Handout to accompany talk on Xpress-Mosel

Bob Daniel, Dash Optimization, April 2003

The BCL Example

```
#include <stdio.h>
#include <iostream.h>
#include "xprb_cpp.h"

using namespace ::dashoptimization;

#define NSHARES 10                // Number of shares
#define NRISK 5                   // Number of high-risk shares
#define NNA 4                     // Number of N American shares

double RET[] = {5,17,26,12,8,9,7,6,31,21};  // Estimated return on investment
int RISK[] = {1,2,3,8,9};         // High-risk values among shares
int NA[] = {0,1,2,3};             // Shares issued in N America

int main(int argc, char **argv)
{
 int s;

 XPRBprob p("FolioSC");           // Initialize a new problem in BCL
 XPRBlinExp Risk, Na, Return, Cap;
 XPRBvar frac[NSHARES];           // Fraction of capital used per share

 for(s=0;s<NSHARES;s++)           // Decision variables (SCs)
   { frac[s] =p.newVar("frac", XPRB_SC, 0, 0.3); frac[s].setLim(0.1);  }

 for(s=0;s<NSHARES;s++) Return += RET[s]*frac[s];
 p.setObj(Return);                // Set the objective function

 for(s=0;s<NRISK;s++) Risk += frac[RISK[s]];
 p.newCtr(Risk <= 1.0/3);         // Limit the %age of high-risk values

 for(s=0;s<NNA;s++) Na += frac[NA[s]];
 p.newCtr(Na >= 0.5);             // Minimum amount of N American values

 for(s=0;s<NSHARES;s++) Cap += frac[s];
 p.newCtr(Cap == 1);              // Spend all the capital

 p.maxim("g");                    // Solve the problem
                                  // Solution printing
 cout << "Total return: " << p.getObjVal() << endl;
 for(s=0;s<NSHARES;s++)           // Solution printing
  cout << s << ": " << frac[s].getSol()*100 << "%" << endl;

 return 0;

}
```

The first Mosel Example

```
model "Portfolio optimization with MIP"
 uses "mmxprs"                      ! Use Xpress-Optimizer

 declarations
  SHARES = 1..10                    ! Set of shares
  RISK = {2,3,4,9,10}               ! Set of high-risk values among shares
  NA = {1,2,3,4}                    ! Set of shares issued in N.-America
  RET: array(SHARES) of real        ! Estimated return on investment
  frac: array(SHARES) of mpvar      ! Fraction of capital used per share
 end-declarations

 RET:= [5,17,26,12,8,9,7,6,31,21]
 Return:= sum(s in SHARES) RET(s)*frac(s)  ! Objective function

 sum(s in RISK) frac(s) <= 1/3      ! Limit the %age of high-risk values
 sum(s in NA) frac(s) >= 0.5        ! Minimum amount of N American values
 sum(s in SHARES) frac(s) = 1       ! Spend all the capital

 forall(s in SHARES) do             ! Upper/lower bounds on investment per share
  frac(s) <= 0.3; frac(s) is_semcont 0.1
 end-do

 maximize(Return)                        ! Solve the problem
 writeln("Total return: ", getobjval)  ! Solution printing
 forall(s in SHARES) writeln(s, ": ", getsol(frac(s))*100, "%")

end-model
```

The Mosel Variable Fixing Heuristic

```
model Coco
  uses "mmxprs"            ! Use the Xpress-MP Optimizer
  include "fixbv_pb.mos"
  include "fixbv_solve.mos"
  solution := solve        ! "solve" is a function
  writeln("The objective value is: ", solution)
end-model

! File fixbv_pb.mos
declarations
  RF = 1..2             ! Range of factories (f)
  RT = 1..4             ! Range of time periods (t)
  [...]
  openm: array(RF,RT) of mpvar ! 1 iff factory f is open in period t
end-declarations

[...]
forall(f in RF, t in 1..NT-1) Closed(f,t) := openm(f,t+1) <= openm(f,t)
forall(f in RF, t in RT) openm(f,t) is_binary


! File fixbv_solve.mos
function solve: real
  declarations
    TOL = 5.0E-4
    osol: array(RF,1..2) of real
  end-declarations
```

```
  setparam("XPRS_verbose", true)
  setparam("XPRS_PRESOLVE", 0); setparam("XPRS_CUTSTRATEGY", 0)
  maximize(XPRS_TOP, MaxProfit)  ! Solve LP at top node
  savebasis(1)
  forall(f in RF, t in 1..2) do
    osol(f,t) := getsol(openm(f,t))
    if(osol(f,t) < TOL) then
      setub(openm(f,t), 0.0)
    elif(1-osol(f,t) < TOL) then
      setlb(openm(f,t), 1.0)
    end-if
  end-do

  maximize(MaxProfit)
  solval := getobjval
  forall(f in RF, t in 1..2)
    if((osol(f,t) < TOL) or (1-osol(f,t) < TOL)) then
      setlb(openm(f,t), 0.0); setub(openm(f,t), 1.0)
    end-if
   end-if
  loadbasis(1)
  setparam("XPRS_MIPABSCUTOFF", solval)
  maximize(MaxProfit)
  returned := getobjval
end-function
```

Mosel Xpress-SLP Example

```
model "Polygon"

  uses "mmxslp"

  declarations
    N = 5
    area: mpvar
    rho, theta: array(1..N) of mpvar
  end-declarations

  forall(i in 1..N-1) do ! Initialization of SLP variables
    0.1 <= rho(i); rho(i) <= 1
    SLPDATA("IV", rho(i), 4*i*(N + 1 - i)/((N+1)^2))
    SLPDATA("IV", theta(i), M_PI*i/N)
  end-do
  ! Objective: sum of areas
  2*area = sum (i in 2..N-1) rho(i)*rho(i-1)*sin(theta(i)-theta(i-1))

  forall(i in 1..N-2, j in i+1..N-1)
              ! 3rd side of all triangles <= 1
    rho(i)^2 + rho(j)^2 - rho(i)*rho(j)*2*cos(theta(j)-theta(i)) <= 1

  ! Vertices in increasing order
  forall(i in 2..N-1) theta(i) >= theta(i-1) +.001

  theta(N-1) <= M_PI ! Boundary conditions

  SLPloadprob( area )
  SLPmaximize

  writeln("Area = ", getobjval)
  forall(i in 1..N-1)
    writeln("V", i, ": r=", getsol(rho(i)), " theta=", getsol(theta(i)))
end-model
```

User Graphs

```
mosel Ugraph
 uses "mmive", "mmsystem"
 declarations
  MACHINES=6; JOBS=6
  graphs, colors: array(1..MACHINES) of integer
  labels: array(1..JOBS) of integer
  curmachine, curjobs, n1, n2, n3: integer
 end-declarations

 colors:= [IVE_WHITE, IVE_YELLOW, IVE_CYAN, IVE_RED, IVE_GREEN, IVE_MAGENTA]
 fopen("schedule.dat", F_INPUT)

 forall (i in 1..MACHINES) do
  graphs(i):= IVEaddplot("Machine "+i, IVE_BLUE)
  labels(i):= IVEaddplot("Jobs for machine "+i, Color(i))
 end-do
end-do
 forall (i in 1..MACHINES) do
  readln(n1, n2)   ! Read machine number & number of jobs
  writeln("Machine ", n1, " Jobs:", n2)
  curmachine:= n; curjobs:= n2
  forall(j in 1..curjobs) do
    readln(n1, n2, n3) ! Read job number, start/finish times
    writeln("On machine ", curmachine, " job ", n1, " starts at ",
            n2, " finishes at ", n3)
    IVEdrawarrow(graphs(curmachine), n2, curmachine, n3, curmachine)
    IVEdrawlabel(labels(n1), (n2+n3)/2, curmachine,
                 "Job "+n1+"\r starts: "+n2+"\r ends: "+n3)
  end-do
 end-do

 IVEzoom(0, 0, 30, 7)
 fclose(F_INPUT)
end-model
```

The 5-leaper knight

```
model 'LEAPERRECTGR'
  uses 'mmxprs', 'mmsystem', 'mmive'

(! Closed TSP form of 5leaper
 Deals with rectangular chessboard
 Run with parameters NROW and NCOL (numbers of squares on the edges)
 e.g. mosel -s -c "exe leaprectgr 'NROW=9,NCOL=6'"
 NROW=NCOL=8 are the smallest possible values for square boards.

 A variant of problem in f5tour in the book
  Applications of optimization with Xpress-MP
  Christelle Guéret, Christian Prins & Marc Sevaux
  Translated and revised by Susanne Heipcke
  Dash Optimization, 2002, ISBN 0-9543503-0-8

 Mosel f5tour and drawing of subtours were written by Susanne Heipcke
 The generation of the legal 5leaps was written by Bob Daniel

 You will find a description of this problem and a few other
 model versions at the following address:  http://www.chlond.demon.co.uk/leapers/
```

```
 A square (i,j) on the rectangle is labelled sq := j + NCOL*(i-1)
 Given a square label sq, i := ceil(sq/NCOL),  j := sq - NCOL*(i-1)

 This variant eliminates all subtours it finds.
!)

  parameters
    NCOL = 8                            ! number of colums
    NROW = 8                            ! number of rows
    DISPLAY = true                      ! 'true' to see graphical output
  end-parameters

  forward function break_subtour: integer
  forward procedure print_sol
  forward procedure draw_sol
  forward procedure find_subtour(start: integer, TOUR: set of integer)

  declarations
    NSQ = NCOL*NROW                     ! number of squares
    RSQ = 1..NSQ                        ! range of squares

    nmoves, ntours, niterations, ntourstotal: integer
    nmovesfrom: array(RSQ) of integer   ! counts possible moves leaving

    x: dynamic array(RSQ,RSQ) of mpvar  ! 1 if we leap from sq1 to sq2
    NEXTX: array(RSQ) of integer        ! next square after sq in the solution
    square1, square2, ii, jj: integer
  end-declarations

! Draw the board
  if (DISPLAY) then
    board:= IVEaddplot("Board", IVE_BLACK)
    forall(i in 1..NROW+1) IVEdrawline(board, i-0.5, 0.5, i-0.5, NCOL+0.5)
    forall(j in 1..NCOL+1) IVEdrawline(board, 0.5, j-0.5, NROW+0.5, j-0.5)
  end-if

! For each pair of squares with a valid move...
  nmoves := 0
  forall(i1,i2 in 1..NCOL, j1,j2 in 1..NROW | (i1-i2)^2+(j1-j2)^2 = 25) do
    square1 := j1+NCOL*(i1-1)
    square2 := j2+NCOL*(i2-1)
    create(x(square1,square2))          ! create the 'x' binary decision variable
    x(square1,square2) is_binary
    nmovesfrom(square1) += 1            ! note a possible move from square
    nmoves += 1
  end-do
  writeln("There are ", nmoves, " possible moves")

! Check that each square is reachable
  forall(sq in RSQ | nmovesfrom(sq)=0) do
    ii := ceil(sq/NCOL)
    jj := sq-NCOL*(ii-1)
    writeln("**** Cannot go anywhere from square (", ii, ",", jj, ")")
    exit(0)
  end-do

! Objective (arbitrary)
  AnyObj := sum(sq1,sq2 in RSQ | exists(x(sq1,sq2)) ) x(sq1,sq2)

! Each square precedes one other
  forall(sq1 in RSQ) sum(sq2 in RSQ | exists(x(sq1,sq2)) ) x(sq1,sq2) = 1

! Each cell is preceded by one other
  forall(sq2 in RSQ) sum(sq1 in RSQ | exists(x(sq1,sq2)) ) x(sq1,sq2) = 1
```

```
! Optimizer settings to speed up the iterations
  setparam("XPRS_CUTSTRATEGY", 0)
  setparam("XPRS_PRESOLVE", 0)

! Solve the problem

  starttime := gettime
  niterations := 0

  while (TRUE) do
    minimize(AnyObj)

    if (getprobstat <> XPRS_OPT) then
      writeln("Problem is infeasible."); exit(0)
    end-if

    forall(sq in RSQ)
      NEXTX(sq):= integer(round(getsol(sum(sq2 in RSQ) sq2*x(sq,sq2) )))

    if (DISPLAY) then draw_sol; end-if

    ntours := break_subtour

    if (ntours=1) then break; end-if

    ! Print a log every 10 iterations
    niterations += 1
    ntourstotal += ntours
    if (niterations mod 10 = 0) then
     writeln(niterations, " (", gettime-starttime, " sec) tours eliminated: ",
               ntourstotal)
    end-if

  end-do

  print_sol

  writeln("Total time: ", gettime-starttime, " sec")



!-------------------------------------------------------------------
! Eliminate all subtours, and return number of (sub)tours
! - Find news subtours starting at each node in turn
! - If found complete tour, return 1
! - Otherwise add subtour elimination cut and continue
!
! Global variables required
!   RSQ: range of squares
!   NSQ: number of squares
!   NEXTX(i): square following i in current solution

  function break_subtour: integer
    declarations
      TOUR, SEENSOFAR: set of integer
      ntours: integer
    end-declarations

    SEENSOFAR := {}
    ntours := 0

    forall(startsquare in RSQ | startsquare not in SEENSOFAR) do
      ! Find (sub)tour containing {startsquare}
```

```
      find_subtour(startsquare, TOUR)
      SEENSOFAR += TOUR
      ntours += 1
      ! If found complete tour, return
      if (getsize(TOUR)>=NSQ) then break; end-if
      ! Add a subtour breaking constraint
      sum(sq in TOUR) x(sq,NEXTX(sq)) <= getsize(TOUR) - 1
    end-do

    returned := ntours
  end-function

!-------------------------------------------------------------------
! Find a subtour starting at square 'start'; result returned in TOUR

  procedure find_subtour(start: integer, TOUR: set of integer)
    declarations
      square: integer
    end-declarations

    TOUR := {}
    square := start
    repeat
      TOUR += {square}
      square := NEXTX(square)
    until (square=start)
  end-procedure

!-------------------------------------------------------------------
! Print the current solution
!
! Global variables required
!     RSQ: range of squares
!     NEXTX(i): square following i in current solution

  procedure print_sol
    declarations
      SEENSOFAR: set of integer
      square: integer
    end-declarations

    SEENSOFAR:={}
    forall(startsquare in RSQ | startsquare not in SEENSOFAR) do
      write(startsquare)
      square := startsquare
      repeat
        SEENSOFAR += {square}
        square := NEXTX(square)
        write(" - ", square)
      until (square=startsquare)
      writeln(' ')
    end-do
  end-procedure

!-------------------------------------------------------------------
! Draw the current solution
!
! Global variables required
!     RSQ: range of squares
!     NEXTX(sq): square following sq in current solution
!   NCOL: number of columns

  procedure draw_sol
```

```
    declarations
      COLOR, GRAPH: array(0..18) of integer
      SEENSOFAR: set of integer
      tour, square, gr, ii1, ii2, jj1, jj2: integer
    end-declarations

    COLOR := [IVE_RED, IVE_WHITE, IVE_GREEN, IVE_MAGENTA, IVE_YELLOW, IVE_CYAN,
              IVE_RGB(160,80,0), IVE_RGB(0,160,80), IVE_RGB(80,0,160),
              IVE_RGB(80,160,0), IVE_RGB(0,80,160), IVE_RGB(160,0,80),
              IVE_RGB(120,120,0), IVE_RGB(0,120,120), IVE_RGB(120,0,120),
              IVE_RGB(160,40,40), IVE_RGB(40,160,40), IVE_RGB(40,40,160),
              IVE_RGB(80,80,80)]

    IVEerase

    SEENSOFAR := {}
    tour := 1

    forall(startsquare in RSQ | startsquare not in SEENSOFAR) do
      gr := (tour-1) mod 19

      if (tour<=19) then
        GRAPH(gr) := IVEaddplot("Tour "+tour, COLOR(gr))
      end-if

      square := startsquare


      repeat
        ii1 := ceil(square/NCOL)
        jj1 := square-NCOL*(ii1-1)
        ii2 := ceil(NEXTX(square)/NCOL)
        jj2 := NEXTX(square)-NCOL*(ii2-1)

        IVEdrawarrow(GRAPH(gr), ii1, jj1, ii2, jj2)

        SEENSOFAR += {square}
        square := NEXTX(square)
      until (square=startsquare)
      tour += 1
    end-do

    ! Uncomment next line to pause at every iteration:
    !IVEpause("Click on pause button to continue")
  end-procedure


end-model
```