Proposal on addition of simplex-specific functions to OSI

This document is a result of discussion by
J. Eckstein, L. Ladanyi, J. Lee,
R. Lougee-Heimer and M. Nediak

May 3, 2001

1 Introduction

We propose to extend OsiSolverInterface to OsiSimplexSolverInterface class to provide access to standard operations on the basis and facilitate control over individual pivots. This should facilitate an implementation of pivot-based IP heuristics. There is also a number of desirable problem and solution query routines that do not seem to be implemented in OsiSolverInterface. Finally, we would like to have a number of additional dot-product methods for OsiPacked-Matrix.

2 Access to the basis

It is the fact that most solvers do some kind of preprocessing of the problem. This may result in elimination of some of the rows and columns, and, maybe, change in the remaining elements of the constraint matrix. Thus the purpose of the following methods is to provide a limited access to the current problem representation only. If the original indices of the structural variables are kept we do not need a special version of the methods isContinuous(int colIndex), isBinary(int colIndex), etc. in the OsiSimplexSolverInterface. Otherwise all these will have to be overridden.

We will assume below that slacks are index together with the primal vector and are assigned indices from getNumCols()+1 to getNumCols()+ getNumRows(). With this assumption we can require isContinuous() to return true when it is given the index of slack variable.

Before we proceed with a description of OsiSimplexSolverInterface methods it is also worth mentioning here that the OsiSolverInterface class needs a method analogous to getColSolution():

```
virtual const double* getRowSlacks() const;
//Returns a pointer to an array of values of slack variables
```

2.1 Access to the basic and non-basic lists

The following two methods are abstract methods of the OsiSimplexSolverInterface class. They will allow implementation of the remaining basis access methods:

```
virtual bool basisIsAvailable() const;
//Returns true if the basis is available

virtual const int* getColStatus() const;
//Returns a status of the structural variables

virtual const int* getRowStatus() const;
//Returns a status of the slack variables
```

These functions return the pointer to arrays of integers containg status of the colunms/rows. Values in the array must belong to the set of appropriately defined constants (indicating that the variable is basic, nonbasic at the lower bound, nonbasic at the upper bound, etc.).

For the rest of the methods, an implementation via getColStatus(), getRowStatus(), getColSolution() and getRowSlacks() must be provided. It can be overridden, if necessary for efficiency, in the descendants of the OsiSimplexSolverInterface class:

```
virtual void getBasIndices(int* indices) const;
//returns basic variable indices (array indices must be of
//appropriate size)

virtual void getBasValues(double* values) const;
//returns basic variable values (array values must be of
//appropriate size)

virtual void getBasIndices(OsiVectorInt& indices) const;
//returns basic variable indices in indices (may be more convenient
//to implement than the version above)

virtual void getBasValues(OsiVectorDouble& values) const;
//returns basic variable values in values (may be more convenient
//to implement than the version above)

virtual void getBasics(OsiPackedVector& xB) const;
//get both in a packed vector

virtual void getNonBasAtLower(int* indices) const;
virtual void getNonBasAtLower(OsiVectorInt& indices) const;
//get a list of non-basic variables at their lower bounds
//(slacks included)

virtual void getNonBasAtUpper(int* indices) const;
virtual void getNonBasAtUpper(OsiVectorInt& indices) const;
//get a list of non-basic variables at their upper bounds
```

```
//(slacks included)
```

2.2 Access to the basic matrix

(they may use some internal buffers of the solver). All of the following methods throw an appropriate exception if operation is not successful.

```
virtual int bSolve(double* x, const double* y);
//Solve B*x=y. Components of x should be in the same order as indices
//returned by getBasIndices(). Both x and y are dense.

virtual int bSolve(double* x, const OsiPackedVectorBase& y);
//Solve B*x=y, same requirement for an order of components.
//The result is returned in a dense vector x.

virtual int bSolve(OsiPackedVector& x, const OsiPackedVectorBase& y);
//Solve B*x=y. Both arguments are sparse vectors.

virtual int btSolve(double* x, const double* y);
//Solve B^t*x=y. Components of y should be specified in the same
//order as indices returned by getBasIndices()

virtual int btSolve(double* x, const OsiPackedVectorBase& y);
//Solve B^t*x=y, checks if y has indices that are non-basic.

virtual int btSolve(OsiPackedVector& x, const OsiPackedVectorBase& y);
//Solve B^t*x=y, same requirement on indices in y.

virtual int bInvACol(double* x, int col);
virtual int bInvACol(OsiPackedVector& x, int col);
//Return a column col of B^(-1)*A in x (dense and sparse versions)

virtual int bInvARow(double* x, int row);
virtual int bInvARow(OsiPackedVector& x, int row);
//Return a row row of B^(-1)*A in x (dense and sparse versions)

virtual int bInvCol(double* x, int col);
virtual int bInvCol(OsiPackedVector& x, int col);
//Return a column col of B^(-1) in x (dense and sparse versions)

virtual int bInvRow(double* x, int row);
virtual int bInvRow(OsiPackedVector& x, int row);
//Return a row row of B^(-1) in x (dense and sparse versions)

virtual int primalPivotResult(int colIn, int& colOut, double& t,
OsiPackedVector& dx, OsiPackedVector& dz, double& obj);
//Query on the result of the primal pivot that would use entering
//column colIn.
//Leaving column is returned in colOut, the step size in t, direction
//of the change in the primal vector - in dx, direction of the change
```

```
//in the reduced costs - in dz, new objective function value - in obj.

virtual int dualPivotResult(int rowIn, int& rowOut, double& t,
OsiPackedVector& dx, OsiPackedVector& dz, double& obj);
//Query on the result of the dual pivot that would use entering
//row rowIn.
//Leaving row is returned in rowOut, the step size in t, direction
//of the change in the primal vector - in dx, direction of the change
//in the reduced costs - in dz, new objective function value - in obj.
```

3 Operations that perform an actual pivot

All of the following return 0 if successful and non-zero error code otherwise.

```
virtual int primalPivot(int colIn, int& colOut, double& t,
OsiPackedVector& dx, OsiPackedVector& dz, double& obj);
//Perform the primal pivot that would use entering column colIn.
//Leaving column is returned in colOut, the step size in t, direction
//of the change in the primal vector - in dx, direction of the change
//in the reduced costs - in dz, new objective function value - in obj.

virtual int dualPivot(int colOut, int& colIn, double& t,
OsiPackedVector& dx, OsiPackedVector& dz, double& obj);
//Perform the dual pivot that would a leaving column colOut.
//Entering column is returned in colIn, the step size in t, direction
//of the change in the primal vector - in dx, direction of the change
//in the reduced costs - in dz, new objective function value - in obj.

virtual int pivot(int colIn, int colOut, int statOut);
//Perform a pivot by substituting a colOut for colIn in the basis.
//The status of the leaving variable is given in statOut
//(>0 for a variable leaving at the upper bound, <=0 - at the
//lower bound).
```

4 Reduced costs access operations

```
virtual void getReducedGradient(OsiPackedVector& z) const;
//Query the reduced costs vector with respect to a current objective
//and a basis

virtual void getReducedGradient(OsiPackedVector& z, const double * c);
virtual void getReducedGradient(OsiPackedVector& z, const
OsiPackedVectorBase& c);
//Find the reduced costs vector with respect to a current basis
//and the cost (gradient) vector c.
//These methods should probably not be made const as they may use some
//internal buffers of the solver.
```

5 Additional methods for OsiPackedMatrix

In computing Matrix-vector products, it may be necessary to perform these operations only for a subset of columns (rows). While the use of OsiPackedVector as an argument of times() and transposeTimes() methods can partially solve this problem, we would like to be able to specify that only a subset of result components is needed. Thus, we suggest using an OsiPackedVector& as a return argument with its index set specifying the required components of the product:

```
void times(const OsiPackedVectorBase& x,
OsiPackedVector& y) const;
//Return A * x in y. Only those components already in y are computed.

void transposeTimes(const OsiPackedVectorBase& x,
OsiPackedVector& y) const;
//Return x * A in y. Only those components already in y are computed.
```