solutore CPLEX. Quest'ultimo è un solutore per problemi di Programmazione Lineare (e non solo) e può quindi essere utilizzato per risolvere il problema riportato nell'esempio.
A questo punto è sufficiente dare il comando per risolvere il problema che è

`solve;`

Digitando questo comando al promtp dei comandi, ovvero

```
ampl: solve;
```

si ottiene il seguente messaggio:

```
CPLEX 12.8.0.0: optimal solution; objective 1;
0 dual simplex iterations (0 in phase I)
```

con il quale AMPL ci comunica che il solutore ha determinato una soluzione ottima con valore della funzione obiettivo pari a 1. Per vedere il valore delle variabili all'ottimo è necessario un altro comando, ovvero

`display` *nome_variabile*;

Quindi digitando, ad esempio,

```
ampl: display x1;
```

otteniamo il valore di $x_1^*$. Analogamente per l'altra variable $x_2^*$. Abbiamo così ottenuto il punto di massimo che stavamo cercando.

Scrivere il modello utilizzando la linea di comando, ovviamente, è piuttosto scomodo, soprattutto perché in questo modo, una volta usciti da AMPL il modello

è completamente perso. Comviene, pertanto, scrivere il modello in un file testo che deve avere estensione `.mod`. Quindi utilizzando un qualsiasi editor di testo possiamo riscrivere il modello nel seguente modo:

esempio.mod

```
var x1;
var x2;

maximize funzione-obiettivo: x1 + x2;

subject to vincolo1: x1 + x2 <= 1;
subject to vincolo2: x1 - x2 <= 2;
subject to x1_non_neg: x1 >= 0;
subject to x2_non_neg: x2 >= 0;
```

A questo punto, dal prompt di AMPL è sufficiente scrivere il seguente comando per leggere il file del modello:

`model` *<PATH>\nome_file.mod*

Quindi, assumendo che la directory dove risiede il file del modello `esempio.mod` sia *C:\MODELLI*, digitando,

```
ampl: model C:\MODELLI\esempio.mod;
```

carichiamo il modello. A questo punto, si procede come descritto in precedenza per risolvere il problema e stampare i risultati ottenuti.

Si può inoltre creare un file contenente i comandi da dare (in modo da non doverli digitare ogni volta). Tale file deve avere estensione `.run`. Un esempio di questo file relativamente all'esempio fino ad ora esaminato è il seguente:

esempio.run

```
reset;
model C:\MODELLI\esempio.mod;
option solver cplex;
solve;
display x1;
display x2;
display funzione_obiettivo;
```

Si noti che nel file è stato aggiunto il comando `reset;` che ha la funzione di eliminare da AMPL dati relativi ad un modello precedentemente risolto. È conveniente introdurlo all'inizio di ogni file `.run`.
Per lanciare il file `.run` *nell'ambiente* AMPL si utilizza il comando `include esempio.run;`. Alternativamente, *fuori dell'ambiente* AMPL è sufficiente avere tale file nella directory che contiene AMPL (oppure in un'altra directory se si è introdotto nel *PATH* il percorso dove risiede AMPL), aprire un terminale del prompt dei comandi in tale directory e dare il comando `ampl esempio.run`.

Infine, per uscire dall'ambiente AMPL è sufficiente il comando `quit;`.

## 4.3 I SOLUTORI

Abbiamo già menzionato il fatto che alcuni solutori sono disponibili nell'istallazione AMPL della Demo version. Di seguito una breve descrizione di alcuni di essi di uso frequente:

- CPLEX
- GUROBI
- KNITRO
- MINOS.

CPLEX.
Risolve problemi di Programmazione Lineare e problemi di Programmazione Quadratica convessi utilizzando il metodo del simplesso e metodi a punti interni. Inoltre risolve anche problemi di Programmazione Lineare e problemi di Programmazione Quadratica convessi con variabili intere utilizzando procedure di tipo Branch-and-Bound. La versione distribuita con la Demo version di AMPL è limitata a 500 variabili e 500 vincoli.

GUROBI.
Risolve problemi di Programmazione Lineare con il metodo del simplesso e con metodi a punti interni. Risolve inoltre problemi di Programmazione Lineare Misti Interi utilizzanzo procedure di tipo Branch-and-Bound. Risolve inoltre anche problemi di Programmazione Programmazione Quadratica e problemi di Programmazione Quadratica Misti Interi. Utilizando la Demo version di AMPL, GUROBI limita la dimensione dei problemi a 500 variabili e 500 vincoli.

KNITRO.
Risolve problemi di Programmazione Non Lineare, Programmazione Lineare, Programmazione Quadratica (convessa e non convessa) Programmazione Non Lineare mista intera.

MINOS.
Risolve problemi di Programmazione Lineare attraverso il metodo del simplesso e problemi di Programmazione Non Lineare utilizzando metodi di tipo gradiente ridotto.

Esistono altri solutori dei quali è stato previsto l'uso con AMPL (non tutti sono disponibili gratuitamente).

## 4.4 ALCUNI ESEMPI DI MODELLI DI PROGRAMMAZIONE LINEARE

Esaminiamo ora alcuni semplici modelli di Programmazione Lineare costruendo prima la formulazione algebrica e poi realizzandone un'implementazione in AMPL. Iniziamo con il modello dell'Esempio 2.3.1 già visto nel paragrafo 2.3.1.

Si riporta la formulazione finale ottenuta

$$\begin{cases} \max\left(250x_1 + 230x_2 + 110x_3 + 350x_4\right) \\ 2x_1 + 1.5x_2 + 0.5x_3 + 2.5x_4 \leq 100 \\ 0.5x_1 + 0.25x_2 + 0.25x_3 + x_4 \leq 50 \\ x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0 \end{cases}$$

e di seguito il file `.mod` che implementa questo modello.

fertilizzanti.mod

```
var x1;
var x2;
var x3;
var x4;

maximize profitto: 250*x1+230*x2+110*x3+350*x4;

subject to vincolo1: 2*x1+1.5*x2+0.5*x3+2.5*x4 <= 100;
s.t. vincolo2: 0.5*x1+0.25*x2+0.25*x3+x4<= 50;
s.t. x1_nonneg: x1 >=0;
s.t. x2_nonneg: x2 >=0;
s.t. x3_nonneg: x3 >=0;
s.t. x4_nonneg: x4 >=0;
```

Esaminiamo ora un altro modello già visto, ed in particolare quello riguardante la produzione multi-plant dell'Esempio 3.4.5

Si riportano sinteticamente le formulazioni nei due casi.

**Formulazione del caso (a)**
Questo caso, nella pratica, corrisponde a costruire due modelli indipendenti: uno riferito al primo impianto, uno riferito al secondo impianto. Una "risorsa" (il materiale grezzo) è già allocata a priori.

Impianto 1: La formulazione relativa al primo impianto è:

$$\begin{cases} \max(10x_1 + 15x_2) \\ 4x_1 + 4x_2 \leq 75 \\ 4x_1 + 2x_2 \leq 80 \\ 2x_1 + 5x_2 \leq 60 \\ x_1 \geq 0, x_2 \geq 0 \end{cases}$$

A questo punto, prima di passare all'impianto 2, vediamo come fare per scrivere in AMPL il precedente problema di PL. Per fare questo è sufficiente creare il seguente file di modello (un semplice file di testo ma con estensione `.mod`) ad esempio `impianto1.mod`.

impianto1.mod

```
var x1;
var x2;

maximize profitto: 10*x1 + 15*x2;

subject to   m_grezzo: 4*x1 + 4*x2 <= 75;
subject to levigatura: 4*x1 + 2*x2 <= 80;
s.t.         pulitura: 2*x1 + 5*x2 <= 60;
s.t.         x1_non_neg: x1 >= 0;
s.t.         x2_non_neg: x2 >= 0;
```

Risolvendo il problema, si ottiene che all'ottimo l'impianto 1 ha un profitto di 225 Euro, ottenuto fabbricando 11.25 unità di $\mathbf{P_1}$ e 7.5 di $\mathbf{P_2}$.

Impianto 2: La formulazione relativa al secondo impianto è:

$$\begin{cases} \max(10x_3 + 15x_4) \\ 4x_3 + 4x_4 \leq 45 \\ 5x_3 + 3x_4 \leq 60 \\ 5x_3 + 6x_4 \leq 75 \\ x_3 \geq 0, x_4 \geq 0 \end{cases}$$

Il modello in AMPL per l'impianto 2 è:

impianto2.mod

```
var x3;
var x4;

maximize profitto: 10*x3 + 15*x4;

subject to   m_grezzo: 4*x3 + 4*x4 <= 45;
subject to levigatura: 5*x3 + 3*x4 <= 60;
s.t.         pulitura: 5*x3 + 6*x4 <= 75;
s.t.         x3_non_neg: x3 >= 0;
s.t.         x4_non_neg: x4 >= 0;
```

Risolvendo il problema si ha che all'ottimo l'impianto 2 ha un profitto di 168.75 Euro ottenuto fabbricando 11.25 unità di $\mathbf{P_2}$ e nessuna unità di $\mathbf{P_1}$.

**Formulazione del caso (b)**
Questo caso corrisponde a costruire un unico modello comprendente entrambi gli impianti. L'allocazione della "risorsa" data dal materiale grezzo è lasciata al modello stesso.

La formulazione relativa a questo caso è:

$$\begin{cases} \max & (10x_1 & + & 15x_2 & + & 10x_3 & + & 15x_4) & & \\ & 4x_1 & + & 4x_2 & + & 4x_3 & + & 4x_4 & \leq & 120 \\ & 4x_1 & + & 2x_2 & & & & & \leq & 80 \\ & 2x_1 & + & 5x_2 & & & & & \leq & 60 \\ & & & & & 5x_3 & + & 3x_4 & \leq & 60 \\ & & & & & 5x_3 & + & 6x_4 & \leq & 75 \\ & x_1 \geq 0, & & x_2 \geq 0, & & x_3 \geq 0, & & x_4 \geq 0 & & \end{cases}$$

e in AMPL si avrà:

impianto1e2.mod

```
var x1;
var x2;
var x3;
var x4;

maximize profitto: 10*(x1+x3) + 15*(x2+x4);
```

```
s.t. m_grezzoTOT:   4*(x1+x2+x3+x4) <= 120;
s.t. levigatura1:   4*x1 + 2*x2 <= 80;
s.t. pulitura1:     2*x1 + 5*x2 <= 60;
s.t. levigatura2:   5*x3 + 3*x4 <= 60;
s.t. pulitura2:     5*x3 + 6*x4 <= 75;

s.t. x1_non_neg:        x1 >= 0;
s.t. x2_non_neg:        x2 >= 0;
s.t. x3_non_neg:        x3 >= 0;
s.t. x4_non_neg:        x4 >= 0;
```

Risolvere questo problema otteniamo un profitto complessivo di 404.17 Euro ottenuto fabbricando nell'impianto 1: 9.17 unità di $\mathbf{P_1}$ e 8.33 unità di $\mathbf{P_2}$ e fabbricando, nell'impianto 2, solo 12.5 unità di $\mathbf{P_2}$. Come abbiamo già visto, è importante notare che il profitto totale in questo caso (404.17 Euro) è superiore alla somma dei profitti ottenuti considerando separatamente i due modelli e che l'impianto 1 ora usa 70Kg di materiale grezzo (contro i 75Kg del caso precedente) mentre l'impianto 2 ne usa 50Kg (contro i 45Kg del caso precedente).

## 4.5 GLI INSIEMI E I PARAMETRI IN AMPL

AMPL permette di scrivere il modello in forma parametrica. Questo, nella sostanza, significa scrivere il modello nel file `.mod` senza specificare i dati che vengono invece scritti separatamente in un file `.dat`. Praticamente nel file dei dati si effettuano le *dichiarazioni*, mentre nel file dei dati si effettuano le *assegnazioni*. Fatto questo, dal prompt di AMPL oltre al comando

`model` $<PATH> \setminus nome\_file.mod$

utilizzato per caricare il modello, sarà necessario un comando per caricare il file dei dati. Tale comando è:

`data` $<PATH> \setminus nome\_file.dat$

Quindi, assumendo che la directory dove risiedono i file del modello (`esempio.mod`) e dei dati (`esempio.dat`) sia $C:\setminus MODELLI$, digitando anche

```
ampl: data C:\MODELLI\esempio.dat;
```

carichiamo anche i dati del modello. A questo punto, si procede come descritto in precedenza per risolvere il problema e stampare i risultati ottenuti.

Per introdurre l'uso di insiemi e parametri, consideriamo di nuovo i modelli presentati nel paragrafo precedente riguardanti i due impianti di produzione. Infatti, AMPL consente di scrivere il problema in modo diverso, utilizzando un concetto molto utilizzato in AMPL: gli *insiemi*. Grazie ad esso, è possibile tenere separati il modello dai dati. Il modello relativo all'impianto 1 può essere infatti riscritto come segue:

impianto.mod

```
set Prodotti;
set Risorse;

param q_max{Risorse} >=0;
param richiesta{Risorse,Prodotti} >=0;
param prezzo{Prodotti} >=0;

var x{Prodotti} >=0;

maximize profitto: sum{i in Prodotti} prezzo[i]*x[i];
```

```
s.t. vincolo_risorsa {j in Risorse}:
sum{i in Prodotti} richiesta[j,i]*x[i] <= q_max[j];
```

---

Ora analiziamo le istruzione del file `impianto.mod`. Anzitutto notiamo le istruzioni

```
set Prodotti;
set Risorse;
```

con le quali si dichiarano `Prodotti` e `Risorse` come insiemi di un numero imprecisato di elementi (prodotti e risorse). Subito dopo abbiamo tre istruzioni che ci servono per definire altrettanti parametri del modello. La prima di queste

```
param q_max{Risorse}>=0;
```

definisce un vettore di parametri con tante componenti quanti sono gli elementi dell'insieme `Risorse` e definisce le quantità massime disponibili di ciascuna risorsa. Il $>= 0$ specifica che i valori immessi devono essere non negativi, AMPL eseguirà automaticamente il controllo sui dati.
L'istruzione successiva ovvero

```
param richiesta{Risorse,Prodotti}>=0;
```

definisce invece una matrice di parametri con tante righe quanti sono le risorse e tante colonne quante sono i prodotti, specificando per ogni prodotto la quantità di risorsa necessaria alla sua realizzazione.

L' istruzione

```
param prezzo{Prodotti} >=0;
```

definisce un vettore di parametri con tante componenti quanti sono gli elementi dell'insieme `Prodotti` specificando il prezzo di ogni prodotto.

La funzione obiettivo `profitto` è definita dalla istruzione

```
maximize profitto: sum{i in Prodotti} prezzo[i]*x[i];
```

I vincoli sono definiti atttraverso l'istruzione che segue:

```
s.t. vincolo_risorsa {j in Risorse}:
     sum{i in Prodotti} richiesta[j,i]*x[i] <= q_max[j];
```

Con essa si definiscono tanti vincoli quanti sono gli elementi dell'insieme `Risorse`.

A questo punto, completato il file della definizione del modello ( `.mod`) è necessario definire un file con estensione `.dat` dove vengono specificati i valori per gli insiemi e i parametri del modello. Per quanto riguarda l'impianto 1 si avrà:

impianto1.dat

```
set Prodotti := P1 P2;
set Risorse  := m_grezzo levigatura pulitura;

param         q_max :=
m_grezzo        75
levigatura      80
pulitura        60;

param  richiesta: P1   P2 :=
m_grezzo          4     4
levigatura        4     2
pulitura          2     5;

param  prezzo :=
P1       10
P2       15;
```

Facciamo notare che in questo modo, avendo cioè a disposizione il file `impianto.mod` contenente il modello separato dai dati del problema (contenuti invece nel file `impianto1.dat`), possiamo risolvere differenti problemi di massimizzazione del profitto semplicemente cambiando i dati contenuti nel file con estensione `.dat`. Sfruttando questo fatto è banale risolvere il problema relativo all'impianto 2: si utilizza lo stesso file `.mod` e un diverso file `.dat`, che è il seguente:

impianto2.dat

```
set Prodotti := P1 P2;
set Risorse  := m_grezzo levigatura pulitura;

param         q_max :=
m_grezzo        45
levigatura      60
pulitura        75;


param  richiesta: P1   P2 :=
m_grezzo          4     4
levigatura        5     3
pulitura          5     6;
```

```
param  prezzo :=
P1       10
P2       15;
```

Nello stesso modo, cioè specificando il solo file dei dati, è possibile risolvere il problema completo:

impianto.dat

```
set Prodotti :=
       P1_impianto1
       P2_impianto1
       P1_impianto2
       P2_impianto2;

set Risorse :=
       m_grezzo
       levigatura1
       pulitura1
       levigatura2
       pulitura2;

param         q_max :=
      m_grezzo         120
      levigatura1      80
      pulitura1        60
      levigatura2      60
      pulitura2        75;


param  richiesta: P1_impianto1   P2_impianto1  P1_impianto2  P2_impianto2:=
 m_grezzo          4                4             4               4
 levigatura1       4                2             0               0
 pulitura1         2                5             0               0
 levigatura2       0                0             5               3
 pulitura2         0                0             5               6;
```

```
param          prezzo :=
        P1_impianto1       10
        P2_impianto1       15
        P1_impianto2       10
        P2_impianto2       15;
```

---

Si può ovviamente modificare il file `.run` scritto in precedenza aggiungendo l'istruzione necessaria per caricare il file dei dati.

È importante ribadire che nel file `.mod` vengono effettuate le *dichiarazioni* di insiemi e parametri, mentre nel file `.dat` vengono effettuate le *assegnazioni*.

Esaminiamo nel dettaglio nel paragrafi che seguono gli elementi base della sintassi di AMPL riguardante *insiemi*, *parametri*, *variabili*, *funzione obiettivo* e *vincoli*.

### 4.5.1 Gli insiemi

Gli insiemi definiscono gli elementi di base con i quali si possono indicizzare variabili, parametri e vincoli del modello.

La dichiarazione di un *insieme generico* (nel file `.mod`) si effettua come segue:

```
set NomeInsieme;
```

L'assegnazione di un *insieme generico* (nel file `.dat`) si effettua come segue:

```
set NomeInsieme := e1 e2 e3 e4 e5 ;
```

Questa istruzione specifica che l'insieme `NomeInsieme` è composto dagli elementi $\{e_1, e_2, e_3, e_4, e_5\}$. Gli insiemi così definiti corrispondono agli insiemi generici. Esiste la possibilità di definire altri tipi di insiemi:

- *insiemi ordinati*

  ```
  set NomeInsieme ordered;
  ```

- *insiemi numerici*

  ```
  set NomeInsieme := 1 .. N;
  set NomeInsieme := 1 .. N by p;
  ```

- *insiemi ordinati e ciclici*

  ```
  set NomeInsieme circular;
  ```

Per quanto riguarda gli insiemi numerici la notazione `NomeInsieme := 1 .. N` definisce tutti i numeri interi tra 1 e $N$, mentre `NomeInsieme := 1 .. N by`

`p` definisice tutti i numeri interi tra 1 e $N$ distanti fra di loro di $p$ numeri. Si osservi, quindi, che per quanto riguarda gli insiemi numerici la dichiarazione di fatto è un'assegnazione e quindi l'assegnazione di un insieme numerico *non* deve essere ripetuta nel file `.dat`.

Riportiamo ora gli operatori e le funzioni più comuni tra due **insiemi generici**:

| **Operatore/Funzione** | **Significato** |
|---|---|
| `A union B` | insieme di elementi che stanno in $A$ o $B$ |
| `A inter B` | insieme di elementi che stanno sia in $A$ che in $B$ |
| `A within B` | $A$ sottoinsieme di $B$ |
| `A diff B` | insieme di elementi che stanno in $A$ ma non in $B$ |
| `A symdiff B` | insieme di elementi che stanno in $A$ o in $B$ ma non in entrambi |
| `card(A)` | numero di elementi che stanno in $A$ |

Di seguito gli operatori e le funzioni più comuni tra due **insiemi ordinati**:

| **Funzione** | **Significato** |
|---|---|
| `first(A)` | primo elemento di $A$ |
| `last(A)` | ultimo elemento di $A$ |
| `next(a,A)` | elemento di $A$ dopo $a$ |
| `prev(a,A)` | elemento di $A$ prima di $a$ |
| `next(a,A,k)` | $k$-esimo elemento di $A$ dopo $a$ |
| `prev(a,A,k)` | $k$-esimo elemento di $A$ prima di $a$ |
| `ord(a,A)` | posizione di $a$ in $A$ |
| `ord0(a,A)` | come `ord(a,A)` ma restituisce 0 se $a$ non è in $A$ |
| `member(k,A)` | elemento di $A$ in $k$-esima posizione |

*Gli insiemi multidimensionali*

In AMPL è possibile definire insiemi a più dimensioni. Tale dimesione deve essere specificata nella dichiarazione di insieme (file `.mod`). La dicharazione

```
set INSIEME dimension p;
```

si usa per indicare che l'insieme `INSIEME` è costituito da $p-$uple *ordinate*, ovvero i suoi elementi sono della forma $(a_1, a_2, \ldots, a_p)$. Un esempio potrebbe essere:

```
set TRIPLE dimension 3;
```

che indica che l'insieme `TRIPLE` è formato da tutte triple ordinate di valori che verranno poi specificate nel file `.dat`, ad esempio, nel seguente modo:

```
set TRIPLE := (a,b,c) (d,e,f) (g,h,i) (l,m.n);
```

oppure

```
set TRIPLE :=
a b c
d e f
g h i
l m n;
```

Un modo alternativo di ottenere insiemi multidimensionali di dimensione $p$ è l'utilizzazione del prodotto cartesiano di $p$ insiemi. Quindi se, ad esempio, $A$, $B$ e $C$ sono stati già dichiarati come insiemi, l'istruzione

```
set TRIPLE := A cross B cross C;
```

indica l'insieme delle triple ordinate $(a, b, c)$ con $a \in A$, $b \in B$ e $c \in C$.

Partendo da un insieme multidimensionale è possibile ottenere gli insiemi componenti l'insieme multidimensionale effettuando un procedimento inverso al precedente. Con riferimento all'esempio precedente, dall'insieme `TRIPLE` si possono ottenere i tre insiemi di partenza nel seguente modo con l'uso di `setof`:

```
set A := setof{(i,j,k) in TRIPLE} i;
set B := setof{(i,j,k) in TRIPLE} j;
set C := setof{(i,j,k) in TRIPLE} k;
```

Oppure si può ottenere l'insieme delle coppie ordinate $(a, b)$, $a \in A$, $b \in B$, a partire dall'insieme `TRIPLE` nel seguente modo:

```
set COPPIE := setof{(i,j,k) in TRIPLE} (i,j);
```

È inoltre possibile definire insiemi in modo implicito, ovvero senza specificarne il nome, ma solo indicando la loro espressione. Alcuni esempi sono i seguenti: dati $A$ e $B$ due insiemi,

```
{A,B}
{a in A, b in B}
A cross B
```

sono espressioni equivalenti per definire il prodotto cartesiano $A \times B$, ovvero l'insieme di tutte le coppie ordinate $(a, b)$ con $a \in A$ e $b \in B$. Un altro esempio è dato dall'espressione

```
{a in A : prezzo[a]>=5}
```

che sta ad indicare tutti gli elementi $a$ dell'insieme $A$ tali che il `prezzo` di $a$ sia maggiore o uguale a 5, dove, come vedremo più avanti, `prezzo` è un parametro indicizzato sull'insieme $A$.

### 4.5.2 I parametri

I parametri rappresentano i dati di un problema. Una volta che essi vengono assegnati (nel file `.dat`), essi non vengono in nessun caso modificati dal solutore. Un parametro deve essere *dichiarato* nel file `.mod` e *assegnato* nel file `.dat`. L'istruzione

```
param t;
```

utilizzata nel file `.mod` effettua la dichiarazione del parametro $t$. È possibile anche dichiarare vettori e matrici di parametri. Quindi se `PROD` e `ZONA` sono due insiemi le istruzioni

```
set PROD;
set ZONA;
param T;
param costi{PROD};
param prezzo{PROD,ZONA};
param domanda{PROD,ZONA,1..T};
```

oltre che dichiarare gli insiemi, definiscono:

- il parametro `T`;
- il parametro `costi` indicizzato dall'insieme `PROD`, ovvero `costi` è un vettore che ha tante componenti quanti sono gli elementi di `PROD`;
- il parametro a due dimensioni `prezzo`, indicizzato dagli insiemi `PROD` e `ZONA`;
- il parametro a tre dimensioni `domanda`, indicizzato dagli `PROD`, `ZONA` e dall'insieme dei numeri interi che vanno da 1 a $T$.

L'assegnazione dei parametri avviene nel file `.dat`. Un esempio di assegnazione dei parametri prima introdotti è il seguente:

```
set PROD := p1 p2;
set ZONA := z1 z2;
param T := 2;

param costi :=
   p1 5
   p2 4;

param prezzo: z1    z2  :=
   p1          2     7
   p2          5     9;
```

```
param domanda:=
  [*,*,1] :   z1    z2:=
     p1       10    15
     p2       13    22

  [*,*,2] :   z1    z2:=
     p1       32    25
     p2       18    15;
```

Si osservi innanzitutto, che la scelta di scrivere i dati incolonnati ha il solo scopo di rendere il file più leggibile: nessuna formattazione particolare è richiesta da AMPL. Inoltre è opportuno soffermarci sull'uso dei ":"; infatti i ":" sono obbligatori se si assegnano valori a due o più vettori di parametri monodimensionali indicizzati sullo stesso insieme, come nel caso di `prezzo` e `domanda`.

Il parametro `domanda` può essere alternativamente assegnato nel seguente modo:

```
param : domanda :=
   p1  z1  1  10
   p1  z1  2  32
   p1  z2  1  15
   p1  z2  2  25
   p2  z1  1  13
   p2  z1  2  18
   p2  z2  1  22
   p2  z2  2  15;
```

Nella dichiarazione dei parametri (file `.mod`) si possono anche includere controlli o restrizioni sui parametri stessi. Ad esempio, le istruzioni

```
param T > 0;
param N integer, <= T;
```

controllano che il parametro $T$ sia positivo e che il parametro $N$ sia un numero intero minore o uguale a $T$. Esiste anche l'istruzione `check` per effettuare controlli contenenti espressioni logiche. Un esempio potrebbe essere il seguente:

```
set PROD;
param offertatot >0;
param offerta{PROD} >0;
check: sum{p in PROD} offerta[p]=offertatot;
```

Esiste, infine, la possibilità di dichiarare parametri "calcolati" come nel seguente esempio:

```
set PROD;
param offerta{PROD};
param offertatot:=sum{p in PROD} offerta[p];
```

### 4.5.3 Le variabili

Le variabili rappresentano le incognite del problema e il loro valore è calcolato dal solutore. Una volta che la soluzione è stata determinta, il valore delle variabili all'ottimo rappresenta la soluzione del problema Si osservi, quindi, la differenza con i parametri: questi ultimi vengono assegnati dall'utente e rimangono costanti; il valore delle variabili cambia durante le iterazioni del solutore. Alle variabili l'utente può eventualmemte (ma è del tutto opzionale) assegnare dei valori inziali che sono poi modificati dal solutore.

La dichiarazione delle variabili è obbligatoria. Per default, una variabile è considerata *reale*, oppure può essere specificata *intera* o *binaria* (a valori in $\{0, 1\}$). L'esempio che segue riporta la dichiarazione di variabili con la specifica del tipo di variabili:

```
var x;
var n integer;
var d binary;
```

Analogamente a quanto avviene per i parametri, anche le variabili possono essere indicizzate da insiemi come riportato nel seguente esempio:

```
set PROD;
set OPERAI;
param dom{PROD};
var num{PROD} integer;
var assegnamento{PROD,OPERAI} binary;
```

Anche nel caso delle variabili si possono introdurre dei controlli contestualmente alla loro dichiarazione come riportato nell'esempio seguente:

```
set PROD;
param dom{PROD};
var x >=0;
var quantita{p in PROD} >=0, <= dom[p];
```

È possibile *fissare* una variabile ad un valore mediante l'istruzione `fix` che può essere utilizzata nel file `.dat` o nel file `.run` e NON nel file `.mod`. Quindi data la variabile $x$, l'istruzione

```
fix x :=4;
```

assegna il valore 4 ad $x$. In questo caso il solutore *non cambierà il valore di tale variabile* che verrà considerata fissata al valore assegnato. Esiste poi il comando opposto `unfix` che sblocca una variabile precedentemente fissata. Nel caso dell'esempio si avrebbe

```
unfix x;
```

Infine, c'è la possibilità di inizializzare una variabile ad un determinato valore con il comando `let` da utilizzare nel file `.dat` o nel file `.run` e NON nel file `.mod`. Se scriviamo

```
let x:= 10;
```

stiamo inizializzando la variabile $x$ al valore 10. Questo vuole dire che l'algoritmo risolutivo assegnerà 10 come valore iniziale della variabile $x$, valore che sarà cambiato dal solutore nel corso delle sue iterazioni. Si osservi, quindi, la differenza fondamentale tra il "fixing" di una variabile che non permette di modificare il valore assegnato a quella variabile e l'assegnazione di un valore iniziale ad una variabile.

### 4.5.4 La funzione obiettivo e i vincoli

La funzione obiettivo è sempre presente in un modello di Programmazione Matematica e rappresenta ciò che vogliamo massimizzare o minimizzare. Essa deve essere specificata nel file del modello (`.mod`). La parola chiave del linguaggo AMPL per introdurre la funzione obiettivo è `minimize` o `maximize` a seconda che ci trovi di fronte ad un problema di minimizzazione o massimizzazione. La sintassi è

```
maximize nome_funzione_obiettivo :  espressione_aritmetica ;
```

oppure

```
minimize nome_funzione_obiettivo :  espressione_aritmetica ;
```

Quindi un esempio potrebbe essere:

```
maximize profitto_totale : sum{i in PROD} prezzo[i]*quantita[i];
```

I vincoli sono parte integrante di un modello di Programmazione Matematica e sono specificati anch'essi nel file del modello (`.mod`). La parola chiave è `subject to` che può essere abbreviata in `s.t.`. La sintassi è

```
subject to nome_vincolo :  espressione_aritmetica e/o logica;
```

Se $x$ è una variabile reale, la più semplice espressione di un vincolo potrebbe essere

```
subject to vincolo : x >=0;
```

In realtà, questo tipo di vincoli semplici viene spesso inserito come restrzione nella dichiarazione della variabile $x$. Anche i vincoli possono essere indicizzati e quindi invece di scrivere tanti vincoli, in una sola espressione si possono esplicitare più vincoli. Un esempio di ciò è il seguente:

```
set PROD;
set REPARTI;

param orelavoro{PROD,REPARTI};
param maxore{REPARTI};
param prezzo{PROD};

var x{PROD};

maximize ricavo : sum{in PROD} prezzo[i]*x[i];

subject to vincoli{j in REPARTI} sum{i in PROD}
                        orelavoro[i,j]*x[i] <= maxore[j];
```

In questo esempio, con un unico vincolo si sono scritti tanti vincoli quanti sono gli elementi dell'insieme `RISORSE` al posto dei vincoli

```
subject to vincolo_1 : sum{i in PROD} orelavoro[i,"R1"]*x[i]
                            <= maxore["R1"];
subject to vincolo_2 : sum{i in PROD} orelavoro[i,"R2"]*x[i]
                            <= maxore["R2"];
   .
   .
   .
   .
subject to vincolo_m : sum{i in PROD} orelavoro[i,"Rm"]*x[i]
                            <= maxore["Rm"];
```

avendo supposto che l'insieme generico `REPARTI` (assegnato nel file `.dat`) sia composto dagli elementi $\{R1, R2, \ldots, Rm\}$. Quest'ultima scrittura estesa vincolo per vincolo non darebbe luogo a messaggi di errore in AMPL, ma oltre che essere molto lunga presenta l'ovvio inconveniente di dover conoscere già nel file del modello (`.mod`) quali sono gli elementi dell'insieme `REPARTI`. Questo contravviene al fatto che un modello deve essere indipendente dai dati; infatti, non utilizzando

la scrittura indicizzata dei vincoli, un cambio degli elementi dell'insieme `REPARTI` (che ricordiamo è assegnato nel file dei dati (`.dat`) ) non permetterebbe più al modello implementato di essere corretto.

### 4.5.5 Le espressioni

Nella costruzione della funzione obiettivo e dei vincoli, così come nell'imporre condizioni sui parametri e sulle variabili si utilizzano espressioni aritmetiche, funzioni e operatori di diverso tipo. Abbiamo già riportato in precedenza i principali operatori e funzioni sugli insiemi. Le principali funzioni e operatori aritmetici e logici sono riportati nelle tabelle che seguono (per una elenco completo si fa riferimento al testo di AMPL già citato.

| **Funzione** | **Significato** |
|---|---|
| `abs(x)` | valore assoluto di $x$ |
| `sin(x)` | $\sin(x)$ |
| `cos(x)` | $\cos(x)$ |
| `tan(x)` | $\tan(x)$ |
| `asin(x)` | $\arcsin(x)$ |
| `acos(x)` | $\arccos(x)$ |
| `atan(x)` | $\arctan(x)$ |
| `exp(x)` | $\exp(x)$ |
| `sqrt(x)` | radice quadrata di $x$, $\sqrt{x}$ |
| `log(x)` | logaritmo naturale di $x$, $\ln(x)$ |
| `log10(x)` | logaritmo in base 10 di $x$, $\log(x)$ |
| `ceil(x)` | parte intera superiore di $x$, $\lceil x \rceil$ |
| `floor(x)` | parte intera inferiore di $x$, $\lfloor x \rfloor$ |

| Operatori aritmetici | Significato |
|---|---|
| ^ | potenza |
| + | somma |
| - | sottrazione |
| * | prodotto |
| / | divisione |
| div | divisione intera |
| mod | modulo |
| sum | sommatoria |
| prod | produttoria |
| min | minimo |
| max | massimo |
| >, >= | maggiore, maggiore o uguale |
| <, <= | minore, minore o uguale |
| = | = |
| <>, != | diverso |

| Operatori logici | Significato |
|---|---|
| not | negazione logica |
| or | "or" logico |
| and | "and" logico |
| exists | quantificatore esistenziale logico |
| forall | quantificatore universale logico |
| if then else | espressione condizionale |

### 4.5.6 Due esempi di modelli di Programmazione Lineare

*Un problema di pianificazione dei trasporti*

**Esempio 4.5.1** *Si devono pianificare i trasporti di un tipo di merce da cinque città, Asti, Bergamo, Como, Domodossola, Lecce (città origini) ad altre quattro città, Mantova, Napoli, Olbia, Palermo (città destinazioni). La tabella che segue riporta il costo unitario del traporto della merce da ciascuna città origine a ciascuna città destinazione, insieme alla domanda di merce da soddisfare esattamente di ogni città destinazione e alla quantità massima di merce disponibile presso ciascuna città origine*

| | Mantova | Napoli | Olbia | Palermo | *disponibilità max* |
|---|---|---|---|---|---|
| Asti | 1 | 3.5 | 4 | 4.5 | 100 |
| Bergamo | 0.1 | 3 | 4.5 | 4.8 | 110 |
| Como | 0.3 | 2.8 | 4.7 | 4.9 | 130 |
| Domodossola | 1 | 2.2 | 3.9 | 4 | 85 |
| Lecce | 1.7 | 2.2 | 5 | 5.3 | 120 |
| *domanda* | 30 | 18 | 45 | 56 | |

*Come si può osservare, la somma dei quantitativi di merce disponibili nelle città origini è maggiore alla somma delle domande delle città destinazione e questo perché nelle città orgini sono disponibili dei depositi dove immagazzinare la merce. Si devono pianificare i trasporti dalle città origini alle città destinazioni in modo da minizzare il costo totale dei trasporti.*

**Formulazione**

Introdotte le varibili di decisione $x_{ij}$, $i = 1, 2, 3, 4, 5$, $j = 1, 2, 3, 4$, associate alla quantità di merce da trasportare dalla città origine $i$-esima alla città destinazione $j$-esima, indicando con $c_{ij}$ il costo unitario del trasporto dalla città origine $i$-esima alla città destinazione $j$-esima e indicando rispettivamente con $a_i$, $i = 1, 2, 3, 4, 5$ e $b_j$, $j = 1, 2, 3, 4$ la disponibilità massima di merce nelle origini e la domanda di merce nelle destinazioni, un modello lineare che rappresenta il problema in analisi è il seguente:

$$\begin{cases} \min \left( \displaystyle\sum_{i=1}^{5} \sum_{j=1}^{4} c_{ij} x_{ij} \right) \\ \displaystyle\sum_{j=1}^{4} x_{ij} \leq a_i \qquad i = 1, \ldots, 5 \\ \displaystyle\sum_{i=1}^{5} x_{ij} = b_j \qquad j = 1, \ldots, 4 \\ x_{ij} \geq 0 \end{cases}$$

Segue il file `trasporto1.mod` che realizza un'implementazione in AMPL del modello ora costruito.

trasporto1.mod

```
####    SEZIONE PER LA DICHIARAZIONE DEGLI INSIEMI    ####

set ORIGINI;         # introduce l'insieme delle origini
set DESTINAZIONI;    # introduce l'insieme delle destinazioni


####    SEZIONE PER LA DICHIARAZIONE DEI PARAMETRI    ####

param  costo_origdest {ORIGINI,DESTINAZIONI} >= 0
                           # vettore di parametri a 2
                           # indici (matrice). Rappresenta
                           # i costi unitari di trasporto
                           # (non negativi) dalle origini
                           # alle destinazioni.

param  max_uscita {ORIGINI} >= 0;
                           # vettore di parametri ad un
                           # indice. Rappresenta la quantita'
                           # (non negativa) massima di merce
                           # che puo' essere trasportata da
                           # ciascuna origine.

param  domanda {DESTINAZIONI} >=0;
                           # vettore di parametri ad un
                           # indice. Rappresenta la quantita'
                           # (non negativa) di merce che deve
                           # essere trasportata a ciascuna
                           # destinazione.

####   SEZIONE PER LA DICHIARAZIONE DELLE VARIABILI  ####

var x {i in ORIGINI,j in DESTINAZIONI} >= 0;
                     # x[i,j] e' l'elemento di una matrice di
                     # variabili (2 indici) e rappresenta il
                     # quantitativo di merce trasportato dalla
                     # origine 'i' alla destinazione 'j'.
```

```
####  SEZIONE PER LA DICHIARAZIONE DELLA FUNZIONE ####
####              OBIETTIVO E DEI VINCOLI          ####

minimize cost_trasporto:  sum{i in ORIGINI,j in DESTINAZIONI}
                          costo_origdest[i,j]*x[i,j];

                  # a ciascun trasporto origine/destinazione
                  # e' associato un costo, i costi poi vengono
                  # sommati

s.t. origini {i in ORIGINI}:
     sum{j in DESTINAZIONI} x[i,j] <= max_uscita[i] ;
              # da ciascuna origine non e' possibile inviare
              # piu' di un determinato quantitativo di merce.


s.t. destinazioni {j in DESTINAZIONI}:
     sum{i in ORIGINI} x[i,j] = domanda[j] ;
          # a ciascuna destinazione deve arrivare
          # esattamente una quantita' determinata di merce.
```

Il file `trasporto1.dat` che permette di specificare i dati da introdurre nel modello precedente è il seguente

trasporto1.dat

```
set ORIGINI := asti bergamo como domodossola lecce;
                     # l'insieme ORIGINI ha 5 elementi
set DESTINAZIONI := mantova napoli olbia palermo;
                     # l'insieme DESTINAZIONI ha 4 elementi

param            max_uscita :=
     asti             100
     bergamo          110
     como             130
     domodossola       85
     lecce            120       ;
                     # il vettore di parametri 'max_uscita'
                     # ha 5 elementi (tanti quante le origini).
```

```
param               domanda   :=
     mantova           30
     napoli            18
     olbia             45
     palermo           56        ;
                    # il vettore di parametri 'domanda'
                    # ha 4 elementi (tanti quante le  destina-
                    # zioni). Si noti che quando si assegnano
                    # gli elementi di un solo vettore non sono
                    # necessari i ':' dopo la parola chiave
                    # 'param'


param costo_origdest : mantova  napoli  olbia  palermo :=
          asti            1       3.5     4        4.5
          bergamo          .1     3       4.5      4.8
          como             .3     2.8     4.7      4.9
          domodossola     1       2.2     3.9      4.0
          lecce           1.7     2.2     5.0      5.3   ;
           # la matrice di parametri 'costo_origdest' contiene
           # 4*5 = 20 elementi, uno per ogni coppia origine/de-
           # stinazione; non e' necessario specificare
           # lo '0' (zero) davanti alla virgola.
```

---

Supponiamo ora che nelle origini ci sia un costo di prelevamento della merce di cui tenere conto nel calcolo del costo totale, aggiuntivo al costo dei trasporti. Nella tabella che segue si riporta i costi unitari di prelevamento da ciascuna città origine.

| | Asti | Bergamo | Como | Domodossola | Lecce |
|---|---|---|---|---|---|
| *costo prelevamento* | 2 | 3 | 4 | 7 | 6 |

Modificare il file `.mod` e il file `.dat` in modo da tener conto di questi costi aggiuntivi.

Chiamati $c_i$, $i = 1, 2, 3, 4, 5$, i costi unitari di prelevamento nella $i$-esima città origine, la funzione obiettivo che rappresenta il costo complessivo diventa

$$\sum_{i=1}^{5}\sum_{j=1}^{4} c_{ij}x_{ij} + \sum_{i=1}^{5} c_i \sum_{j=1}^{4} x_{ij}$$

Il file `trasporto1es.mod` che segue riporta il file del modello modificato in accordo con quanto richiesto dall'esercizio

`trasporto1es.mod`

```
set ORIGINI;
set DESTINAZIONI;

param  costo_origdest {ORIGINI,DESTINAZIONI} >= 0;

param  costo_origine {ORIGINI} >= 0;

param  max_uscita {ORIGINI} >= 0;

param  domanda {DESTINAZIONI} >=0;

var x {i in ORIGINI,j in DESTINAZIONI} >= 0;

minimize cost_trasporto: sum{i in ORIGINI,j in DESTINAZIONI}
           costo_origdest[i,j]*x[i,j] +  sum{i in ORIGINI}
           costo_origine[i]* sum{j in DESTINAZIONI} x[i,j];


 s.t. origini {i in ORIGINI}:
     sum{j in DESTINAZIONI} x[i,j] <= max_uscita[i] ;

 s.t. destinazioni {j in DESTINAZIONI}:
     sum{i in ORIGINI} x[i,j] = domanda[j] ;
```

Come si può vedere è stato aggiunta l'istruzione

```
param costo_origine {ORIGINI}
```

per rappresentare i costi di prelevemento da ciascuna città origine. Inoltre è stata modificata la funzione obiettivo.

Il file `trasporto1es.dat` che segue riporta il nuovo file di dati per il modello modificato come richiesto dall'esercizio.

trasporto1es.dat

```
set ORIGINI := asti bergamo como domodossola lecce;

set DESTINAZIONI := mantova napoli olbia palermo;

param:           costo_origine  max_uscita :=
     asti              2          100
     bergamo           3          110
     como              4          130
     domodossola       7           85
     lecce             6          120       ;

param               domanda   :=
     mantova           30
     napoli            18
     olbia             45
     palermo           56        ;

param costo_origdest : mantova  napoli  olbia  palermo :=
          asti            1       3.5     4        4.5
          bergamo          .1     3       4.5      4.8
          como             .3     2.8     4.7      4.9
          domodossola     1       2.2     3.9      4.0
          lecce           1.7     2.2     5.0      5.3   ;
```

Introduciamo ora ulteriori modifiche del modello dei trasporti considerato, prendendo in considerazione elementi aggiuntivi che sono di solito presenti nei problemi di trasporto. Supponiamo quindi che ci sia

1. un costo aggiuntivo da imputarsi a tasse portuali ad ogni unità di merce trasportata alla città di Olbia pari a 0.1;
2. un vincolo che impone che almeno i 4/5 della merce trasportata provenga da città origine del nord Italia;

3. un vincolo che impone che almeno i 2/3 della merce trasportata raggiunga città destinazione del sud Italia e delle isole.

4. un ulteriore costo di trasporto per ogni unità di merce trasportata per qualche coppia di città origine e città destinazione da imputarsi, ad esempio, a pedaggi autostradali; la tabella che segue riporta per quali coppia di città esiste questo costo aggiuntivo e a quanto ammonta

| | | | | | |
|---|---|---|---|---|---|
| Bergamo—Napoli | 2 | Domodossola—Mantova | 0.5 | Lecce—Olbia | 3.5 |
| Bergamo—Olbia | 3.5 | Domodossola—Palermo | 3.8 | Lecce—Palermo | 3.1 |

Per tener conto di queste esigenze aggiuntive, sarà necessario introdurre nuovi insiemi nel modello in AMPL: l'insieme `NORD` delle città del nord, l'insieme `SUD` delle città del sud e l'insieme `ISOLE` delle città che si trovano nelle isole. Inoltre si dovrà fare uso degli operatori tra insiemi di *intersezione* e *unione* (`inter` e `union`). L'uso di questi operatori è molto semplice e permette di definire gli insiemi intersezione e unione, ad esempio, nel seguente modo:

```
set ORIGINI;           # insieme delle citta' origini
set DESTINAZIONI;      # insieme delle citta' destinazioni
set NORD;              # insieme di alcune citta' del nord
set SUD;               # insieme di alcune citta' del sud
set ISOLE;             # insieme di alcune citta' delle isole

set ORI_NORD := ORIGINI inter NORD;
                       # insieme intersezione degli
                       # insiemi ORIGINI e NORD

set DEST_SUDISOLE := DESTINAZIONI inter {SUD union ISOLE};
                       # insieme intersezione dell'insieme
                       # DESTINAZIONI con l'insieme
                       # (SUD unione ISOLE)
```

Il file `trasporto2.mod` che segue riporta il nuovo modello di trasporti che tiene conto delle nuove esigenze sia nella funzione obiettivo sia per la presenza di nuovi vincoli.

`trasporto2.mod`

```
####    SEZIONE PER LA DICHIARAZIONE DEGLI INSIEMI    ####

set ORIGINI;
```

```
set DESTINAZIONI;
set NORD;
set SUD;
set ISOLE;

set ORI_NORD := ORIGINI inter NORD;

set DEST_SUDISOLE := DESTINAZIONI inter {SUD union ISOLE};
                      # introduce l'insieme intersezione
                      # dell'insieme DESTINAZIONI con l'insieme
                      # (SUD unione ISOLE)

      # le due dichiarazioni che seguono definiscono l'insieme
      # COPPIE formato da coppie ordinate di elementi: il primo
      # appartenente ad ORIGINI, il secondo a DESTINAZIONI.
      # Tuttavia, mentre la prima dichiarazione definisce COPPIE
      # come prodotto cartesiano e quindi contiene TUTTE le coppie
      # possibili, la seconda lo definisce solo come un sottoinsieme
      # del prodotto cartesiano (i cui elementi devono essere
      # specificati nel file .dat)

#set COPPIE := {ORIGINI,DESTINAZIONI};
set COPPIE within {ORIGINI,DESTINAZIONI};

param  costo_origdest {ORIGINI,DESTINAZIONI} >= 0;

param  costo_origine {ORIGINI} >= 0;

param  max_uscita {ORIGINI} >= 0;

param  domanda {DESTINAZIONI} >=0;

param costo_coppie {COPPIE} >=0;
            # per gli elementi  appartenenti all'insieme
            # COPPIE vi e' un ulteriore costo
            # di trasporto (pedaggio autostradale)

param costo_portuale;
            # costo unitario da imputarsi ad ogni unita' di
            # prodotto trasportato alla destinazione "olbia"
```

```
var x {i in ORIGINI,j in DESTINAZIONI} >= 0;

minimize cost_trasporto:  sum{i in ORIGINI,j in DESTINAZIONI}
       costo_origdest[i,j]*x[i,j] +  sum{i in ORIGINI}
       costo_origine[i]* sum{j in DESTINAZIONI} x[i,j] +
       sum{(i,j) in COPPIE} costo_coppie[i,j] * x[i,j] +
       sum{(i,"olbia") in {ORIGINI,DESTINAZIONI}}
                          costo_portuale * x[i,"olbia"];

                # a ciascun trasporto origine/destinazione
                # e' associato un costo, i costi poi vengono
                # sommati; agli elementi appartenenti allo
                # insieme COPPIE e' associato un costo in piu'.

s.t. origini {i in ORIGINI}:
     sum{j in DESTINAZIONI} x[i,j] <= max_uscita[i] ;

s.t. destinazioni {j in DESTINAZIONI}:
     sum{i in ORIGINI} x[i,j] = domanda[j] ;

s.t. origini_nord: sum {i in ORI_NORD,j in DESTINAZIONI}
      x[i,j] >= 4/5*sum {i in ORIGINI,j in DESTINAZIONI} x[i,j] ;
         # dalle citta' di origine del nord vengono effettuati
         # almeno i 4/5 dei trasporti totali.

s.t. destinazioni_isole:
      sum {i in ORIGINI,j in DESTINAZIONI : j in {SUD union ISOLE}}
      x[i,j] >= 2/3 * sum {i in ORIGINI,j in DESTINAZIONI} x[i,j] ;
         # verso le citta' di destinazione del sud e delle
         # isole vengono effettuati almeno i 2/3 dei trasporti
         # totali.
```

---

Il file di dati `trasporto2.dat` che segue permette di introdurre i dati assegnati nel modello.

`trasporto2.dat`

```
set ORIGINI := asti bergamo como domodossola lecce;

set DESTINAZIONI := mantova napoli olbia palermo;
```

---

```
set NORD := asti varese bergamo brescia como cremona pavia
            domodossola mantova bergamo padova
            sondrio torino milano venezia;

set SUD := bari lecce napoli cosenza foggia salerno
           reggiocalabria taranto crotone;

set ISOLE := cagliari olbia palermo sassari trapani;

param:           costo_origine  max_uscita :=
     asti              2          100
     bergamo           3          110
     como              4          130
     domodossola       7           85
     lecce             6          120       ;

param               domanda   :=
     mantova           30
     napoli            18
     olbia             45
     palermo           56        ;

param costo_origdest : mantova  napoli  olbia  palermo :=
          asti            1       3.5     6.0      4.5
          bergamo          .1     3       4.5      4.8
          como             .3     2.8     4.7      4.9
          domodossola     1       2.2     3.9      4.0
          lecce           1.7     2.2     5.0      5.3   ;

param: COPPIE :                   costo_coppie   :=
        bergamo napoli               2
        bergamo olbia                3.5
        domodossola mantova          0.5
        domodossola palermo          3.8
        lecce olbia                  3.5
        lecce palermo                3.1          ;
           # le possibili coppie origine/destinazione sono
           # 5*4 = 20 ma l'insieme COPPIE ne contiene solo 6

param costo_portuale := 0.1 ;
```

```
   # per ogni unita' di merce che ha come destinazione "olbia"
   # vi e' questo costo aggiuntivo;
```

---

Si osservi che in questo file dei dati non è stato assegnato esplicitamente l'insime `COPPIE` con un'istruzione del tipo

```
set COPPIE :=
   (bergamo , napoli) (bergamo , olbia) (domodossola , mantova)
   (domodossola , palermo)  (lecce , olbia)  (lecce , palermo);
```

oppure

```
set COPPIE :=
   bergamo napoli
   bergamo olbia
   domodossola mantova
   domodossola palermo
   lecce olbia
   lecce palermo;
```

Infatti, AMPL, in questo caso prenderà come elementi dell'insieme `COPPIE` (che ricordiamo è stato dichiarato come *sottoinsieme* del prodtto cartesiano) tutte e sole le coppie specificate nell'assegnazione del parametro `costo_coppie`.

Risolvendo il problema completo con il solutore CPLEX si ha:

```
ampl: solve;
CPLEX 12.8.0.0: optimal solution; objective 933.92
7 dual simplex iterations (0 in phase I)
```

La soluzione ottenuta è la seguente:

```
ampl: display x;
x :=
asti        mantova    0.2
asti        napoli    18
asti        olbia     25.8
asti        palermo   56
bergamo     mantova   29.8
bergamo     napoli     0
bergamo     olbia      0
```

```
bergamo     palermo    0
como        mantova    0
como        napoli     0
como        olbia     19.2
como        palermo    0
domodossola mantova    0
domodossola napoli     0
domodossola olbia      0
domodossola palermo    0
lecce       mantova    0
lecce       napoli     0
lecce       olbia      0
lecce       palermo    0
;

ampl: display cost_trasporto;
cost_trasporto = 933.92
```

*Un problema di produzione industriale multiperiodo*

Consideriamo di nuovo il modello per la produzione industriale *multiperiodo* dell'Esempio 3.4.11 che implementeremo in AMPL utilizzando alcune delle strutture del linguaggio che abbiamo introdotto nel paragrafo precedente. In particolare, vedremo l'uso oltre che degli insiemi generici, anche degli insiemi numerici, ordinati e ciclici. Naturalmente facciamo riferimento alla formulazione già esaminata a pagina 45.

Per realizzare tale implementazione definiamo un insieme *generico* che chiameremo `tipi` per quanto riguarda il tipo di pneumatico (tipo A e tipo B). Per quanto riguarda le linee, pensando all'attribuzione a ciascuna linea di un numero intero, definiamo l'insieme delle linee (`linee`) come un insieme *numerico* formato dai numeri interi da 1 a `numlinee`, dove `numlinee` è un parametro che corrisponde al numero di linee presenti e che verrà assegnato nel file dei dati. Naturalmente, la dichiarazione del parametro `numlinee` dovrà precedere quella dell'insieme `linee`. Si ribadisce che in questo modo il modello definito nel file dei dati è indipendente dai dati che saranno introdotti nel file `.dat`. Infine, si sceglie di utilizzare un insieme (ordinato) *ciclico* per quanto riguarda l'insieme dei mesi. Questa scelta è dettata dai vincoli che dovranno essere implementati e sarà esaminata nel dettaglio nel seguito.

Per quanto riguarda le variabili, sono state introdotte le variabili con tre indici `x{tipi,linee,mesi}` e le variabili `x_im{tipi,mesi}` relative alle quantità immagazzinate. Poiché nell'ultimo periodo non è previsto l'immagazzinamento, sarà

necessario imporre che il quantitativo di pneumatici immagazzinato nell'ultimo periodo sia pari a 0.

Un file `.mod` che rappresenta bene il modello in esame è riportato di seguito.

pneumatici.mod

```
set tipi;   # insieme generico

param numlinee integer;   # parametro numlinee con
                          # controllo dell'interezza

set linee:= 1..numlinee;    # insieme numerico

set mesi circular;   # insieme ordinato ciclico

param ordine{mesi,tipi}>=0;
param disponibilita{mesi,linee}>=0;
param tempi{tipi,linee}>=0;

param costo_orario>=0;
param costo_immagazzinamento>=0;

param costo_materiale_grezzo{tipi}>=0;

var x{tipi,linee,mesi}>=0;  # quantita' di pneumatici di
                            # ciascun tipo prodotti da
                            # ciascuna linea in ciascun mese

var x_im{tipi,mesi}>=0; # quantita' di pneumatici di ciascun
                        # tipo immagazzinato in ciascun mese
                        # ove questo e' possibile
                        # (sara' quindi necessario porre
                        # x_im[i,last(mesi)]=0
                        # al variare di {i in tipi})


minimize costo_totale:
 sum{i in tipi, j in linee, k in mesi}
                          costo_orario*tempi[i,j]*x[i,j,k] +
 sum{i in tipi, j in linee, k in mesi}
                          costo_materiale_grezzo[i]*x[i,j,k]+
 sum{i in tipi, l in mesi} costo_immagazzinamento*x_im[i,l];
```

```
# vincoli disponibilita' delle macchine

s.t. vincoli_disponibilita_linee{k in mesi, j in linee}:
      sum{i in tipi} tempi[i,j]*x[i,j,k] <= disponibilita[k,j];



# vincoli dovuti alla richiesta e all'immagazzinamento

s.t. vincoli_richiesta{k in mesi, i in tipi}:
        x_im[i,prev(k,mesi)]+sum{j in linee} x[i,j,k] =
                                    ordine[k,i]+x_im[i,k];
```

---

Vale la pena soffermarci sulla definizione dei vincoli. Iniziamo dai vincoli sulla disponibilità delle macchine. Per ogni mese $k$, si deve esprimere un vincolo per ogni linea di produzione. Quindi, in prima analisi, potevamo scrivere i seguenti vincoii:

```
s.t. vincoli_disponibilita_linea1{k in mesi}:
      sum{i in tipi} tempi[i,1]*x[i,1,k] <= disponibilita[k,1];

s.t. vincoli_disponibilita_linea2{k in mesi}:
      sum{i in tipi} tempi[i,2]*x[i,2,k] <= disponibilita[k,2];
    .
    .
    .
```

avendo inidicizzato i vincoli sull'insieme dei mesi. Ma questa scrittura, oltre che essere lunga, prevede di conoscere già nel file del modello quante sono le linee di produzione, dato che invece deve essere un parametro assegnato nel file dei dati. Quindi conviene indicizzare i vincoli oltre che sulla base dei mesi, anche rispetto alle linee di produzione.

Per quanto riguarda i vincoli dovuti alla richiesta e all'immagazzinamento, essendo l'insieme `mesi` un insieme ciclico, possiamo evitare di tenere separati i vincoli riguardanti il primo e l'ultimo mese purché si imponga che il quantitativo di pneumatici immagazzinato nell'ultimo periodo sia pari a 0. Anche in questo caso, per ragioni analoghe, si è evitato di scrivere i vincoli nella forma

```
s.t. vincoli_richiesta_tipoA{k in mesi}:
            x_im["tipoA",prev(k,mesi)]+
            sum{j in linee} x["tipoA",j,k] =
            ordine[k,"tipoA"]+x_im["tipoA",k];

s.t. vincoli_richiesta_tipoB{k in mesi}:
            x_im["tipoB",prev(k,mesi)]+
            sum{j in linee} x["tipoB",j,k] =
            ordine[k,"tipoB"]+x_im["tipoB",k];
    .
    .
    .
```

Si osservi che se l'insieme dei mesi fosse stato definito solo come insieme ordinato (ma non ciclico), quando $k = 1$, l'istruzione prev(k,mesi) sarebbe stata errata, non essendo definito in questo caso nell'insieme dei mesi l'elemento predecessore del primo.
L'assegnazione dei parametri nel file `.dat` è molto standard. Come abbiamo già detto dobbiamo *fissare* a 0 le variabili che identificano le quantità di pneumatici immagazzinati nell'ultimo mese. Il fixing delle variabili si può effettuare o nel file `.dat` o nel file `.run`. In questo caso, trattandosi di una richiesta parte integrante del modello, lo effettuiamo nel file `.dat`. Si tratterebbe di scrivere le istruzioni

```
fix x_im["tipoA",last(mesi)]:=0;
fix x_im["tipoB",last(mesi)]:=0;
  .
  .
  .
```

Anche in questo caso, sia per brevità di scrittura, ma soprattutto per rendere il modello valido anche nel caso in cui si cambino i dati, scegliamo di fissare le variabili lasciando variare il *tipo* all'interno dell'insieme `tipi`. In questo caso si può procedere utilizzando una sintassi del linguaggio AMPL non ancora introdotta: si tratta dell'istruzione `for`. Il suo uso verrà specificato meglio nel seguito, ma la sintassi è abbastanza standard. In questo caso è

```
for {i in tipi}{fix x_im[i,last(mesi)]:=0;}
```

Quindi il file del dati per il modello in esame può essere scritto come segue:

pneumatici.dat

```
set mesi := ott nov dic;
```

```
set tipi := tipoA tipoB;

param numlinee := 2;

param ordine:   tipoA tipoB :=
       ott     16000  14000
       nov      7000   4000
       dic      4000   6000;

param disponibilita:       1     2 :=
                ott      2000  3000
                nov       400   800
                dic       200  1000;

param tempi:                 1     2 :=
                tipoA      0.10  0.12
                tipoB      0.12  0.18;

param costo_orario := 6000;

param costo_immagazzinamento := 350;

param costo_materiale_grezzo :=
                        tipoA 2500
                        tipoB 4000;

for {i in tipi}{fix x_im[i,last(mesi)]:=0;}
```

---

Risolvendo il problema con il solutore CPLEX si ha:

```
ampl: solve;
CPLEX 12.8.0.0: optimal solution; objective 202521666.7
15 dual simplex iterations (0 in phase I)
```

La soluzione ottenuta è la seguente:

```
ampl: display x, x_im, costo_totale;
x :=
tipoA 1 ott     333.333
tipoA 1 nov     333.333
tipoA 1 dic       0
```

---

```
tipoA 2 ott   15666.7
tipoA 2 nov    6666.67
tipoA 2 dic    4000
tipoB 1 ott   16388.9
tipoB 1 nov    3055.56
tipoB 1 dic    1666.67
tipoB 2 ott       0
tipoB 2 nov       0
tipoB 2 dic    2888.89
;

x_im :=
tipoA ott      0
tipoA nov      0
tipoA dic      0
tipoB ott   2388.89
tipoB nov   1444.44
tipoB dic      0
;

costo_totale = 202522000
```

## 4.6 I PRINCIPALI COMANDI AMPL

Abbiamo già utilizzato alcuni comandi AMPL per poter determinare la soluzione ottima di alcuni esempi di modelli. Ad esempio, abbiamo già utilizzato il comando `solve` per invocare il solutore, oppure il comando `option solver cplex` per definire il solutore da utilizzare ed anche il comando `display` per visualizzare il risultato ottenuto. Vogliamo ora dare un quadro più generale riguardante l'uso di questi comandi.
Innanzitutto, i comandi vengono dati su riga di comando al prompt di AMPL, oppure sono inseriti in un file `.run`. Essi possono essere di fatto utilizzati per scrivere dei veri e propri programmi in AMPL.

### 4.6.1 Il comando `option`

Il comando `option` serve per visualizzare o cambiare il valore delle *opzioni*. Le *opzioni* sono *variabili di stato* dell'ambiente AMPL. Ciascuna di esse ha un nome ed un valore che può essere un numero o una stringa di caratteri. Per aver un'idea

di quali sono le variabili di stato in AMPL digitare sul prompt dei comandi di AMPL il comando

```
option;
```

Verrà visualizzato un lungo elenco di tutte le variabili di stato di AMPL e il loro valore corrente. Il comando `option` senza ulteriore specificazione serve infatti per visualizzare il valore delle varibili di stato. Il comando `option` accetta una "wild card" che è rappresentata dal carattere "*" ed è utilizzato per rappresentare qualsiasi stringa. Quindi, ad esempio, con il comando `option presolve*` si otterrà la lista di tutte le opzioni il cui nome inizia per `presolve` e il loro valore corrente.

Per visualizzare il valore corrente di un'opzione specifica si dovrà specificare il `nomeopzione`. Quindi per visualizzare il valore dell'opzione `nomeopzione` sarà sufficiente specificare

```
option nomeopzione;
```

Per modificare il valore dell'opzione `nomeopzione` sarà sufficiente il comando che indichi questo nuovo valore:

```
option nomeopzione nuovovalore;
```

Abbiamo già visto un esempio di questo comando quando abbiamo selezionato il solutore da utilizzare con il comando `option solver cplex`. In questo caso la variabile di stato è `solver` che viene impostata al valore `cplex`.

Prestare molta attenzione al fatto che il comando `option` non controlla subito che il valore assegnato abbia senso o meno; un messaggio di errore si avrà solo in fase di esecuzione.

Infine, per riportare tutte le opzioni al loro valore di default si utilizza il comando

```
reset options;
```

Per un elenco completo di tutte le opzioni si rimanda al testo di AMPL già citato. Ne riportiamo di seguito solamente tre di uso frequente:

- `solver` specifica il solutore. Il suo valore di default (che è `minos`) può essere cambiato specificando il nome di un diverso solutore.
- `presolve` specifica le opzioni del preprocessamento. Il preprocessamento è un'operazione che AMPL può effettuare allo scopo di ridurre il problema, ad esempio, eliminando vincoli ridondanti, fissando valori di alcune variabili, etc. Tale procedimento è molto utile (e a volte indispensabile) nella

risoluzione di problemi a grandi dimensioni. Il valore di default è 10. Per inibire il preprocessamento è sufficiente assegnare valore 0 a `presolve`.

- `show_stats` specifica il livello di dettaglio delle informazioni sul problema e sulla soluzione che devono essere visualizzate. Il valore di default è 0, in corrispondenza del quale vengono visualizzate informazione minime. Assegnando il valore 1 o superiori a `show_stats` aumenta il livello di dettaglio delle informazioni visualizzate.

Attraverso il comando `option` si possono inoltre specificare le opzioni relative al solutore utilizzato.

### 4.6.2 Il comando `display`

Il comando `display` si utilizza per visualizzare oggetti presenti nel modello come, ad esempio, gli elementi di un insieme, il valore delle variabili, dei parametri, della funzione obiettivo, dei vincoli. Nella sua versione più semplice consente di visualizzare il valore di un oggetto denominato `nomeoggetto` tramite il comando

```
display nomeoggetto;
```

Dopo il comando `display` posso essere anche elencati un certo numero i oggetti da visualizzare separati dalla virgola. Con il comando `display` possono essere anche utilizzate espressioni algebriche o logiche come riportato negli esempi che seguono (facendo riferimento agli oggetti utilizzati nell'esempio precedente:

```
display mesi;
display x;
display costo_totale;
display {i in tipi} x[i,1,"nov"];

display sum{i in tipi, j in linee, k in mesi}
                    costo_materiale_grezzo[i]*x[i,j,k];

display {i in tipi, k in mesi : x[i,1,k] > 100} x[i,1,j];
```

Non forniamo spiegazioni dettagliate di queste istruzioni perché sono molto intuitive. Ci soffermiamo solamente sull'uso dei ":" che può essere introdotto nei costrutti logici, come nell'ultimo comando `display` dell'esempio, con il significato di "*tale che*".

Le opzioni del comando `display` riguardano la formattazione delle informazione da visualizzare e l'approssimazione utilizzata nell'arrotondamento dei valori numerici da visualizzare. Per esse si fa riferimento al Capitolo 12 del testo di AMPL già citato ed in particolare alle Tabelle 12-1 e 12-2.

### 4.6.3 Reindirizzamento dell'output dei comandi

È molto utile poter reindirizzare l'output dei comandi in un file nel quale conservare tale output. Questo vale per tutti i comandi, ma in particolare per il comando `display`. Infatti, in questo modo si può facilmente memorizzare la soluzione ottima e altre informazioni. Per creare un file testo di output `nomefile.out` nel quale reindirizzare l'output del comando `display` è sufficiente il comando

```
display oggetto > nomefile.out;
```

In questo modo viene creato o eventualmente sovrascritto (se già esistente) il file `nomefile.out` nel quale verrà scritto l'output del comando. Se si vuole "appendere", ovvero aggiungere alla fine del file, altri output è sufficiente il comando

```
display oggetto2 >> nomefile.out;
```

In questo modo, nel file `nomefile.out`, dopo il/i valore/i di `oggetto` compariranno il/i valore/i di `oggetto2`.

Quindi, sempre in riferimento all'esempio precedente, è possibile aggiungere nel file `.run` (ovviamente dopo il comando `solve`) i comandi

```
display x > risultati.out;
display x_im >> risultati.out;
display costo_totale >> risultati.out;
```

per creare il file `risultati.out` contenente i valori della variabili all'ottimo e il valore ottimo della funzione obiettivo.

### 4.6.4 Il comando `display` per visualizzare altre grandezze relative alle variabili all'ottimo

Nella risoluzione di problemi di Programmazione Lineare, AMPL oltre a fornire (ove esista) la soluzione ottima del problema, permette di visualizzare anche altri elementi del problema come i prezzi ombra i costi ridotti associati alla soluzione ottima. Per visualizzare questi elementi è sufficiente aggiungere dei suffissi al nome della variable. In particolare, se $x$ è un variabile del problema, possiamo utilizzare il comando

```
display x.lb, x.ub;
```

per visualizzare il lower bound e l'upper bound della variabile $x$. Quindi, ad esempio, se $x$ è una variabile definita non negativa, il comando fornirà il valore `0` per il lower bound e il valore `Infinity` per l'upper bound. Il comando

```
display x.slack;
```

visualizza la differenza tra il valore della variabile e il più vicino bound.

Il concetto di "bound" e di "slack" ha un'interpretazione analoga anche per i vincoli di un modello. Ovvero si pensa al vincolo scritto nella forma

$$\textit{lower bound} \quad \leq \quad \textit{body} \quad \leq \quad \textit{upper bound}$$

e quindi, se `vinc` è l'etichetta data ad un vincolo, il comando

```
display vinc.lb, vinc.body, vinc.ub;
```

visualizza il lower bound del vincolo, il valore della parte variabile del vincolo e l'upper bound del vincolo. Il comando

```
display vinc.slack;
```

visualizza la differenza tra il valore del vincolo e il più vicino bound.

Il comando `display` si può utilizzare anche per avere informzioni sulle quantità *duali* associate al problema. Come sarà esaminato nel Capitolo 8, a ciascun vincolo di un problema di Programmazione Lineare si può associare una variabile duale e il valore ottimo di tale variabile viene chiamato *prezzo ombra* o *valore marginale*. Con il comando

```
display vincolo;
```

si visualizza tale valore, ovviamente senza la necessità di dover costruire esplicitamente il problema duale. Similmente il comando

```
display x.rc;
```

visualizza il *costo ridotto* associato alla variabile $x$.

L'uso delle quantità duali e la loro interpretazione verrà trattata nel dettaglio nel prossimo Capitolo 8 nel quale verrà affrontata *l'analisi di sensitività* della soluzione ottima rispetto a parametri di un problema di Programmazione Lineare.

### 4.6.5 Comandi per aggiornare il modello: `reset`, `drop` e `restore`

Sono disponibili comandi per modificare anche solo parzialmente un modello. Il comando `reset`, già utilizzato, cancella completamente il modello e i dati. Equivale ad uscire (con il comando `quit`) da AMPL e rientrare. Esistono poi comandi per far in modo che alcuni vincoli siano ignorati. Il comando è `drop`. Quindi utilizzando i comandi

```
drop vincolo1;
drop vincoli_risorse{i in RISORSE};
drop vincolo_budget["periodo1"];
```

si ottiene che i vincoli corrispondenti vengano ignorati. Con il comando `restore` si ripristinano vincoli che fossero stati eventualmente in precedenza "ignorati".

### 4.6.6 Altri utili comandi: `show`, `xref`, `expand`

Sono comandi che servono per visualizzare componenti del modello.

- Il comando `show` visualizza tutte le componenti del modello, ovvero parametri, insiemi, variabili, vincoli e funzione obiettivo.
- Il comando `xref` visualizza tutte le componenti del modello che dipendono da una specifica componente.
- Il comando `expand` applicato ad una funzione obiettivo genera la sua espressione completa esplicita. Applicato ad un vincolo genera tutti i vincoli derivanti da un vincolo scritto in forma indicizzata su un insieme. Applicato ad una variabile, visualizza tutti i coefficienti non nulli di questa variabile nei termini lineari della funzione obiettivo e dei vincoli. Se inoltre la variabile compare anche in espressioni non lineari allora viene aggiunto all'output l'espressione `+ nonlinear`.

### 4.6.7 Nomi generici per variabili, vincoli, e funzioni obiettivo

AMPL rende disponibili parametri che forniscono il *numero* delle variabili, dei vincoli e delle funzioni obiettivo del problema:

- `_nvars`: numero delle variabili del problema
- `_ncons`: numero dei vincoli del problema
- `_nobjs`: numero delle funzioni obiettivo del problema.

Sono disponibili inoltre parametri che forniscono i *nomi* delle componenti del problema:

- `_varname`: nomi delle variabili del problema
- `_conname`: nomi dei vincoli del problema
- `_objname`: nomi delle funzioni obiettivo del problema.

Infine, sono disponibili sinonimi per le componenti del problema:

- `_var`: sinonimo per le variabili del problema
- `_con`: sinonimo per i vincoli del problema
- `_obj`: sinonimo per le funzioni obiettivo del problema

Utilizzando questi *sinonimi* è possibile scrivere un file `.run` che può essere utilizzato per la soluzione di problemi diversi senza dover indicare volta per volta il nome specifico delle variabili e della funzione obiettivo nel comando `display`. Un esempio di un tale file `.run` è il seguente:

```
reset;
model modello.mod;
data modello.dat;

option solver cplex;
solve;

display _varname, _var;
display _objname, _obj;
```