```
// -------------------------------------------------------------- -*- C++ -*-
// File: ilodiet.cpp
// Version 12.6.3
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2015. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------
//
// A dietary model.
//
// Input data:
// foodMin[j]      minimum amount of food j to use
// foodMax[j]      maximum amount of food j to use
// foodCost[j]     cost for one unit of food j
// nutrMin[i]      minimum amount of nutrient i
// nutrMax[i]      maximum amount of nutrient i
// nutrPer[i][j]   nutrition amount of nutrient i in food j
//
// Modeling variables:
// Buy[j]          amount of food j to purchase
//
// Objective:
// minimize sum(j) Buy[j] * foodCost[j]
//
// Constraints:
// forall nutr i: nutrMin[i] <= sum(j) Buy[j] * nutrPer[i][j] <= nutrMax[i]
//

#include <ilcplex/ilocplex.h>           // mandatory headers
ILOSTLBEGIN

void usage(const char* name) {
   cerr << endl;
   cerr << "usage:   " << name << " [options] <file>" << endl;
   cerr << "options: -c  build model by column" << endl;
   cerr << "         -i  use integer variables" << endl;
   cerr << endl;
}

void buildModelByRow(IloModel           mod,
                     IloNumVarArray     Buy,
                     const IloNumArray  foodMin,
                     const IloNumArray  foodMax,
                     const IloNumArray  foodCost,
                     const IloNumArray  nutrMin,
                     const IloNumArray  nutrMax,
                     const IloNumArray2 nutrPer,
                     IloNumVar::Type type) {

   IloEnv env = mod.getEnv();
   Buy.clear();
   IloNumVarArray tmp(env, foodMin, foodMax, type);
          // temporary array of variables created with bounds
   Buy.add(tmp);                    // copy temp array in permanent array
   tmp.end();             // destroy temporary object

   IloInt i, j;
   IloInt n = foodCost.getSize();
   IloInt m = nutrMin.getSize();

   mod.add(IloMinimize(env, IloScalProd(Buy,foodCost)));
                                        // create objective with scalar product
   for (i = 0; i < m; i++) {
      IloExpr expr(env);
      for (j = 0; j < n; j++) {
         expr += Buy[j] * nutrPer[i][j];
                           // fill expression of the constraint
      }
      mod.add(nutrMin[i] <= expr <= nutrMax[i]);
                           // create & add constraint to the model
      expr.end();      // destroy temporary object
   }
}

void buildModelByColumn(IloModel           mod,
                        IloNumVarArray     Buy,
                        const IloNumArray  foodMin,
                        const IloNumArray  foodMax,
                        const IloNumArray  foodCost,
                        const IloNumArray  nutrMin,
                        const IloNumArray  nutrMax,
                        const IloNumArray2 nutrPer,
                        IloNumVar::Type    type) {

   IloEnv env = mod.getEnv();
   IloInt i, j;
   IloInt n = foodCost.getSize();
   IloInt m = nutrMin.getSize();

   IloRangeArray range (env, nutrMin, nutrMax);
                  // temp array of constraints (to intersect with variables)
   mod.add(range);
   IloObjective cost = IloAdd(mod, IloMinimize(env));
                                          // create objective & add it to model

   for (j = 0; j < n; j++) {
      IloNumColumn col = cost(foodCost[j]);
                                          // cost of variable j in objective
      for (i = 0; i < m; i++) {
         col += range[i](nutrPer[i][j]);
                                 // coeff of variable j in constraint i
      }
      Buy.add(IloNumVar(col, foodMin[j], foodMax[j], type));
                           // create variable j & add to the model
      col.end();   // destroy temporary object
   }
   range.end();    // destroy temporary object
}
```

```
int main(int argc, char **argv)
{
   IloEnv env;  // create environment

   try {
      const char*     filename  = "../../../examples/data/diet.dat";
      IloBool         byColumn  = IloFalse;
      IloNumVar::Type varType   = ILOFLOAT;
      IloInt i;

      for (i = 1; i < argc; i++) {
         if (argv[i][0] == '-') {
            switch (argv[i][1]) {
            case 'c':
               byColumn = IloTrue;
               break;
            case 'i':
               varType = ILOINT;
               break;
            default:
               usage(argv[0]);
               throw (-1);
            }
         }
         else {
            filename = argv[i];
            break;
         }
      }

      ifstream file(filename);
      if ( !file ) {
         cerr << "ERROR: could not open file '" << filename
              << "' for reading" << endl;
         usage(argv[0]);
         throw (-1);
      }

      // model data

      IloNumArray  foodCost(env), foodMin(env), foodMax(env);
      IloNumArray  nutrMin(env), nutrMax(env);      // values arrays
      IloNumArray2 nutrPer(env);     // 2D matrix of values

//------------------------------------------------------------------
// Don't use because it assumes a specific file format.
// Just read the file as you would do normally and store data
// in IloArray<> or other structures
      file >> foodCost >> foodMin >> foodMax;
      file >> nutrMin >> nutrMax;
      file >> nutrPer;
 //------------------------------------------------------------------

      IloInt nFoods = foodCost.getSize();
      IloInt nNutr  = nutrMin.getSize();

      if ( foodMin.getSize() != nFoods ||
           foodMax.getSize() != nFoods ||
           nutrPer.getSize() != nNutr  ||
           nutrMax.getSize() != nNutr    ) {
         cerr << "ERROR: Data file '" << filename
              << "' contains inconsistent data" << endl;
         throw (-1);
      }

      for (i = 0; i < nNutr; i++) {
         if (nutrPer[i].getSize() != nFoods) {
            cerr << "ERROR: Data file '" << argv[0]
                 << "' contains inconsistent data" << endl;
            throw (-1);
         }
      }

      // Build model

      IloModel       mod(env);   // create model
      IloNumVarArray Buy(env);   // create array of variables

      if ( byColumn ) {
         buildModelByColumn(mod, Buy, foodMin, foodMax, foodCost,
                            nutrMin, nutrMax, nutrPer, varType);
      }
      else {
         buildModelByRow(mod, Buy, foodMin, foodMax, foodCost,
                         nutrMin, nutrMax, nutrPer, varType);
      }

      // Solve model

      IloCplex cplex(mod);      // create CPLEX object from model
      cplex.exportModel("diet.lp");  // write model in text file

      cplex.solve();               // solve model

        // retrieve solution
      cplex.out() << "solution status = " << cplex.getCplexStatus() << endl;
      cplex.out() << endl;
      cplex.out() << "cost   = " << cplex.getObjValue() << endl;
      for (i = 0; i < foodCost.getSize(); i++) {
         cplex.out() << "  Buy" << i << " = " << cplex.getValue(Buy[i]) << endl;
      }
   }
   catch (IloException& ex) {              // exception management
      cerr << "Error: " << ex << endl;
   }
   catch (...) {
      cerr << "Error" << endl;
   }

   env.end();
            // destroy environment and all objects tied to it

   return 0;
}
```

```
// -------------------------------------------------------------- -*- C++ -*-
// File: facility.cpp
// Version 12.6.3
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2015. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------


#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

typedef IloArray<IloNumArray>    FloatMatrix;       // 2D matrix of values
typedef IloArray<IloNumVarArray> NumVarMatrix;      // 2D matrix of variables


//------------------------------------------------------------------------
// Input data:
//
// capacity[j] : capacity of depot j
// fixedCost[j] : fixed cost of depot j
// cost[i][j] : variable cost from customer i to depot j
//
// Variables:
//
// open[j] : 1 if depot open, 0 otherwise
// supply[i][j] : 1 if customer i is linked to depot j
//
// objective:
// minimize sum(j) fixedCost(j) * open(j) + sum(i,j) cost(i,j) * supply(i,j)
//
// constraints:
//
// sum(j) supply(i,j) == 1     // each customer is linked to one depot
//
// sum(i) supply(i,j) <= capacity(j) * open(j)  // capacity of dépôts
//
//------------------------------------------------------------------------




int main(int argc, char **argv)
{
   IloEnv env;   // create environment
   try {
      IloInt i, j;
      IloNumArray capacity(env), fixedCost(env);
      FloatMatrix cost(env);
      IloInt      nbLocations;
      IloInt      nbClients;
```

```
      //--------------------------------------------------------------------
      // Don't use because it assumes a specific file format.
      // Just read the file as you would do normally and store data
      // in IloArray<> or other structures

      const char* filename  = "../../../examples/data/facility.dat";
      if (argc > 1)
         filename = argv[1];
      ifstream file(filename);
      if (!file) {
         cerr << "ERROR: could not open file '" << filename
              << "' for reading" << endl;
         cerr << "usage:   " << argv[0] << " <file>" << endl;
         throw(-1);
      }

      file >> capacity >> fixedCost >> cost;
      //--------------------------------------------------------------------

      nbLocations = capacity.getSize();
      nbClients   = cost.getSize();

      IloBool consistentData = (fixedCost.getSize() == nbLocations);
      for(i = 0; consistentData && (i < nbClients); i++)
         consistentData = (cost[i].getSize() == nbLocations);
      if (!consistentData) {
         cerr << "ERROR: data file '"
              << filename << "' contains inconsistent data" << endl;
         throw(-1);
      }

      IloNumVarArray open(env, nbLocations, 0, 1, ILOINT);
                          // array of binary variables of size nbLocations
      NumVarMatrix supply(env, nbClients);
      for(i = 0; i < nbClients; i++)
         supply[i] = IloNumVarArray(env, nbLocations, 0, 1, ILOINT);
           // 2D matrix of binary variables of size nbClients * nbLocations

      IloModel model(env);
      for(i = 0; i < nbClients; i++)
         model.add(IloSum(supply[i]) == 1);  // ∑j supply[i][j] == 1
      for(j = 0; j < nbLocations; j++) {
         IloExpr v(env);
         for(i = 0; i < nbClients; i++)
            v += supply[i][j];                  // ∑i supply[i][j]
         model.add(v <= capacity[j] * open[j]);
                                                // ∑i supply[i][j] <= cap[j]*open[j]
         v.end();     // destroy temp object
      }

      IloExpr obj = IloScalProd(fixedCost, open);
      for(i = 0; i < nbClients; i++) {
         obj += IloScalProd(cost[i], supply[i]); // ∑i,j supply[i][j]*cost[i][j]
      }
      model.add(IloMinimize(env, obj));
      obj.end();          // destroy temp object
```

```
      IloCplex cplex(env);
      cplex.extract(model);  // same as IloCplex cplex(model);
      cplex.solve();
      cplex.out() << "Solution status: " << cplex.getCplexStatus() << endl;

      IloNum tolerance =
                  cplex.getParam(IloCplex::Param::MIP::Tolerances::Integrality);
      cplex.out() << "Optimal value: " << cplex.getObjValue() << endl;
      for(j = 0; j < nbLocations; j++) {
         if (cplex.getValue(open[j]) >= 1 - tolerance) {
            cplex.out() << "Facility " << j << " is open, it serves clients ";
            for(i = 0; i < nbClients; i++) {
               if (cplex.getValue(supply[i][j]) >= 1 - tolerance)
                  cplex.out() << i << " ";
            }
            cplex.out() << endl;
         }
      }
   }
   catch(IloException& e) {             // exceptions management
      cerr  << " ERROR: " << e << endl;
   }
   catch(...) {
      cerr  << " ERROR" << endl;
   }
   env.end();
   return 0;
}
```

```
// -------------------------------------------------------------- -*- C++ -*-
// File: cutstock.cpp
// Version 12.6.3
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2015. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------
//
// Master problem:
//
// data:
// rollWidth       width of roll
// amount[j]       needed amount of each piece
// size[i]         width of each piece
//
// variables:
// Cut[i]          cutting patterns
// price[j]        dual variable of each cut constraint
// Use[j]          number of each pattern chosen
//
// Objective:
// minimize sum(i) Cut[i]
//
// Constraints:
// forall pieces: sum(i) use[i][j] * Cut[i] >= amount[j]
//
//
// Subproblem:
//
// Objective:
// minimize sum(j) (-price[j]) + 1
//
// Constraints:
// sum(j) size[j] * Use[j] <= rollwidth


#include <ilcplex/ilocplex.h>    // mandatory headers
ILOSTLBEGIN

#define RC_EPS 1.0e-6

static void readData (const char* filename, IloNum& rollWidth,
                      IloNumArray& size, IloNumArray& amount);
static void report1 (IloCplex& cutSolver, IloNumVarArray Cut,
                     IloRangeArray Fill);
static void report2 (IloAlgorithm& patSolver,
                     IloNumVarArray Use,
                     IloObjective obj);
static void report3 (IloCplex& cutSolver, IloNumVarArray Cut);
```

```
/// MAIN PROGRAM ///

int
main(int argc, char **argv)
{
   IloEnv env;  // create environment
   try {
      IloInt  i, j;

      IloNum      rollWidth;
      IloNumArray amount(env);   // arrays of values
      IloNumArray size(env);

      if ( argc > 1 )
         readData(argv[1], rollWidth, size, amount);
      else
         readData("../../../examples/data/cutstock.dat",
                  rollWidth, size, amount);

      /// CUTTING-OPTIMIZATION PROBLEM ///

      IloModel cutOpt (env);                        // Master problem model (MP)
      IloObjective   RollsUsed = IloAdd(cutOpt, IloMinimize(env));
                                       // MP objective created and added to model
      IloRangeArray  Fill = IloAdd(cutOpt,
                                   IloRangeArray(env, amount, IloInfinity));
                    // cut constraint array created and added to model
      IloNumVarArray Cut(env);    // array of variables

      IloInt nWdth = size.getSize();
      for (j = 0; j < nWdth; j++) {
         Cut.add(IloNumVar(RollsUsed(1) + Fill[j](int(rollWidth / size[j]))));
                         // initial variables created and added to model
      }

      IloCplex cutSolver(cutOpt);    // master problem solver

      /// PATTERN-GENERATION PROBLEM ///

      IloModel patGen (env);             // Subproblem model (SP)

      IloObjective ReducedCost = IloAdd(patGen, IloMinimize(env, 1));
                                       // SP objective created and added to model
      IloNumVarArray Use(env, nWdth, 0.0, IloInfinity, ILOINT);
                                                    // array of variables
      patGen.add(IloScalProd(size, Use) <= rollWidth);
                        // pattern constraints created and added to model

      IloCplex patSolver(patGen);    // subproblem solver

      /// COLUMN-GENERATION PROCEDURE ///

      IloNumArray price(env, nWdth);
                                  // dual values of cutting cuts (MP)
      IloNumArray newPatt(env, nWdth); // new cutiing patterns
```

```
      /// COLUMN-GENERATION PROCEDURE ///

      for (;;) {
         /// OPTIMIZE OVER CURRENT PATTERNS ///

         cutSolver.solve();            // solve MP
         report1 (cutSolver, Cut, Fill);

         /// FIND AND ADD A NEW PATTERN ///

         for (i = 0; i < nWdth; i++) {
           price[i] = -cutSolver.getDual(Fill[i]);  // dual values
         }
         ReducedCost.setLinearCoefs(Use, price);
                              // modify SP objective

         patSolver.solve();    // solve SP
         report2 (patSolver, Use, ReducedCost);

         if (patSolver.getValue(ReducedCost) > -RC_EPS) break;
                 // column generation procedure break condition

         patSolver.getValues(newPatt, Use);
                         // retrieve new column
         Cut.add( IloNumVar(RollsUsed(1) + Fill(newPatt)) );
           // create and add new variable to MP
      }

      cutOpt.add(IloConversion(env, Cut, ILOINT));
                                        // convert MP to MILP

      cutSolver.solve();     // solve MP (MILP)
      cout << "Solution status: " << cutSolver.getCplexStatus() << endl;
      report3 (cutSolver, Cut);
   }
   catch (IloException& ex) {   // exceptions management
      cerr << "Error: " << ex << endl;
   }
   catch (...) {
      cerr << "Error" << endl;
   }

   env.end();

   return 0;
}


static void readData (const char* filename, IloNum& rollWidth,
                      IloNumArray& size, IloNumArray& amount)
{
   ifstream in(filename);
   if (in) {
      in >> rollWidth;
      in >> size;
      in >> amount;
   }
   else {
      cerr << "No such file: " << filename << endl;
      throw(1);
   }
}

static void report1 (IloCplex& cutSolver, IloNumVarArray Cut,
                     IloRangeArray Fill)
{
   cout << endl;
   cout << "Using " << cutSolver.getObjValue() << " rolls" << endl;
   cout << endl;
   for (IloInt j = 0; j < Cut.getSize(); j++) {
      cout << "  Cut" << j << " = " << cutSolver.getValue(Cut[j]) << endl;
   }
   cout << endl;
   for (IloInt i = 0; i < Fill.getSize(); i++) {
      cout << "  Fill" << i << " = " << cutSolver.getDual(Fill[i]) << endl;
   }
   cout << endl;
}

static void report2 (IloAlgorithm& patSolver, IloNumVarArray Use,
                     IloObjective obj)
{
   cout << endl;
   cout << "Reduced cost is " << patSolver.getValue(obj) << endl;
   cout << endl;
   if (patSolver.getValue(obj) <= -RC_EPS) {
      for (IloInt i = 0; i < Use.getSize(); i++)  {
         cout << "  Use" << i << " = " << patSolver.getValue(Use[i]) << endl;
      }
      cout << endl;
   }
}

static void report3 (IloCplex& cutSolver, IloNumVarArray Cut)
{
   cout << endl;
   cout << "Best integer solution uses "
        << cutSolver.getObjValue() << " rolls" << endl;
   cout << endl;
   for (IloInt j = 0; j < Cut.getSize(); j++) {
      cout << "  Cut" << j << " = " << cutSolver.getValue(Cut[j]) << endl;
   }
}


/* Example Input file:
115
[25, 40, 50, 55, 70]
[50, 36, 24, 8, 30]
*/
```

```
// -------------------------------------------------------------- -*- C++ -*-
// File: foodmanufact.cpp
// Version 12.6.1
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2014. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------
//
// foodmanufact.cpp -  An implementation of an example from H.P.
//                     Williams' book Model Building in Mathematical
//                     Programming.  This example solves a
//                     food production planning problem.  It
//                     demonstrates the use of CPLEX's
//                     linearization capability.

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

typedef IloArray<IloNumVarArray> NumVarMatrix;
typedef IloArray<IloNumArray>    NumMatrix;

typedef enum { v1, v2, o1, o2, o3 } Product;
const IloInt nbMonths   = 6;
const IloInt nbProducts = 5;

int
main()
{
   IloEnv env;
   try {
      NumMatrix cost(env, nbMonths);
      cost[0]=IloNumArray(env, nbProducts, 110.0, 120.0, 130.0, 110.0, 115.0);
      cost[1]=IloNumArray(env, nbProducts, 130.0, 130.0, 110.0,  90.0, 115.0);
      cost[2]=IloNumArray(env, nbProducts, 110.0, 140.0, 130.0, 100.0,  95.0);
      cost[3]=IloNumArray(env, nbProducts, 120.0, 110.0, 120.0, 120.0, 125.0);
      cost[4]=IloNumArray(env, nbProducts, 100.0, 120.0, 150.0, 110.0, 105.0);
      cost[5]=IloNumArray(env, nbProducts,  90.0, 100.0, 140.0,  80.0, 135.0);

      // Variable definitions
      IloNumVarArray produce(env, nbMonths, 0, IloInfinity);
      NumVarMatrix   use(env, nbMonths);
      NumVarMatrix   buy(env, nbMonths);
      NumVarMatrix   store(env, nbMonths);
      IloInt i, p;
      for (i = 0; i < nbMonths; i++) {
         use[i]   = IloNumVarArray(env, nbProducts, 0, IloInfinity);
         buy[i]   = IloNumVarArray(env, nbProducts, 0, IloInfinity);
         store[i] = IloNumVarArray(env, nbProducts, 0, 1000);
      }
      IloExpr profit(env);

      IloModel model(env);

      // For each type of raw oil we must have 500 tons at the end
      for (p = 0; p < nbProducts; p++) {
        store[nbMonths-1][p].setBounds(500, 500);
      }

      // Constraints on each month
      for (i = 0; i < nbMonths; i++) {
         // Not more than 200 tons of vegetable oil can be refined
         model.add(use[i][v1] + use[i][v2] <= 200);

         // Not more than 250 tons of non-vegetable oil can be refined
         model.add(use[i][o1] + use[i][o2] + use[i][o3] <= 250);

         // Constraints on food composition
         model.add(3 * produce[i] <=
                   8.8 * use[i][v1] + 6.1 * use[i][v2] +
                   2   * use[i][o1] + 4.2 * use[i][o2] + 5 * use[i][o3]);
         model.add(8.8 * use[i][v1] + 6.1 * use[i][v2] +
                   2   * use[i][o1] + 4.2 * use[i][o2] + 5 * use[i][o3]
                   <= 6 * produce[i]);
         model.add(produce[i] == IloSum(use[i]));

         // Raw oil can be stored for later use
         if (i == 0) {
            for (IloInt p = 0; p < nbProducts; p++)
               model.add(500 + buy[i][p] == use[i][p] + store[i][p]);
         }
         else {
            for (IloInt p = 0; p < nbProducts; p++)
              model.add(store[i-1][p] + buy[i][p] == use[i][p] + store[i][p]);
         }

         // Logical constraints
         // The food cannot use more than 3 oils
         // (or at least two oils must not be used)
         model.add((use[i][v1] == 0) + (use[i][v2] == 0) + (use[i][o1] == 0) +
                   (use[i][o2] == 0) + (use[i][o3] == 0) >= 2);

         // When an oil is used, the quantity must be at least 20 tons
         for (p = 0; p < nbProducts; p++)
            model.add((use[i][p] == 0) || (use[i][p] >= 20));

         // If products v1 or v2 are used, then product o3 is also used
         model.add(IloIfThen(env, (use[i][v1] >= 20) || (use[i][v2] >= 20),
           use[i][o3] >= 20));

         // Objective function
         profit += 150 * produce[i] - IloScalProd(cost[i], buy[i]) -
                   5 * IloSum(store[i]);
      }

      // Objective function
      model.add(IloMaximize(env, profit));

      IloCplex cplex(model);
```

```
      if (cplex.solve()) {
         cout << "Solution status: " << cplex.getStatus() << endl;
         cout << " Maximum profit = " << cplex.getObjValue() << endl;
         for (IloInt i = 0; i < nbMonths; i++) {
            IloInt p;
            cout << " Month " << i << " " << endl;
            cout << "  . buy   ";
            for (p = 0; p < nbProducts; p++) {
               cout << cplex.getValue(buy[i][p]) << "\t ";
            }
            cout << endl;
            cout << "  . use   ";
            for (p = 0; p < nbProducts; p++) {
               cout << cplex.getValue(use[i][p]) << "\t ";
            }
            cout << endl;
            cout << "  . store ";
            for (p = 0; p < nbProducts; p++) {
               cout << cplex.getValue(store[i][p]) << "\t ";
            }
            cout << endl;
         }
      }
      else {
         cout << " No solution found" << endl;
      }
   }
   catch (IloException& ex) {
      cerr << "Error: " << ex << endl;
   }
   catch (...) {
      cerr << "Error" << endl;
   }
   env.end();
   return 0;
}
```

```
// -------------------------------------------------------------- -*- C++ -*-
// File: ilobendersatsp.cpp
// Version 12.6.3
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2015. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------
//
//
// Example ilobendersatsp.cpp solves a flow MILP model for an
// Asymmetric Traveling Salesman Problem (ATSP) instance
// through Benders decomposition.
//
// The arc costs of an ATSP instance are read from an input file.
// The flow MILP model is decomposed into a master ILP and a worker LP.
//
// The master ILP is then solved by adding Benders' cuts during
// the branch-and-cut process via the cut callback functions.
// The cut callback functions add to the master ILP violated Benders' cuts
// that are found by solving the worker LP.
//
// The example allows the user to decide if Benders' cuts have to be separated:
//
// a) Only to separate integer infeasible solutions.
// In this case, Benders' cuts are treated as lazy constraints through the
// class IloCplex::LazyConstraintCallbackI.
//
// b) Also to separate fractional infeasible solutions.
// In this case, Benders' cuts are treated as lazy constraints through the
// class IloCplex::LazyConstraintCallbackI.
// In addition, Benders' cuts are also treated as user cuts through the
// class IloCplex::UserCutCallbackI.
//
//
// To run this example, command line arguments are required:
//     ilobendersatsp.cpp {0|1} [filename]
// where
//     0         Indicates that Benders' cuts are only used as lazy constraints,
//               to separate integer infeasible solutions.
//     1         Indicates that Benders' cuts are also used as user cuts,
//               to separate fractional infeasible solutions.
//
//     filename  Is the name of the file containing the ATSP instance (arc
costs).
//               If filename is not specified, the instance
//               ../../../examples/data/atsp.dat is read
//
//
// ATSP instance defined on a directed graph G = (V, A)
// - V = {0, ..., n-1}, V0 = V \ {0}
// - A = {(i,j) : i in V, j in V, i != j }
// - forall i in V: delta+(i) = {(i,j) in A : j in V}
// - forall i in V: delta-(i) = {(j,i) in A : j in V}
// - c(i,j) = traveling cost associated with (i,j) in A
//
// Flow MILP model
//
// Modeling variables:
// forall (i,j) in A:
//    x(i,j) = 1, if arc (i,j) is selected
//           = 0, otherwise
// forall k in V0, forall (i,j) in A:
//    y(k,i,j) = flow of the commodity k through arc (i,j)
//
// Objective:
// minimize sum((i,j) in A) c(i,j) * x(i,j)
//
// Degree constraints:
// forall i in V: sum((i,j) in delta+(i)) x(i,j) = 1
// forall i in V: sum((j,i) in delta-(i)) x(j,i) = 1
//
// Binary constraints on arc variables:
// forall (i,j) in A: x(i,j) in {0, 1}
//
// Flow constraints:
// forall k in V0, forall i in V:
//    sum((i,j) in delta+(i)) y(k,i,j) - sum((j,i) in delta-(i)) y(k,j,i) =
q(k,i)
//    where q(k,i) =  1, if i = 0
//                 = -1, if k == i
//                 =  0, otherwise
//
// Capacity constraints:
// forall k in V0, for all (i,j) in A: y(k,i,j) <= x(i,j)
//
// Nonnegativity of flow variables:
// forall k in V0, for all (i,j) in A: y(k,i,j) >= 0
//

#include <string>

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

typedef IloArray<IloIntVarArray> Arcs;

// Declarations for functions in this program

void createMasterILP(IloModel mod, Arcs x, IloNumArray2 arcCost);

void createWorkerLP(IloCplex cplex, IloNumVarArray v, IloNumVarArray u,
                    IloObjective obj, IloInt numNodes);

IloBool separate(const Arcs x, const IloNumArray2 xSol, IloCplex cplex,
                 const IloNumVarArray v, const IloNumVarArray u,
                 IloObjective obj, IloExpr cutLhs, IloNum& cutRhs);

void usage(char *progname);
```

```
// Implementation class for the user-defined lazy constraint callback.
// The function BendersLazyCallback allows to add Benders' cuts as lazy
// constraints.
//
// Apply to integer nodes only
//
ILOLAZYCONSTRAINTCALLBACK5(BendersLazyCallback, Arcs, x, IloCplex, workerCplex,
                           IloNumVarArray, v, IloNumVarArray, u,
                           IloObjective, workerObj)
{
   IloInt i;
   IloEnv masterEnv = getEnv();
   IloInt numNodes = x.getSize();

   // Get the current x solution

   IloNumArray2 xSol(masterEnv, numNodes);
   for (i = 0; i < numNodes; ++i) {
      xSol[i] = IloNumArray(masterEnv);
      getValues(xSol[i], x[i]);
   }

   // Benders' cut separation

   IloExpr cutLhs(masterEnv);   // cut expression
   IloNum cutRhs;
   IloBool sepStat = separate(x, xSol, workerCplex, v, u, workerObj, cutLhs,
cutRhs);
   if ( sepStat ) {
      add(cutLhs >= cutRhs).end();
       // build cut, add it to the model and destroy temp object
   }

   // Free memory

   cutLhs.end();
   for (i = 0; i < numNodes; ++i)
      xSol[i].end();
   xSol.end();

   return;

} // END BendersLazyCallback


void usage (char *progname)
{
   cerr << "Usage:     " << progname << " {0|1} [filename]"            << endl;
   cerr << " 0:        Benders' cuts only used as lazy constraints,"   << endl;
   cerr << "           to separate integer infeasible solutions."      << endl;
   cerr << " 1:        Benders' cuts also used as user cuts,"          << endl;
   cerr << "           to separate fractional infeasible solutions."   << endl;
   cerr << " filename: ATSP instance file name."                       << endl;
   cerr << "           File ../../../examples/data/atsp.dat "
        << "used if no name is provided."                              << endl;

} // END usage
```

```
// Implementation class for the user-defined user cut callback.
// The function BendersUserCallback allows to add Benders' cuts as user cuts.
//
// Apply to non integer nodes only
//
ILOUSERCUTCALLBACK5(BendersUserCallback, Arcs, x, IloCplex, workerCplex,
                    IloNumVarArray, v, IloNumVarArray, u,
                    IloObjective, workerObj)
{
   // Skip the separation if not at the end of the cut loop

   if ( !isAfterCutLoop() )
      return;

   IloInt i;
   IloEnv masterEnv = getEnv();
   IloInt numNodes = x.getSize();

   // Get the current x solution

   IloNumArray2 xSol(masterEnv, numNodes);
   for (i = 0; i < numNodes; ++i) {
      xSol[i] = IloNumArray(masterEnv);
      getValues(xSol[i], x[i]);
   }

   // Benders' cut separation

   IloExpr cutLhs(masterEnv);         // cut expression
   IloNum cutRhs;
   IloBool sepStat = separate(x, xSol, workerCplex, v, u, workerObj, cutLhs,
cutRhs);
   if ( sepStat ) {
      add(cutLhs >= cutRhs).end();
       // build cut, add it to the model and destroy temp object
   }

   // Free memory

   cutLhs.end();
   for (i = 0; i < numNodes; ++i)
      xSol[i].end();
   xSol.end();

   return;

} // END BendersUserCallback
```

```
int main(int argc, char **argv)
{
   IloEnv masterEnv;   // 2 separate environments (only 1 was necessary)
   IloEnv workerEnv;

   try {
      const char* fileName = "../../../examples/data/atsp.dat";

       // Check the command line arguments

      if ( argc != 2 && argc != 3) {
         usage (argv[0]);
         throw (-1);
      }

      if ( (argv[1][0] != '1' && argv[1][0] != '0') ||
           argv[1][1] != '\0' ) {
         usage (argv[0]);
         throw (-1);
      }

      IloBool separateFracSols = ( argv[1][0] == '0' ? IloFalse : IloTrue );

      masterEnv.out() << "Benders' cuts separated to cut off: ";
      if ( separateFracSols ) {
         masterEnv.out() << "Integer and fractional infeasible solutions." <<
endl;
      }
      else {
         masterEnv.out() << "Only integer infeasible solutions." << endl;
      }

      if ( argc == 3 )  fileName = argv[2];

      // Read arc_costs from data file (17 city problem)

      IloNumArray2 arcCost(masterEnv);
      ifstream data(fileName);
      if ( !data ) throw(-1);
      data >> arcCost;
      data.close();

      // create master ILP

      IloModel masterMod(masterEnv, "atsp_master");
      IloInt numNodes = arcCost.getSize();
      Arcs x(masterEnv, numNodes);
      createMasterILP(masterMod, x, arcCost);

      // Create worker IloCplex algorithm and worker LP for Benders' cuts
separation

      IloCplex workerCplex(workerEnv);
      IloNumVarArray v(workerEnv);
      IloNumVarArray u(workerEnv);
      IloObjective workerObj(workerEnv);   // SP objective
      createWorkerLP(workerCplex, v, u, workerObj, numNodes);

      // Set up the cut callback to be used for separating Benders' cuts

      IloCplex masterCplex(masterMod);
      masterCplex.setParam(IloCplex::Param::Preprocessing::Presolve, IloFalse);
                           // set presolve to OFF for MP


      // Set the maximum number of threads to 1.
      // This instruction is redundant: If MIP control callbacks are registered,
      // then by default CPLEX uses 1 (one) thread only.
      // Note that the current example may not work properly if more than 1
threads
      // are used, because the callback functions modify shared global data.
      // We refer the user to the documentation to see how to deal with multi-
thread
      // runs in presence of MIP control callbacks.

      masterCplex.setParam(IloCplex::Param::Threads, 1);

      // Turn on traditional search for use with control callbacks

      masterCplex.setParam(IloCplex::Param::MIP::Strategy::Search,
                           IloCplex::Traditional);

      masterCplex.use(BendersLazyCallback(masterEnv, x, workerCplex, v, u,
workerObj));
      if ( separateFracSols )
         masterCplex.use(BendersUserCallback(masterEnv, x, workerCplex, v, u,
workerObj));

      // Solve the model and write out the solution

      if ( masterCplex.solve() ) {

         IloCplex::CplexStatus solStatus= masterCplex.getCplexStatus();
         masterEnv.out() << endl << "Solution status: " << solStatus << endl;

         masterEnv.out() << "Objective value: "
                         << masterCplex.getObjValue() << endl;

         if ( solStatus == IloCplex::Optimal ) {

            // Write out the optimal tour

            IloInt i, j;
            IloNumArray2 sol(masterEnv, numNodes);
            IloIntArray succ(masterEnv, numNodes);

            for (j = 0; j < numNodes; ++j)
               succ[j] = -1;

            for (i = 0; i < numNodes; i++) {
               sol[i] = IloNumArray(masterEnv);
               masterCplex.getValues(sol[i], x[i]);
               for(j = 0; j < numNodes; j++) {
                  if ( sol[i][j] > 1e-03 ) succ[i] = j;
```

```
            }
         }

         masterEnv.out() << "Optimal tour:" << endl;
         i = 0;
         while ( succ[i] != 0 ) {
            masterEnv.out() << i << ", ";
            i = succ[i];
         }
         masterEnv.out() << i << endl;
      }
      else {
         masterEnv.out() << "Solution status is not Optimal" << endl;
      }
   }
   else {
      masterEnv.out() << "No solution available" << endl;
   }
}

catch (const IloException& e) {
   cerr << "Exception caught: " << e << endl;
}
catch (...) {
   cerr << "Unknown exception caught!" << endl;
}

// Close the environments

masterEnv.end();
workerEnv.end();
return 0;

} // END main

// This routine creates the master ILP (arc variables x and degree constraints).
//
// Modeling variables:
// forall (i,j) in A:
//    x(i,j) = 1, if arc (i,j) is selected
//           = 0, otherwise
//
// Objective:
// minimize sum((i,j) in A) c(i,j) * x(i,j)
//
// Degree constraints:
// forall i in V: sum((i,j) in delta+(i)) x(i,j) = 1
// forall i in V: sum((j,i) in delta-(i)) x(j,i) = 1
//
// Binary constraints on arc variables:
// forall (i,j) in A: x(i,j) in {0, 1}
//
void createMasterILP(IloModel mod, Arcs x, IloNumArray2 arcCost)
{
   IloInt i, j;
   IloEnv env = mod.getEnv();
   IloInt numNodes = x.getSize();

   // Create variables x(i,j) for (i,j) in A
   // For simplicity, also dummy variables x(i,i) are created.
   // Those variables are fixed to 0 and do not participate to
   // the constraints.

   char varName[100];
   for (i = 0; i < numNodes; ++i) {
      x[i] = IloIntVarArray(env, numNodes, 0, 1);
      x[i][i].setBounds(0, 0);
      for (j = 0; j < numNodes; ++j) {
         sprintf(varName, "x.%d.%d", (int) i, (int) j);
         x[i][j].setName(varName);
      }
      mod.add(x[i]);
        // add matrix of variables X, one vector (dimension) at a time
   }

   // Create objective function: minimize sum((i,j) in A ) c(i,j) * x(i,j)

   IloExpr obj(env);
   for (i = 0; i < numNodes; ++i) {
      arcCost[i][i] = 0;
      obj += IloScalProd(x[i], arcCost[i]);
   }
   mod.add(IloMinimize(env, obj));
   obj.end();

   // Add the out degree constraints.
   // forall i in V: sum((i,j) in delta+(i)) x(i,j) = 1

   for (i = 0; i < numNodes; ++i) {
      IloExpr expr(env);
      for (j = 0;   j < i; ++j)
         expr += x[i][j];
      for (j = i+1; j < numNodes; ++j)
         expr += x[i][j];
      mod.add(expr == 1);
      expr.end();
   }

   // Add the in degree constraints.
   // forall i in V: sum((j,i) in delta-(i)) x(j,i) = 1

   for (i = 0; i < numNodes; i++) {
      IloExpr expr(env);
      for (j = 0;   j < i; j++)
         expr += x[j][i];
      for (j = i+1; j < numNodes; j++)
         expr += x[j][i];
      mod.add(expr == 1);
      expr.end();
   }

}// END createMasterILP
```

```
// This routine set up the IloCplex algorithm to solve the worker LP, and
// creates the worker LP (i.e., the dual of flow constraints and
// capacity constraints of the flow MILP)
//
// Modeling variables:
// forall k in V0, i in V:
//    u(k,i) = dual variable associated with flow constraint (k,i)
//
// forall k in V0, forall (i,j) in A:
//    v(k,i,j) = dual variable associated with capacity constraint (k,i,j)
//
// Objective:
// minimize sum(k in V0) sum((i,j) in A) x(i,j) * v(k,i,j)
//          - sum(k in V0) u(k,0) + sum(k in V0) u(k,k)
//
// Constraints:
// forall k in V0, forall (i,j) in A: u(k,i) - u(k,j) <= v(k,i,j)
//
// Nonnegativity on variables v(k,i,j)
// forall k in V0, forall (i,j) in A: v(k,i,j) >= 0
//
void createWorkerLP(IloCplex cplex, IloNumVarArray v, IloNumVarArray u,
                    IloObjective obj, IloInt numNodes)
{

   IloInt i, j, k;
   IloEnv env = cplex.getEnv();
   IloModel mod(env, "atsp_worker");

   // Set up IloCplex algorithm to solve the worker LP

   cplex.extract(mod);
   cplex.setOut(env.getNullStream());  // pas d'output de CPLEX

   // Turn off the presolve reductions and set the CPLEX optimizer
   // to solve the worker LP with primal simplex method.

   cplex.setParam(IloCplex::Param::Preprocessing::Reduce, 0);
   cplex.setParam(IloCplex::Param::RootAlgorithm, IloCplex::Primal);

   // Create variables v(k,i,j) forall k in V0, (i,j) in A
   // For simplicity, also dummy variables v(k,i,i) are created.
   // Those variables are fixed to 0 and do not partecipate to
   // the constraints.

   IloInt numArcs  = numNodes * numNodes;
   IloInt vNumVars = (numNodes-1) * numArcs;
   IloNumVarArray vTemp(env, vNumVars, 0, IloInfinity);
   for (k = 1; k < numNodes; ++k) {
      for (i = 0; i < numNodes; ++i) {
         vTemp[(k-1)*numArcs + i *numNodes + i].setBounds(0, 0);
      }
   }
   v.clear();
   v.add(vTemp);   // copy temporary vector of variables
                   // in permananet one
   vTemp.end();
   mod.add(v);

   // Set names for variables v(k,i,j)

   for (k = 1; k < numNodes; ++k) {
      for(i = 0; i < numNodes; ++i) {
         for(j = 0; j < numNodes; ++j) {
            char varName[100];
            sprintf(varName, "v.%d.%d.%d", (int) k, (int) i, (int) j);
            v[(k-1)*numArcs + i*numNodes + j].setName(varName);
         }
      }
   }

   // Associate indices to variables v(k,i,j)

   IloIntArray vIndex(env, vNumVars);
   for (j = 0; j < vNumVars; ++j)
   {
      vIndex[j] = j;
      v[j].setObject(&vIndex[j]);  // assign index (ptr) to variable
   }

   // Create variables u(k,i) forall k in V0, i in V

   IloInt uNumVars = (numNodes-1) * numNodes;
   IloNumVarArray uTemp(env, uNumVars, -IloInfinity, IloInfinity);
   u.clear();
   u.add(uTemp);  // copy temporary vector of variables
                  // in permananet one
   uTemp.end();
   mod.add(u);

   // Set names for variables u(k,i)

   for (k = 1; k < numNodes; ++k) {
      for(i = 0; i < numNodes; ++i) {
         char varName[100];
         sprintf(varName, "u.%d.%d", (int) k, (int) i);
         u[(k-1)*numNodes + i].setName(varName);
      }
   }

   // Associate indices to variables u(k,i)

   IloIntArray uIndex(env, uNumVars);
   for (j = 0; j < uNumVars; ++j)
   {
      uIndex[j] = vNumVars + j;
      u[j].setObject(&uIndex[j]);  // assign index (ptr) to variable
   }

   // Initial objective function is empty

   obj.setSense(IloObjective::Minimize);
   mod.add(obj);
```

```
   // Add constraints:
   // forall k in V0, forall (i,j) in A: u(k,i) - u(k,j) <= v(k,i,j)

   for (k = 1; k < numNodes; ++k) {
      for(i = 0; i < numNodes; ++i) {
         for(j = 0; j < numNodes; ++j) {
            if ( i != j ) {
               IloExpr expr(env);
               expr -= v[(k-1)*numArcs + i*(numNodes) + j];
               expr += u[(k-1)*numNodes + i];
               expr -= u[(k-1)*numNodes + j];
               mod.add(expr <= 0);
               expr.end();
            }
         }
      }
   }

}// END createWorkerLP

// This routine separates Benders' cuts violated by the current x solution.
// Violated cuts are found by solving the worker LP
//
IloBool separate(const Arcs x, const IloNumArray2 xSol, IloCplex cplex,
                 const IloNumVarArray v, const IloNumVarArray u,
                 IloObjective obj, IloExpr cutLhs, IloNum& cutRhs)
{
   IloBool violatedCutFound = IloFalse;

   IloEnv env = cplex.getEnv();
   IloModel mod = cplex.getModel();

   IloInt numNodes = xSol.getSize();
   IloInt numArcs  = numNodes * numNodes;
   IloInt i, j, k, h;

   // Update the objective function in the worker LP:
   // minimize sum(k in V0) sum((i,j) in A) x(i,j) * v(k,i,j)
   //          - sum(k in V0) u(k,0) + sum(k in V0) u(k,k)

   mod.remove(obj);               // remove precedent objective
   IloExpr objExpr = obj.getExpr();
   objExpr.clear();
   for (k = 1; k < numNodes; ++k) {
      for (i = 0; i < numNodes; ++i) {
         for (j = 0; j < numNodes; ++j) {
               objExpr +=  xSol[i][j] * v[(k-1)*numArcs + i*numNodes + j];
         }
      }
   }
   for (k = 1; k < numNodes; ++k) {
      objExpr += u[(k-1)*numNodes + k];
      objExpr -= u[(k-1)*numNodes];
   }
   obj.setExpr(objExpr);
   mod.add(obj);              // add new objective
   objExpr.end();

   // Solve the worker LP

   cplex.solve();

   // A violated cut is available iff the solution status is Unbounded

   if ( cplex.getCplexStatus() == IloCplex::Unbounded ) {

      IloInt vNumVars = (numNodes-1) * numArcs;
      IloNumVarArray var(env);
      IloNumArray val(env);

      // Get the violated cut as an unbounded ray of the worker LP

      cplex.getRay(val, var);

      // Compute the cut from the unbounded ray. The cut is:
      // sum((i,j) in A) (sum(k in V0) v(k,i,j)) * x(i,j) >=
      // sum(k in V0) u(k,0) - u(k,k)

      cutLhs.clear();
      cutRhs = 0.;

      for (h = 0; h < val.getSize(); ++h) {

         IloInt *index_p = (IloInt*) var[h].getObject();
            // récupère l'index (ptr) de la variable
         IloInt index = *index_p;

         if ( index >= vNumVars ) {
            index -= vNumVars;
            k = index / numNodes + 1;
            i = index - (k-1)*numNodes;
            if ( i == 0 )
               cutRhs += val[h];     // update RHS
            else if ( i == k )
               cutRhs -= val[h];
         }
         else {
            k = index / numArcs + 1;
            i = (index - (k-1)*numArcs) / numNodes;
            j = index - (k-1)*numArcs - i*numNodes;
            cutLhs += val[h] * x[i][j];
                           // update constraint expression
         }
      }

      var.end();
      val.end();

      violatedCutFound = IloTrue;
   }

   return violatedCutFound;

} // END separate
```

```
// ------------------------------------------------------------ -*- C++ -*-
// File: iloadmipex8.cpp
// Version 12.8.0
// ------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2017. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// ------------------------------------------------------------------------
//
// iloadmipex8.c -- Solve a capacitated facility location problem with cutting
// planes using the new callback interface.
//
// We are given a set of candidate locations J and a set of clients C.
// Facilities should be opened in some of the candidate location so that the
// clients demands can be served. The facility location problem consists in
// deciding in which locations facilities should be opened and assigning each
// clients to a facilities at a minimum cost.
//
// A fixed cost is associated to opening a facility, and a linear cost is
// associated to the demand supplied from a given facility to a client.
//
// Furthermore, each facility has a capacity and can only serve |C| - 1
// clients. Note that in the variant of the problem considered here, each
// client is served by only one facility.
//
// The problem is formulated as a mixed integer linear program using binary
// variables: used[j], for all locations j in J, indicating if a facility is
// opened in location j; supply[c][j], for each client c in C and facility j in
// J, indicating the demand of client c supplied from facility j.
//
// Then, the following model formulates the facility location problem.
//
//  Minimize sum(j in J) fixedCost[j] * used[j] +
//           sum(j in J) sum(c in C) cost[c][j] * supply[c][j]
//  Subject to:
//    sum(j in J) supply[c][j] == 1                    for all c in C,
//    sum(c in C) supply[c][j] <= (|C| - 1) * used[j]  for all j in J,
//    supply[c][j] in {0, 1}                   for all c in C, j in J,
//    used[j] in {0, 1}                                for all j in J.
//
// The first set of constraints are the demand constraints ensuring that the
// demand of each client is satisfied. The second set of constraints are the
// capacity constraints, ensuring that if a facility is placed in location j
// the capacity of that facility is not exceeded.
//
// The program in this file, formulates a facility location problem and solves
// it.  Furthermore, different cutting planes methods are implemented using the
// callback API to help the solution of the problem:
//
// - Disaggregated capacity cuts separated algorithmically (see function
//   desegregate for details),
//
// - Disagregated capacity cuts separated using a cut table (see function
//   cutsfromtable),
//
// - Capacity constraints separated as lazy constraints (see function
//   lazycapacity).
//
// Those different methods are invoked using the callback API.
//
// See the usage message below for how to switch between these options.

#include <sstream>
#include <ilcplex/ilocplex.h>

using namespace std;

static void usage(const char *progname)
{
   cerr << "Usage: " << progname << "[options...]" << endl
        << " By default, a user cut callback is useed to dynamically"    << endl
        << " separate constraints."  << endl << endl
        << " Supported options are:" << endl
        << "  -table       Instead of the default behavior, use a"      << endl
        << "               static table that holds all cuts and"        << endl
        << "               scan that table for violated cuts."          << endl
        << "  -no-cuts     Do not separate any cuts."                   << endl
        << "  -lazy        Do not include capacity constraints in the"  << endl
        << "               model. Instead, separate them from a lazy"   << endl
        << "               constraint callback."                        << endl
        << "  -data=<dir>  Specify the directory in which the data"     << endl
        << "               file facility.dat is located."               << endl
      ;
   exit(2);
}

typedef IloArray<IloNumArray>    FloatMatrix;
typedef IloArray<IloNumVarArray> NumVarMatrix;

#define EPS 1e-6 // epsilon useed for violation of cuts

// This is the class implementing the callback for facility location.
//
// It has three main functions:
//    - disaggregateCutSep: add disagrregated constraints linking clients and
location.
//    - tableCuts: do the same using a cut table.
//    - lazyCapacity: adds the capacity constraint as a lazy constrain.
//
class FacilityCallback: public IloCplex::Callback::Function {
   private:
      /* Empty constructor is forbidden. */
      FacilityCallback ()
      {}

      /* Copy constructor is forbidden. */
      FacilityCallback(const FacilityCallback &tocopy);

      /* Variables for opening facilities. */
      IloNumVarArray opened;
```

```
      /* Variables representing amount supplied from facility j to customer c.
*/
      NumVarMatrix supply;

      /* Table of cuts that can be separated. */
      IloRangeArray cuts;

   public:
      /* Constructor with data */
      FacilityCallback(const IloNumVarArray &_opened,
                       const NumVarMatrix &_supply):
         opened(_opened), supply(_supply), cuts(opened.getEnv())
      {}

      // Separate the disaggregated capacity constraints.
      // In the model we have for each location j the constraint
      //    sum(c in clients) supply[c][j] <= (nbClients-1) * opened[j]
      // Clearly, a client can only be serviced from a location that is opened,
      // so we also have a constraint
      //    supply[c][j] <= opened[j]
      // that must be satisfied by every feasible solution. These constraints
tend
      // to be violated in LP relaxation. In this callback we separate them.
      inline void
      separateDisagregatedCuts (const IloCplex::Callback::Context &context)
const {
         IloInt const nbLocations = opened.getSize();
         IloInt const nbClients = supply.getSize();

         // For each j and c check whether in the current solution (obtained by
         // calls to getValue()) we have supply[c][j] > opened[j]. If so, then
we have
         // found a violated constraint and add it as a cut.
         for (IloInt j = 0; j < nbLocations; ++j) {
            for (IloInt c = 0; c < nbClients; ++c) {
               IloNum const s = context.getRelaxationPoint(supply[c][j]);
               IloNum const o = context.getRelaxationPoint(opened[j]);
               if ( s > o + EPS) {
                  cout << "Adding: " << supply[c][j].getName() << " <= "
                       << opened[j].getName() << " [" << s << " > " << o << "]"
<< endl;
                  context.addUserCut( supply[c][j] - opened[j] <= 0,
                                      IloCplex::UseCutPurge, IloFalse);
               }
            }
         }
      }

      // Variant of separateDisagregatedCuts that looks for violated cuts in
      // the static table cuts.
      inline void
      cutsFromTable (const IloCplex::Callback::Context &context) const {
         for (IloInt i = 0; i < cuts.getSize(); ++i) {
            const IloRange& cut = cuts[i];
            IloNum const lhs = context.getRelaxationValue(cut.getExpr());
            if (lhs < cut.getLB() - EPS || lhs > cut.getUB() + EPS ) {
               cout << "Adding: " << cut << " [lhs = " << lhs << "]" << endl;
               context.addUserCut(cut, IloCplex::UseCutPurge, IloFalse);
            }
         }
      }

      // Function to populate the cut table used by cutsFromTable.
      void populateCutTable (const IloEnv &env)
      {
         IloInt const nbLocations = opened.getSize();
         IloInt const nbClients = supply.getSize();
         // Generate all disaggregated constraints and put them into a
         // table that is scanned by the callback.
         cuts = IloRangeArray(env);
         for (IloInt j = 0; j < nbLocations; ++j)
            for (IloInt c = 0; c < nbClients; ++c)
               cuts.add(supply[c][j] - opened[j] <= 0.0);
      }

      // Lazy constraint callback to enforce the capacity constraints.
      // If used then the callback is invoked for every integer feasible
solution
      // CPLEX finds. For each location j it checks whether constraint
      //    sum(c in C) supply[c][j] <= (|C| - 1) * opened[j]
      // is satisfied. If not then it adds the violated constraint as lazy
constraint.
      inline void
      lazyCapacity (const IloCplex::Callback::Context &context) const {
         IloInt const nbLocations = opened.getSize();
         IloInt const nbClients = supply.getSize();
         if ( !context.isCandidatePoint() )
            throw IloCplex::Exception(-1, "Unbounded solution");
         for (IloInt j = 0; j < nbLocations; ++j) {
            IloNum isUsed = context.getCandidatePoint(opened[j]);
            IloNum served = 0.0; // Number of clients currently served from j
            for (IloInt c = 0; c < nbClients; ++c)
               served += context.getCandidatePoint(supply[c][j]);
            if ( served > (nbClients - 1.0) * isUsed + EPS ) {
               IloNumExpr sum = IloExpr(context.getEnv());
               for (IloInt c = 0; c < nbClients; ++c)
                  sum += supply[c][j];
               sum -= (nbClients - 1) * opened[j];
               cout << "Adding lazy capacity constraint " << sum << " <= 0" <<
endl;
               context.rejectCandidate(sum <= 0.0);
               sum.end();
            }
         }
      }

      // This is the function that we have to implement and that CPLEX will call
      // during the solution process at the places that we asked for.
      virtual void invoke (const IloCplex::Callback::Context &context);

      /// Destructor
      virtual ~FacilityCallback();
};
```

```
/* Implementation of the invoke function */
void
FacilityCallback::invoke (const IloCplex::Callback::Context &context)
{
   if ( context.inRelaxation() ) {
      if ( cuts.getSize() > 0 ) {
         cutsFromTable(context);
      }
      else {
         separateDisagregatedCuts(context);
      }
   }

   if ( context.inCandidate() )
         lazyCapacity (context);

}

/// Destructor
FacilityCallback::~FacilityCallback()
{
   cuts.endElements();
}

int
main(int argc, char **argv)
{
   IloEnv env;
   try {
      // Set default arguments and parse command line.
      char const *datadir = "../../../examples/data";
      bool tableCuts = false;
      bool lazy = false;
      bool separateCuts = true;

      for (int i = 1; i < argc; ++i) {
         if ( ::strncmp(argv[i], "-data=", 6) == 0 )
            datadir = argv[i] + 6;
         else if ( ::strcmp(argv[i], "-table") == 0 )
            tableCuts = true;
         else if ( ::strcmp(argv[i], "-lazy") == 0 )
            lazy = true;
         else if ( ::strcmp(argv[i], "-no-cuts") == 0 )
            separateCuts = false;
         else {
            cerr << "Unknown argument " << argv[i] << endl;
            usage(argv[0]);
         }
      }

      // Setup input file name and open the file.
      stringstream filename;
      filename << datadir << "/" << "facility.dat";
      ifstream file(filename.str().c_str());
      if (!file) {
         cerr << "ERROR: could not open file '" << filename.str()
              << "' for reading" << endl;
         usage(argv[0]);
      }

      // Input data.
      IloNumArray fixedCost(env);
      FloatMatrix cost(env);
      file >> fixedCost >> cost;
      IloInt nbLocations = fixedCost.getSize();
      IloInt nbClients   = cost.getSize();

      // Create variables.
      // - opened[j]      If location j is opened.
      // - supply[c][j] Amount shipped from location j to client c. This is a
      //                number in [0,1] and specifies the percentage of c's
      //                demand that is served from location i.
      IloNumVarArray opened(env, nbLocations, 0, 1, ILOINT);
      opened.setNames("opened");
      NumVarMatrix supply(env, nbClients);
      for (IloInt c = 0; c < nbClients; ++c) {
         supply[c] = IloNumVarArray(env, nbLocations, 0, 1, ILOINT);
         std::stringstream s;
         s << "supply(" << c << ")";
         supply[c].setNames(s.str().c_str());
      }

      IloModel model(env);
      // The supply for each client must sum to 1, i.e., the demand of each
      // client must be met.
      for (IloInt c = 0; c < nbClients; ++c)
         model.add(IloSum(supply[c]) == 1);

      // Capacity constraint for each location. We just require that a single
      // location cannot serve all clients, that is, the capacity of each
      // location is nbClients-1. This makes the model a little harder to
      // solve and allows us to separate more cuts.
      if ( !lazy ) {
         for (IloInt j = 0; j < nbLocations; ++j) {
            IloExpr v(env);
            for (IloInt c = 0; c < nbClients; ++c)
               v += supply[c][j];
            model.add(v <= (nbClients - 1) * opened[j]);
            v.end();
         }
      }

      // Objective function. We have the fixed cost for openeding a location
      // and the cost proportional to the amount that is shipped from a
      // location.
      IloExpr obj = IloScalProd(fixedCost, opened);
      for (IloInt c = 0; c < nbClients; ++c) {
         obj += IloScalProd(cost[c], supply[c]);
      }
      model.add(IloMinimize(env, obj));
      obj.end();

      IloCplex cplex(env);
```

```
      cplex.extract(model);

      // Tweak some CPLEX parameters so that CPLEX has a harder time to
      // solve the model and our cut separators can actually kick in.
      cplex.setParam(IloCplex::Param::MIP::Strategy::HeuristicFreq, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::MIRCut, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::Implied, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::Gomory, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::FlowCovers, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::PathCut, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::LiftProj, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::ZeroHalfCut, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::Cliques, -1);
      cplex.setParam(IloCplex::Param::MIP::Cuts::Covers, -1);

      // Now we get to setting up the callback.
      // We instanciate a FacilityCallback and set the contextMask parameter.
      FacilityCallback fcCallback(opened,supply);
      CPXLONG contextMask = 0;
      if ( separateCuts ) {
         contextMask |= IloCplex::Callback::Context::Id::Relaxation;
         if ( tableCuts ) {
            fcCallback.populateCutTable(env);
         }
      }

      if ( lazy )
         contextMask |= IloCplex::Callback::Context::Id::Candidate;

      // If contextMask is not zero we add the callback.
      if ( contextMask != 0 )
         cplex.use(&fcCallback, contextMask);

      if ( !cplex.solve() )
         throw IloAlgorithm::Exception("No feasible solution found");

      IloNum tolerance = cplex.getParam(
         IloCplex::Param::MIP::Tolerances::Integrality);

      cout << "Solution status:                   " << cplex.getStatus() <<
endl;
      cout << "Nodes processed:                   " << cplex.getNnodes() <<
endl;
      cout << "Active user cuts/lazy constraints: " <<
cplex.getNcuts(IloCplex::CutUser) << endl;
      cout << "Optimal value:                     " << cplex.getObjValue() <<
endl;
      for (IloInt j = 0; j < nbLocations; j++) {
         if (cplex.getValue(opened[j]) >= 1 - tolerance) {
            cout << "Facility " << j << " is opened, it serves clients";
            for (IloInt c = 0; c < nbClients; ++c) {
               if (cplex.getValue(supply[c][j]) >= 1 - tolerance)
                  cout << " " << c;
            }
            cout << endl;
         }
      }
   }
   catch(IloException& e) {
      cerr << "Concert exception caught" << endl;
      throw;
   }
   catch(...) {
      cerr << "Unknown exception caught" << endl;
      throw;
   }
   env.end();
   return 0;
}
```

```
// -------------------------------------------------------------- -*- C++ -*-
// File: iloadmipex9.cpp
// Version 12.8.0
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2017. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------
//
// iloadmipex9.cpp -  Inject heuristic solutions from the generic callback
//                    for optimizing an all binary MIP problem
//
// To run this example, command line arguments are required.
// i.e.,   iloadmipex9   filename
//
// Example:
//     iloadmipex9  example.mps
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
#include <math.h>
#include <map>
#include <algorithm>


static void usage (const char *progname);

// This is the class implementing the heuristic callback.
class HeuristicCallback: public IloCplex::Callback::Function {
   public:
      // Constructor with data
      HeuristicCallback(IloCplex cplex, IloNumVarArray _vars) :
         vars(_vars),
         obj(cplex.getEnv(), vars.getSize())
      {
         // Generate the objective as a double array for easy look up
         IloObjective objexpr = cplex.getObjective();
         std::map<IloInt, double> objmap;

         for (IloExpr::LinearIterator it =
IloExpr(objexpr.getExpr()).getLinearIterator(); it.ok(); ++it) {
            objmap[it.getVar().getId()] = it.getCoef();
         }

         // Fill into a double array
         IloInt cols = vars.getSize();
         for (IloInt j = 0; j  <  cols; ++j) {
            std::map<IloInt, double>::iterator it =
objmap.find(vars[j].getId());
            if ( it != objmap.end() ) {
               obj[j] = it->second;
            }
         }
      }

      void Rounddown (const IloCplex::Callback::Context& context) {
         IloNumArray x(context.getEnv());
         try {
            context.getRelaxationPoint(vars, x);

            // Heuristic motivated by knapsack constrained problems.
            // Rounding down all fractional values will give an integer
            // solution that is feasible, since all constraints are <=
            // with positive coefficients

            double relobj = context.getRelaxationObjective();

            IloInt cols = vars.getSize();
            for (IloInt j = 0; j < cols; j++) {
               // Set the fractional variable to zero
               // Note that we assume all-binary variables. If there are
               // non-binary variables then the update must of course be
               // different.
               if ( x[j] ) {
                  // Set the fractional variables to zero
                  double integral;
                  double frac = modf(x[j], &integral);
                  frac        = (std::min) (1.0-frac,frac);

                  if ( frac > 1.0e-6 ) {
                     relobj -= obj[j]*x[j];
                     x[j]    = 0.0;
                  }
               }
            }

            // Post the rounded solution
            context.postHeuristicSolution(vars, x, relobj,
IloCplex::Callback::Context::SolutionStrategy::CheckFeasible);

            x.end();
         }
         catch (...) {
            x.end();
            throw;
         }
      }

      // This is the function that we have to implement and that CPLEX will call
      // during the solution process at the places that we asked for.
      virtual void invoke (const IloCplex::Callback::Context& context);
   private:
      // Variables
      IloNumVarArray vars;

      // Objective
      IloNumArray obj;
};
```

```
// Implementation of the invokeCallback function
void
HeuristicCallback::invoke (const IloCplex::Callback::Context &context)
{
   if ( context.inRelaxation() ) {
      // Call rounding heuristic
      Rounddown (context);
   }
}

int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
      }

      IloObjective   obj;
      IloNumVarArray vars(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, vars, rng);

      cplex.extract(model);

      // Now we get to setting up the callback.
      // We instanciate a HeuristicCallback and set the wherefrom parameter.
      HeuristicCallback heurCallback(cplex, vars);
      CPXLONG wherefrom = 0;

      wherefrom |= IloCplex::Callback::Context::Id::Relaxation;

      // We add the callback.
      cplex.use(&heurCallback, wherefrom);

      // Switch off regular heuristics to give the callback a chance
      cplex.setParam(IloCplex::Param::MIP::Strategy::HeuristicFreq, -1);

      if ( !cplex.solve() ) {
         cerr << "No solution found! Status = " << cplex.getStatus() << endl;
         throw(-1);
      }

      IloNumArray vals(env);
      cplex.getValues(vals, vars);
      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;
      env.out() << "Values          = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
      env.end();
      throw;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
      env.end();
      throw;
   }

   env.end();
   return 0;
}  // END main


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
} // END usage
```

```
// -------------------------------------------------------------- -*- C++ -*-
// File: ilobendersatsp2.cpp
// Version 12.8.0
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2017. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------
//
//
// Example ilobendersatsp2.cpp solves a flow MILP model for an
// Asymmetric Traveling Salesman Problem (ATSP) instance
// through Benders decomposition.
//
// The arc costs of an ATSP instance are read from an input file.
// The flow MILP model is decomposed into a master ILP and a worker LP.
//
// The master ILP is then solved by adding Benders' cuts via the generic
// callback function benders_callback during the branch-and-cut process.
//
// The callback benders_callback adds to the master ILP violated Benders'
// cuts that are found by solving the worker LP.
//
// The example allows the user to decide if Benders' cuts have to be separated
// just as lazy constraints or also as user cuts. In particular:
//
// a) Only to separate integer infeasible solutions.
// In this case, benders_callback is called with
// contextid=CPX_CALLBACKCONTEXT_CANDIDATE. The current candidate integer
// solution can be queried with CPXXcallbackgetcandidatepoint, and it can be
rejected
// by the user, optionally providing a list of lazy constraints, with the
// function CPXXcallbackrejectcandidate.
//
// b) Also to separate fractional infeasible solutions.
// In this case, benders_callback is called with
// contextid=CPX_CALLBACKCONTEXT_RELAXATION. The current fractional solution
// can be queried with CPXXcallbackgetrelaxationpoint. Cutting planes can then
// be added via CPXXcallbackaddusercuts.
//
// The example shows how to properly support deterministic parallel search
// with a user callback (there a significant departure here frome the legacy
// control callbacks):
//
// a) To avoid race conditions (as the callback is called simultaneously by
// multiple threads), each thread has its own working copy of the data
// structures needed to separate cutting planes. Access to global data
// is read-only.
//
// b) Thread-local data for all threads is created on THREAD_UP
// and destroyed on THREAD_DOWN. This guarantees determinism.
//
// To run this example, command line arguments are required:
//     ilobendersatsp.cpp {0|1} [filename]
// where
//     0         Indicates that Benders' cuts are only used as lazy constraints,
//               to separate integer infeasible solutions.
//     1         Indicates that Benders' cuts are also used as user cuts,
//               to separate fractional infeasible solutions.
//
//     filename  Is the name of the file containing the ATSP instance (arc
costs).
//               If filename is not specified, the instance
//               ../../../examples/data/atsp.dat is read
//
//
// ATSP instance defined on a directed graph G = (V, A)
// - V = {0, ..., n-1}, V0 = V \ {0}
// - A = {(i,j) : i in V, j in V, i != j }
// - forall i in V: delta+(i) = {(i,j) in A : j in V}
// - forall i in V: delta-(i) = {(j,i) in A : j in V}
// - c(i,j) = traveling cost associated with (i,j) in A
//
// Flow MILP model
//
// Modeling variables:
// forall (i,j) in A:
//    x(i,j) = 1, if arc (i,j) is selected
//           = 0, otherwise
// forall k in V0, forall (i,j) in A:
//    y(k,i,j) = flow of the commodity k through arc (i,j)
//
// Objective:
// minimize sum((i,j) in A) c(i,j) * x(i,j)
//
// Degree constraints:
// forall i in V: sum((i,j) in delta+(i)) x(i,j) = 1
// forall i in V: sum((j,i) in delta-(i)) x(j,i) = 1
//
// Binary constraints on arc variables:
// forall (i,j) in A: x(i,j) in {0, 1}
//
// Flow constraints:
// forall k in V0, forall i in V:
//    sum((i,j) in delta+(i)) y(k,i,j) - sum((j,i) in delta-(i)) y(k,j,i) =
q(k,i)
//    where q(k,i) =  1, if i = 0
//                 = -1, if k == i
//                 =  0, otherwise
//
// Capacity constraints:
// forall k in V0, for all (i,j) in A: y(k,i,j) <= x(i,j)
//
// Nonnegativity of flow variables:
// forall k in V0, for all (i,j) in A: y(k,i,j) >= 0
//
```

```
#include <ilcplex/ilocplex.h>
#include <string>
#include <vector>

ILOSTLBEGIN

typedef IloArray<IloIntVarArray> Arcs;

// Declarations for functions in this program

void createMasterILP(IloModel mod, Arcs x, IloNumArray2 arcCost);

void usage(char *progname);

/** The BendersATSP thread-local class. */

class Worker
{
public:
   // The constructor sets up the IloCplex algorithm to solve the worker LP, and
   // creates the worker LP (i.e., the dual of flow constraints and
   // capacity constraints of the flow MILP)
   //
   // Modeling variables:
   // forall k in V0, i in V:
   //    u(k,i) = dual variable associated with flow constraint (k,i)
   //
   // forall k in V0, forall (i,j) in A:
   //    v(k,i,j) = dual variable associated with capacity constraint (k,i,j)
   //
   // Objective:
   // minimize sum(k in V0) sum((i,j) in A) x(i,j) * v(k,i,j)
   //          - sum(k in V0) u(k,0) + sum(k in V0) u(k,k)
   //
   // Constraints:
   // forall k in V0, forall (i,j) in A: u(k,i) - u(k,j) <= v(k,i,j)
   //
   // Nonnegativity on variables v(k,i,j)
   // forall k in V0, forall (i,j) in A: v(k,i,j) >= 0
   Worker(IloInt _numNodes)
      : numNodes(_numNodes),
      numArcs(numNodes * numNodes),
      vNumVars((numNodes-1) * numArcs),
      uNumVars((numNodes-1) * numNodes),
      cplex(env),
      v(env, vNumVars, 0, IloInfinity),
      vIndex(env, vNumVars),
      u(env, uNumVars, -IloInfinity, IloInfinity),
      uIndex(env, uNumVars),
      obj(env)
   {
      IloInt i, j, k;
      IloModel mod(env, "atsp_worker");

      // Set up IloCplex algorithm to solve the worker LP
      cplex.extract(mod);
      cplex.setOut(env.getNullStream());

      // Turn off the presolve reductions and set the CPLEX optimizer
      // to solve the worker LP with primal simplex method.
      cplex.setParam(IloCplex::Param::Preprocessing::Reduce, 0);
      cplex.setParam(IloCplex::Param::RootAlgorithm, IloCplex::Primal);

      // Create variables v(k,i,j) forall k in V0, (i,j) in A
      // For simplicity, also dummy variables v(k,i,i) are created.
      // Those variables are fixed to 0 and do not partecipate to
      // the constraints.
      for (k = 1; k < numNodes; ++k) {
         for (i = 0; i < numNodes; ++i) {
            v[(k-1)*numArcs + i *numNodes + i].setBounds(0, 0);
         }
      }
      mod.add(v);

      // Set names for variables v(k,i,j)
      for (k = 1; k < numNodes; ++k) {
         for(i = 0; i < numNodes; ++i) {
            for(j = 0; j < numNodes; ++j) {
               char varName[100];
               sprintf(varName, "v.%d.%d.%d", (int) k, (int) i, (int) j);
               v[(k-1)*numArcs + i*numNodes + j].setName(varName);
            }
         }
      }

      // Associate indices to variables v(k,i,j)
      for (j = 0; j < vNumVars; ++j)
      {
         vIndex[j] = j;
         v[j].setObject(&vIndex[j]);
      }

      // Create variables u(k,i) forall k in V0, i in V
      mod.add(u);

      // Set names for variables u(k,i)
      for (k = 1; k < numNodes; ++k) {
         for(i = 0; i < numNodes; ++i) {
            char varName[100];
            sprintf(varName, "u.%d.%d", (int) k, (int) i);
            u[(k-1)*numNodes + i].setName(varName);
         }
      }

      // Associate indices to variables u(k,i)
      for (j = 0; j < uNumVars; ++j)
      {
         uIndex[j] = vNumVars + j;
         u[j].setObject(&uIndex[j]);
      }

      // Initial objective function is empty
      obj.setSense(IloObjective::Minimize);
      mod.add(obj);
```

```
      // Add constraints:
      // forall k in V0, forall (i,j) in A: u(k,i) - u(k,j) <= v(k,i,j)
      for (k = 1; k < numNodes; ++k) {
         for(i = 0; i < numNodes; ++i) {
            for(j = 0; j < numNodes; ++j) {
               if ( i != j ) {
                  IloExpr expr(env);
                  expr -= v[(k-1)*numArcs + i*(numNodes) + j];
                  expr += u[(k-1)*numNodes + i];
                  expr -= u[(k-1)*numNodes + j];
                  mod.add(expr <= 0);
                  expr.end();
               }
            }
         }
      }
   }
   ~Worker()
   {
      // Free all memory associated to env
      env.end();
   }

   // This routine separates Benders' cuts violated by the current x solution.
   // Violated cuts are found by solving the worker LP
   IloBool separate(const Arcs& x, const IloNumArray2& xSol, IloExpr& cutLhs,
IloNum& cutRhs)
   {
      IloBool violatedCutFound = IloFalse;
      IloModel mod = cplex.getModel();

      // Update the objective function in the worker LP:
      // minimize sum(k in V0) sum((i,j) in A) x(i,j) * v(k,i,j)
      //          - sum(k in V0) u(k,0) + sum(k in V0) u(k,k)
      mod.remove(obj);
      IloExpr objExpr = obj.getExpr();
      objExpr.clear();
      for (IloInt k = 1; k < numNodes; ++k) {
         for (IloInt i = 0; i < numNodes; ++i) {
            for (IloInt j = 0; j < numNodes; ++j) {
               objExpr +=  xSol[i][j] * v[(k-1)*numArcs + i*numNodes + j];
            }
         }
      }
      for (IloInt k = 1; k < numNodes; ++k) {
         objExpr += u[(k-1)*numNodes + k];
         objExpr -= u[(k-1)*numNodes];
      }
      obj.setExpr(objExpr);
      mod.add(obj);
      objExpr.end();

      // Solve the worker LP
      cplex.solve();

      // A violated cut is available iff the solution status is Unbounded
      if ( cplex.getStatus() == IloAlgorithm::Unbounded ) {

         IloNumVarArray var(env);
         IloNumArray val(env);

         // Get the violated cut as an unbounded ray of the worker LP
         cplex.getRay(val, var);

         // Compute the cut from the unbounded ray. The cut is:
         // sum((i,j) in A) (sum(k in V0) v(k,i,j)) * x(i,j) >=
         // sum(k in V0) u(k,0) - u(k,k)
         cutLhs.clear();
         cutRhs = 0.0;

         for (IloInt h = 0; h < val.getSize(); ++h) {
            IloInt *index_p = (IloInt*) var[h].getObject();
            IloInt index = *index_p;

            if ( index >= vNumVars ) {
               index -= vNumVars;
               IloInt k = index / numNodes + 1;
               IloInt i = index - (k-1)*numNodes;
               if ( i == 0 )
                  cutRhs += val[h];
               else if ( i == k )
                  cutRhs -= val[h];
            }
            else {
               IloInt k = index / numArcs + 1;
               IloInt i = (index - (k-1)*numArcs) / numNodes;
               IloInt j = index - (k-1)*numArcs - i*numNodes;
               cutLhs += val[h] * x[i][j];
            }
         }
         var.end();
         val.end();

         violatedCutFound = IloTrue;
      }

      return violatedCutFound;
   } // END separate
private:
   IloInt numNodes;
   IloInt numArcs;
   IloInt vNumVars;
   IloInt uNumVars;
   IloEnv env;
   IloCplex cplex;
   IloNumVarArray v;
   IloIntArray vIndex;
   IloNumVarArray u;
   IloIntArray uIndex;
   IloObjective obj;
};
```

```
class BendersATSPCallback : public IloCplex::Callback::Function
{
public:
   BendersATSPCallback(Arcs _x, IloInt numWorkers=1)
      : x(_x), workers(numWorkers, 0) {}

   ~BendersATSPCallback()
   {
      IloInt numWorkers = workers.size();
      for (IloInt w = 0; w < numWorkers; w++) {
         delete workers[w];
      }
      workers.clear();
   }

   void invoke(const IloCplex::Callback::Context& context)
   {
      int const threadNo =
context.getIntInfo(IloCplex::Callback::Context::Info::ThreadId);
      IloInt numNodes = x.getSize();

      // setup
      if (context.inThreadUp()) {
         delete workers[threadNo];
         workers[threadNo] = new Worker(numNodes);
         return;
      }

      // teardown
      if (context.inThreadDown()) {
         delete workers[threadNo];
         workers[threadNo] = 0;
         return;
      }

      IloEnv env = context.getEnv();
      IloNumArray2 xSol(env, numNodes);

      // Get the current x solution
      switch (context.getId()) {
      case IloCplex::Callback::Context::Id::Candidate:
         if ( !context.isCandidatePoint() ) // The model is always bounded
            throw IloCplex::Exception(-1, "Unbounded solution");
         for (IloInt i = 0; i < numNodes; ++i) {
            xSol[i] = IloNumArray(env);
            context.getCandidatePoint(x[i], xSol[i]);
         }
         break;
      case IloCplex::Callback::Context::Id::Relaxation:
         for (IloInt i = 0; i < numNodes; ++i) {
            xSol[i] = IloNumArray(env);
            context.getRelaxationPoint(x[i], xSol[i]);
         }
         break;
      default:
         // Free memory
         for (IloInt i = 0; i < numNodes; ++i) xSol[i].end();
         xSol.end();
         throw IloCplex::Exception(-1, "Unexpected contextID");
      }

      // Get the right worker
      Worker* worker = workers[threadNo];

      // Separate cut
      IloExpr cutLhs(env);
      IloNum cutRhs;
      IloBool sepStat = worker->separate(x, xSol, cutLhs, cutRhs);

      // Free memory
      for (IloInt i = 0; i < numNodes; ++i) xSol[i].end();
      xSol.end();

      if (sepStat) {
         // Add the cut
         IloRange r(env, cutRhs, cutLhs, IloInfinity);

         switch (context.getId()) {
            case IloCplex::Callback::Context::Id::Candidate:
               context.rejectCandidate(r);
               break;
            case IloCplex::Callback::Context::Id::Relaxation:
               context.addUserCut(r,
                                  IloCplex::UseCutPurge,
                                  IloFalse);
               break;
            default:
               r.end();
               throw IloCplex::Exception(-1, "Unexpected contextID");
         }
         r.end();
      }
   }
private:
   Arcs x;
   std::vector<Worker*> workers;
};



int
main(int argc, char **argv)
{
   IloEnv masterEnv;

   try {
      const char* fileName = "../../../examples/data/atsp.dat";

       // Check the command line arguments
      if ( argc != 2 && argc != 3) {
         usage (argv[0]);
         throw (-1);
      }
```

```
      if ( (argv[1][0] != '1' && argv[1][0] != '0') ||
           argv[1][1] != '\0' ) {
         usage (argv[0]);
         throw (-1);
      }

      IloBool separateFracSols = ( argv[1][0] == '0' ? IloFalse : IloTrue );

      masterEnv.out() << "Benders' cuts separated to cut off: ";
      if ( separateFracSols ) {
         masterEnv.out() << "Integer and fractional infeasible solutions." <<
endl;
      }
      else {
         masterEnv.out() << "Only integer infeasible solutions." << endl;
      }

      if ( argc == 3 )  fileName = argv[2];

      // Read arc_costs from data file (9 city problem)
      IloNumArray2 arcCost(masterEnv);
      ifstream data(fileName);
      if ( !data ) throw(-1);
      data >> arcCost;
      data.close();

      // create master ILP
      IloModel masterMod(masterEnv, "atsp_master");
      IloInt numNodes = arcCost.getSize();
      Arcs x(masterEnv, numNodes);
      createMasterILP(masterMod, x, arcCost);
      IloCplex masterCplex(masterMod);

      int numThreads = masterCplex.getNumCores();

      // Set up the callback to be used for separating Benders' cuts
      BendersATSPCallback cb(x, numThreads);
      CPXLONG contextmask = IloCplex::Callback::Context::Id::Candidate
                          | IloCplex::Callback::Context::Id::ThreadUp
                          | IloCplex::Callback::Context::Id::ThreadDown;
      if ( separateFracSols )
         contextmask |= IloCplex::Callback::Context::Id::Relaxation;
      masterCplex.use(&cb, contextmask);

      // Solve the model and write out the solution
      if ( masterCplex.solve() ) {

         IloAlgorithm::Status solStatus= masterCplex.getStatus();
         masterEnv.out() << endl << "Solution status: " << solStatus << endl;

         masterEnv.out() << "Objective value: "
                         << masterCplex.getObjValue() << endl;

         if ( solStatus == IloAlgorithm::Optimal ) {

            // Write out the optimal tour
            IloInt i, j;
            IloNumArray2 sol(masterEnv, numNodes);
            IloIntArray succ(masterEnv, numNodes);
            for (j = 0; j < numNodes; ++j)
               succ[j] = -1;

            for (i = 0; i < numNodes; i++) {
               sol[i] = IloNumArray(masterEnv);
               masterCplex.getValues(sol[i], x[i]);
               for (j = 0; j < numNodes; j++) {
                  if ( sol[i][j] > 1e-03 ) succ[i] = j;
               }
            }

            masterEnv.out() << "Optimal tour:" << endl;
            i = 0;
            while ( succ[i] != 0 ) {
               masterEnv.out() << i << ", ";
               i = succ[i];
            }
            masterEnv.out() << i << endl;
         }
         else {
            masterEnv.out() << "Solution status is not Optimal" << endl;
         }
      }
      else {
         masterEnv.out() << "No solution available" << endl;
      }

   }
   catch (const IloException& e) {
      cerr << "Exception caught: " << e << endl;
      masterEnv.end();
      throw;
   }
   catch (...) {
      cerr << "Unknown exception caught!" << endl;
      masterEnv.end();
      throw;
   }

   // Close the environments
   masterEnv.end();

   return 0;
} // END main


// This routine creates the master ILP (arc variables x and degree constraints).
//
// Modeling variables:
// forall (i,j) in A:
//    x(i,j) = 1, if arc (i,j) is selected
//           = 0, otherwise
//
// Objective:
// minimize sum((i,j) in A) c(i,j) * x(i,j)
```

```
//
// Degree constraints:
// forall i in V: sum((i,j) in delta+(i)) x(i,j) = 1
// forall i in V: sum((j,i) in delta-(i)) x(j,i) = 1
//
// Binary constraints on arc variables:
// forall (i,j) in A: x(i,j) in {0, 1}
//
void
createMasterILP(IloModel mod, Arcs x, IloNumArray2 arcCost)
{
   IloInt i, j;
   IloEnv env = mod.getEnv();
   IloInt numNodes = x.getSize();

   // Create variables x(i,j) for (i,j) in A
   // For simplicity, also dummy variables x(i,i) are created.
   // Those variables are fixed to 0 and do not partecipate to
   // the constraints.

   char varName[100];
   for (i = 0; i < numNodes; ++i) {
      x[i] = IloIntVarArray(env, numNodes, 0, 1);
      x[i][i].setBounds(0, 0);
      for (j = 0; j < numNodes; ++j) {
         sprintf(varName, "x.%d.%d", (int) i, (int) j);
         x[i][j].setName(varName);
      }
      mod.add(x[i]);
   }

   // Create objective function: minimize sum((i,j) in A ) c(i,j) * x(i,j)

   IloExpr obj(env);
   for (i = 0; i < numNodes; ++i) {
      arcCost[i][i] = 0;
      obj += IloScalProd(x[i], arcCost[i]);
   }
   mod.add(IloMinimize(env, obj));
   obj.end();

   // Add the out degree constraints.
   // forall i in V: sum((i,j) in delta+(i)) x(i,j) = 1

   for (i = 0; i < numNodes; ++i) {
      IloExpr expr(env);
      for (j = 0;   j < i; ++j)  expr += x[i][j];
      for (j = i+1; j < numNodes; ++j)  expr += x[i][j];
      mod.add(expr == 1);
      expr.end();
   }

   // Add the in degree constraints.
   // forall i in V: sum((j,i) in delta-(i)) x(j,i) = 1

   for (i = 0; i < numNodes; i++) {
      IloExpr expr(env);
      for (j = 0;   j < i; j++)  expr += x[j][i];
      for (j = i+1; j < numNodes; j++)  expr += x[j][i];
      mod.add(expr == 1);
      expr.end();
   }

}// END createMasterILP


void usage (char *progname)
{
   cerr << "Usage:     " << progname << " {0|1} [filename]"                <<
endl;
   cerr << " 0:        Benders' cuts only used as lazy constraints,"       <<
endl;
   cerr << "           to separate integer infeasible solutions."          <<
endl;
   cerr << " 1:        Benders' cuts also used as user cuts,"              <<
endl;
   cerr << "           to separate fractional infeasible solutions."       <<
endl;
   cerr << " filename: ATSP instance file name."                           <<
endl;
   cerr << "           File ../../../examples/data/atsp.dat "
        << "used if no name is provided."                                  <<
endl;

} // END usage
```