**MODELLO PL**

int numVariabili = ...;

int numVincoli = ...;

range Variabili = 1..numVariabili;

range Vincoli = 1..numVincoli;

float C[Variabili] = ...;

float A[Vincoli][Variabili] = ...;

float B[Vincoli] = ...;

dvar float+ X[Variabili];

**minimize** sum(j in Variabili) C[j]\*X[j];

subject to{

forall(i in Variabili) //sommatoria

vincoli : sum(j in Variabili) A[i][j]\*X[j] = B[i];

}

**GRAFI**

**METODO 1**

tuple nodo{

int indic;

int pesoN\_1;

}

{nodo} insiemeNodi = ...;

tuple arco{

int nodoOut;

int nodoIn;

int peso;

}

{arco} insiemeArchi;

**METODO 2**

int NNodi = ...;

range Nodi = 1..NNodi;

int peso\_N1[Nodi] = ...;

int peso\_ecc....

int Peso\_NK[Nodi] = ...;

tuple arco{

int nodoOut;

int nodoIn;

}

{arco} insiemeArchi = ...;

int PesoA\_1[insiemeArchi] = ...;

...

int Peso!\_q[insiemeArchi] = ...;

**CAMMINO DI COSTO MINIMO S-T FLUSSO**

int nNodi = ...;

range Nodi = 1...nNodi;

tuple arco{

int NodoOut;

int NodoIn;

}

{archi} Archi = ...;

int S = ...; int T = ...;

int costoA[Archi] = ...;

dvar float+ X[Archi];

**minimize** sum(a in Archi) costoA[a]\*F[a];

subject to{

sum( a in Archi : a.nodoOut == S) F[a] = 1;

- sum( a in Archi : a.nodoIn == T) F[a] == -1;

forall(i in Nodi : i!= S && j!=T)

sum(a in Archi : a.nodoOut == i) F[a] - sum(a in Archi:a.nodoIn == i) F[a] == 0;}

**PROBLEMA DEI TRASPORTI**

int NNodi = ...;

range Nodi = 1..NNodi;

tuple arco{ int NodoIn; int NodoOut;}

{arco} InsiemeArchi = { <i,j> | i in Nodi, j in Nodi : i!=j};

int divergenza[Nodi] = ...;

int costoA[InsiemeArchi] = ...; OPPURE int costoA[Nodi][Nodi]=…;

int capA[InsiemeArchi] = ...; OPPURE capA[Nodi][Nodi] = …;

dvar float+ f[InsiemeArchi];

**minimize** sum(i in Nodi) sum(j in Nodi: j!=1) costoA[i][j]\*f[i][j];

forall(i in Nodi)

sum(j in Nodi : j!=i) f[i][j] - sum(j in Nodi: j!=i)f[j][i] == divergenza[i];

int nOrigini = ...; int nDestinazioni = ...;

range Origini = 1..nOrigini;

range Destinazioni = 1..nDestinazioni;

int D[origini] = ...;

int R[destinazioni] = ...;

int C[Origini][Destinazioni] = ...;

dvar float+ F[Origini][Destinazioni];

minimize sum(i in Origini, j in Destinazioni) C[i][j]\*F[i][j];

subject to{

forall(i in Origini)

sum(j in Destinazioni) F[i][j] == D[i]; //disponibilità

forall(j in Destinazioni)

sum(i in Origini) F[i][j] == R[j]; //richiesta}

**ASSEGNAMENTO = TRASPORTI CON mp=ms=m e divergenze tutte = 1**

**PROBLEMA DI PROGRAMMAZIONE LINEARE INTERA**

int numVariabili = ...;

int numVincoli = ...;

range Vincoli = 1..numVincoli;

range Variabili = 1..Variabili;

float C[Variabili] = ...;

float A[Vincoli][Variabili] = ...;

B[Vincoli] = ...;

dvar+ X[Variabili] in 0..1; //se x deve essere compreso tra 0 e 1

**KNAPSACK MULTIPLO**

int nrOggetti = ...;

int nrContenitori = ...;

range Oggetti = 1..nrOggetti;

range Contenitori = 1..nrContenitori;

int v[Oggetti] = ...; //valore da maximize

int w[Contenitori] = ...; //ingombro degli oggetti

dvar int+ X[Contenitori][Oggetti] in 0..1;

**maximize** sum (i in Contenitori, j in Oggetti) v[j]\*x[i][j];

subject to{

forall(j in Oggetti)

sum(i in Contenitori) X[i][j] <= 1;

forall(i in Contenitori)

sum(j in Oggetti) w[j]\*Xèi][j] <= b[i]}

**ASSEGNAMENTO GENERALIZZATO**

int nrOggetti = ...;

int nrContenitori = ...;

range Oggetti = 1..nrOggetti;

range Contenitori = 1..nrContenitori;

int v[Contenitori][Oggetti] = ...;

int w[Contenitori][Oggetti] = ...;

int b[Contenitori] = ...;

dvar int+ X[Contenitori] = ...;

**maximize** sum(i in Contenitori, j in Oggetti) v[i][j]\*x[i]+[j];

subject to{

forall(j in Oggetti)

sum(i in Contenitori) X[i][j] == 1;

forall(i in Contenitori)

sum(j in Oggetti) w[i][j]\*x[i][j]<=b[i];}

**SET COVERING MATRICE CON RIGHE PARI AD OGGETTI**

int m = ...;

int n = ...;

range nElementi = 1..m;

range nSottinsiemi = 1..n;

int A[nElementi][nSottinsiemi] = ...;

int C[nSottinsiemi] = ...;

dvar int+ X[nSottinsiemi] in o..1;

minimize sum(j in nSottinsiemi) c[j]\*x[j];

subject to{

forall(i in Elementi)

sum(j in nSottinsiemi) A[i][j]\*X[j]>=1; //per **PARTITIONING** Ax==1}

**PLANT LOCATION NON CAPACITATO**

int m = ...;

int n = ...;

range nClienti = 1..m; UTENTI

range nImpianti = 1..n; LOCAZIONI

int F[nImpianti] = ...;

int C[nClienti][nImpianti]=...;

d ìar int+ Y[nImpianti] = ...;

dvar int+ X[nClienti][nImpianti] in 0..1;

**minimize** sum(i in nClienti) sum(j in nImpianti) C[i][j]\*X[i+[j+ + sum(j in nImpianti) F[j]\*Y[j];

subject to{

forall(i in nClienti)

sum (j in nImpianti) X[i][j] == 1;

forall(i in nClienti, j in nImpianti)

X[i][j] <= Y[i][j];} **\*\*\***

**SE CAPACITATO LA DOMANDA NON è UNITARIA MA PARI A D[i] con capacità limitata Kj di erogazione da parte dell’impianto**

forall(i in nClienti, j in nImpianti)

D[i]\*X[i][j] <= Y[i][j]\*K[j];}

**P-MEDIANA**

int m = ...; int n = ...;

int p = ...;

range nClienti = 1..m;

range nImpianti = 1..n;

int F[nImpianti] = ...;

int C[nClienti][nImpianti]=...;

dvar int+ Y[nImpianti] in 0..1;

dvar int+ X[nClienti][nImpianti] in 0..1;

minimize sum(i in nClienti) sum j in nImpianti) C[i][j]\*X[i][j];

subject to{

forall(i in Clienti)

sum(j in nImpianti) X[i][j] == 1;

forall(i in nClienti, j in nImpianti)

X[i][j] <= Y[j];

sum(j in nImpianti) Y[j] == p;}

**P-CENTRO**

int m = ...;

int n = ...;

int p = ...;

range nClienti = 1..m;

range nImpianti = 1..n;

int F[nImpianti] = ...;

int C[nClienti][nImpianti]=...;

dvar int+ Y[nImpianti] in 0..1;

dvar int+ X[nClienti][nImpianti] in 0..1;

dvar float+ CostoMax;

**minimize** CostoMax;

subject to{

forall(i in nClienti)

sum(j in nImpianti) X[i][j+==1;

forall(i in nClienti, j in nImpianti) X[i][j] <= Y[j];

sum(j in nImpianti) Y[j]==p;

forall(i in nClienti, j in nImpianti)

CostoMax >= C[i][j]\*X[i][j];}

**ASSEGNAMENTO**

**PROBLEMA DEI TRASPORTI TALE CHE NORIGINI = NDESTINAZIONI = M**

int nOrigini = ...;

range Origini = 1..nOrigini;

int C[Origini][nOrigini] = ...;

dvar float+ F[Origini][nOrigini];

minimize sum(i in Origini, j in Origini) C[i][j]\*F[i][j];

subject to{

forall(i in Origini)

sum(j in Destinazioni) F[i][j] == 1; //domanda

forall(j in Destinazioni)

sum(i in Origini) F[i][j] == 1; //richiesta

}

**COMMESSO VIAGGIATORE ASIMMETRICO**

**DANTZIG FULKERSON JOHNSON**

intNNodi= ...;

rangeNodi = 1..Nnodi;

intCosto[Nodi][Nodi] = ...;

dvar int X[Nodi][Nodi] in 0..1;

dvar float+ U[Nodi];

minimize sum(i inNodi, jin Nodi : i != j) Costo[i][j]\*X[i][j];

subject to{

forall(i inNodi) archi\_uscenti\_da\_i:

sum(j inNodi: j != i) X[i][j] == 1;

forall(i inNodi)archi\_entranti\_in\_i:

sum(j in Nodi: j != i) X[j][i] == 1;

forall(i inNodi : i !=1)

forall(j inNodi: j != 1 && j != i)U[i] -U[j] + NNodi\*X[i][j] <= NNodi-1;}

tuple coppia{int i;int j;}

{coppia} Coppie= {<i, j> | i inNodi, j inNodi: X[i][j]==1};

execute DISPLAY {writeln("Coppie= ", Coppie);

writeln("Posizione=", U);}

**VINCOLI DI CONNESSIONE DFJ**

forall(winSottoinsiemi: Card[w] >=2&& Card[w]<=nNodi-2)

sum(iinNodi, jinNodi: W[w][i]==1 && W[w][j]==0)X[i][j] <=Card[w]-1;

**VINCOLI ELIINAZIONE DFJ**

forall(winSottoinsiemi: Card[w] >=2&& Card[w]<=nNodi-2)

sum(iinNodi, jinNodi: W[w][i]==1 && W[w][j]==1 && i!= j)X[i][j] <=Card[w]-1;

**KNAPSACK oggetti in contenitore con valori e ingombri e capacità unica del contenitore B**

int nrOggetti = ...;

range Oggetti = 1..nrOggetti;

int v[Oggetti] = ...; //valore da maximize

int w[oggetti] = ...; //ingombro degli oggetti

dvar int+ X[Oggetti] in 0..1;

**maximize** sum (j in Oggetti) v[j]\*x[i][j];

subject to{

sum(j in Oggetti) w[j]\*X[i][j] <= B}

**KNAPSACK INTERO BOUNDED CON NUMERO DI COPIE DELL’OGGETTO J PARI A Uj oggetti in contenitore con valori e ingombri e capacità unica del contenitore B**

int nrOggetti = ...;

range Oggetti = 1..nrOggetti;

int u[Oggetti] = …; // numero delle copie

int v[Oggetti] = ...; //valore da maximize

int w[oggetti] = ...; //ingombro degli oggetti

dvar int+ X[Oggetti] in 0..1;

**maximize** sum (j in Oggetti) v[j]\*x[i][j];

subject to{

sum(j in Oggetti) w[j]\*X[i][j] <= B

&& X[j]>=0 && X[j]<=u[j];}

**CHANGE MAKING**

**KNAPSACK INTERO BOUNDED CON VALORE UNITARIO**

int nrOggetti = ...;

range Oggetti = 1..nrOggetti;

int u[Oggetti] = …; // numero delle copie

int v[Oggetti] = ...; //valore da maximize

int w[oggetti] = ...; //ingombro degli oggetti

dvar int+ X[Oggetti] in 0..1;

**maximize** sum (j in Oggetti) x[i][j];

subject to{

sum(j in Oggetti) w[j]\*X[i][j] <= B

&& X[j]>=0 && X[j]<=u[j];}

**BIN PACKING MINIMIZZA I CONTENITORI**

int nrOggetti = ...;

int nrContenitori = …;

range Contenitori = 1..nrContenitori;

range Oggetti = 1..nrOggetti;

int w[oggetti] = ...; //ingombro degli oggetti

dvar int+ x[oggetti] in 0..1;

dvar int+ Y[Contenitori] in 0..1;

**minimize** sum (j in Contenitori) y[j];

subject to{

sum(j in Contenitori) x[i] = 1;

sum(i in Oggetti) w[j]\*x[i][j] <= B[j];}

**MASSIMO FLUSSO**

int nNodi = ...;

range Nodi = 1...nNodi;

tuple arco{

int NodoOut;

int NodoIn;

}

{archi} Archi = ...;

int S = ...; int T = ...;

int U[Archi] = ...; //capacità dell’arco

dvar float+ F[Archi];

**maximize** sum(a in Archi : a.nodoOut == S) F[a];

subject to{

forall(i in Nodi : i!= S && j!=T)

sum(a in Archi : a.nodoOut == i) F[a] - sum(a in Archi:a.nodoIn == i) F[a] == 0;

F[a] in 0..U[a]; }