The OS Dip Solver:
A Generic Block-Angluar Decomposition Implementation

Horand Gassmann, Jun Ma, Kipp Martin

October 8, 2010

Abstract

In this document we describe how to use the Decomposition in Integer Programming (Dip) project with the Optimization Services (OS) project. In particular, we describe the OS Dip solver which is a generic block-angluar decomposition algorithm. The OS Dip solver allows users to provide their own algorithm to solve the subproblems. The code for this example is contained in the folder `ApplicationTemplates/osDip`.

1 Introduction and Motivation

We follow the notation of Galati and Ralphs. The integer program of interest is:

$$z_{IP} = \min\{c^\top x \mid A'x \geq b',\ A''x \geq b'',\ x \in \mathbb{Z}^n\} \tag{1}$$

The problem is divided into two constraint sets $A'x \geq b'$ which we refer to as the relaxed, coupling, or block constraints and the core constraints $A''x \geq b''$. We then define the following polyhedron based on the relaxed constraints.

$$\mathcal{P} = \text{conv}(\{x \in \mathbb{Z}^n \mid A'x \geq b'\}) \tag{2}$$

The LP relaxation of the origianl problem is:

$$z_{LP} = \min\{c^\top x \mid A'x \geq b',\ A''x \geq b'',\ x \in \mathbb{R}^n\} \tag{3}$$

Ideally, the constraints $A'x \geq b'$ should be selected so that solving Z_D is an easy *hard problem* and provides better bounds than Z_{LP}.

$$z_D = \min\{c^\top x \mid A'x \geq b',\ x \in \mathcal{P},\ x \in \mathbb{R}^n\} \tag{4}$$

A generic block-angular decomposition algorithm is now available. It is based on the Decomposition in Integer Programming (Dip) project jointly with the Optimization Services (OS) project. We call this the OS Dip solver. It has the following features:

1. All subproblems are solved via an oracle; either the default oracle contained in our distribution or one provided by the user.
2. The OS Dip Solver code is independent of the oracle used to optimize the subproblems.
3. Variables are assigned to blocks using an OS option file, the block definition and assignment of variables to these blocks has no effect on the OS Dip Solver code.
4. Different blocks can be assigned different solver oracles based on the option values given in the OSoL file.
5. There is a default oracle implemented (called OSDipBockCoinSolver) that currently uses Cbc.
6. Users can add their own oracles without altering the OS Dip Solver code. This is done via polymorphic factories. The user creates a separate file containing the oracle class. The user-provided Oracle class inherits from the generic OSDipBlockSolver class. The user need only: 1) add the object file name for the new oracle to the Makefile, and 2) add the necessary line to OSDipFactoryInitializer.h indicating that the new oracle is present.

In particular, the implementation of the OS Dip solver provides a virtual class `OSDipBlockSolver` with a pure virtual function `solve()`. The user is expected to provide a class that inherits from `OSDipBlockSolver` and implements the method `solve()`. The `solve()` method should optimize a linear objective function over $\mathcal{P}$. More details are provided in Section 3. The implementation is such that the user only has to provide a class with a solve method. The user does not have to edit or alter any of the OS Dip Solver code. By using polymorphic factories the actual solver details are hidden from the OS Solver. A default solver, `OSDipBlockCoinSolver`, is provided. This default solver takes no advantage of special structure and simply calls COIN-OR bf Cbc.

2 Building and Testing the OS-Dip Example

Currently, the Decomposition in Integer Programming (**Dip**) package is not a dependency of the Optimization Services (**OS**) package – **Dip** is not included in the **OS** Externals file. In order to run the OS Dip solver it is necessary to download both the **OS** and **Dip** projects. Download order is irrelevant. In the discussion that follows we assume that for both **OS** and **Dip** the user has successfully completed a `configure`, `make`, and `make install`. We also assume that the user is working with the trunk version of both **OS** and **Dip.**

The OS Dip solver C++ code is contained in `TemplateApplication/osDip`. The `configure` will create a `Makefile` in the `TemplateApplication/osDip` folder. The `Makefile` must be edited to reflect the location of the **Dip** project. The `Makefile` contains the line

```
DIPPATH = /Users/kmartin/coin/dip-trunk/vpath-debug/
```

This setting assumes that there is a **lib** directory:

```
/Users/kmartin/coin/dip-trunk/vpath-debug/lib
```

with the **Dip** library that results from `make install` and an `include` directory

```
/Users/kmartin/coin/dip-trunk/vpath/include
```

with the **Dip** header files generated by `make install`. The user should adjust

```
/Users/kmartin/coin/dip-trunk/vpath/
```

to a path containing the **Dip** `lib` and `include` directories. After building the executable by executing the `make` command run the `osdip` application using the command:

```
./osdip --param osdip.parm
```

This should produce the following output.

```
FINISH SOLVE
Status= 0 BestLB= 16.00000   BestUB= 16.00000   Nodes= 1
SetupCPU= 0.01 SolveCPU= 0.10 TotalCPU= 0.11 SetupReal= 0.08
SetupReal= 0.12 TotalReal= 0.16
Optimal Solution
------------------------
Quality = 16.00
0      1.00
1      1.00
12     1.00
13     1.00
14     1.00
15     1.00
17     1.00
```

If you see this output, life is good and things are working. If this doesn't work, I almost certainly did something stupid and forget to fix it. The file `osdip.parm` is a parameter file. The use of the parameter file is explained in Section 8.

3 The OS Dip Solver – Code Description and Key Classes

The OS Dip Solver uses **Dip** to implement a Dantzig-Wofe decomposition algorithm for block-angular integer programs. Here are some key classes.

OSDipBlockSolver: This is a virtual class with a pure virtual function:

```
void solve(double *cost, std::vector<IndexValuePair*> *solIndexValPair,
double *optVal)
```

OSDipBlockSolverFactory: This is also virtual class with a pure virtual function:

```
OSDipBlockSolver* create()
```

This class also has the static method

```
OSDipBlockSolver* createOSDipBlockSolver(const string &solverName)
```

and a map

```
std::map<std::string, OSDipBlockSolverFactory*> factories;
```

Factory: This class inherits from the class **OSDipBlockSolverFactory**. Every sover class that inherits from the **OSDipBlockSolver** class should have a **Factory** class memember and since this **Factory** class member inherits from the **OSDipBlockSolverFactory** class it should implement a `create()` method that creates an optject in the class inheriting from **OSDipBlockSolver**.

OSDipFactoryInitializer: This class initializes the static map

```
OSDipBlockSolverFactory::factories
```

in the **OSDipBlockSolverFactory** class.

OSDipApp: This class inherits from the **Dip** class **DecompApp**. In **OSDipApp** we implement methods for creating the core (coupling) constraints, i.e. the constraints $A''x \geq b''$. This is done by implementing the `createModels()` method. Regardless, of the problem none of the relaxed or block constraints in $A'x \geq b'$ are created. These are treated implicitly in the solver class that inherits from the class **OSDipBlockSolver.** This class also implemnts a method that defines the variables that appear only in the blocks (**createModelMasterOnlys2**), and a method for generating an initial master (the method **generateInitVars()**).

Since the constraints $A'x \geq b'$ are treated explicitly by the Dip solver the `solveRelaxed()` method must be implemented. In our implementation we have the **OSDipApp** class data member:

```
std::vector<OSDipBlockSolver* > m_osDipBlockSolver;
```

when the `solveRelaxed()` method is called for block `whichBlock` in turn we make the call

```
m_osDipBlockSolver[whichBlock]->solve(cost, &solIndexValPair, &varRedCost);
```

and the appropriated solver in class **OSDipBlockSolver** is called. Finally, the **OSDipApp** class also initiates the reading of the OS option and instance files. How these files are used is discussed in Section 7. Based on option input data this class also creates the appropriate solver object for each block, i.e. it populates the `m_osDipBlockSolver` vector.

OSDipInterface: This class is used as an interface between the **OSDipApp** class and classes in the **OS** library. This provides a number of get methods to provide information to **OSDipApp** such as the coefficients in the A'' matrix, objective function coefficients, number of blocks etc. The **OSDipInterface** class reads the input OSiL and OSoL files and creates in-memory data structures based on these files.

OSDipBlockCoinSolver: This class inherits from the **OSDipBlockSolver** class. It is meant to illustrate how to create a solver class. This class solves each block by calling **Cbc**. Use of this class provides a generic black angular decomposition algorithm.

There is also **OSDip_Main.cpp:** which has the `main()` routine and is the entry point for the executable. It first creates a new price-branch-and-cut decomposition algorithm and then an Alps solver for which the `solve()` method is called.

4 User Requirements

The **OSDipBlockCoinSolver** class provides a solve method for optimizing a linear objective function over $\mathcal{P}$ given a linear objective function. However, this takes no advantage of the special structure available in the blocks. Therefore, the user may wish to implement his or her own solver class. In this case the user is required to do the following:

1. implement a class that inherits from the **OSDipBlockSolver** class and implements the solve method,
2. implement a class **Factory** that inherits from the class **OSDipBlockSolverFactory** and implements the `create()` method,
3. edit the file **OSDipFactoryInitializer.h** and add a line:

```
OSDipBlockSolverFactory::factories["MyBlockSolver"] = new
MyBlockSolver::Factory;
```

4. alter the Makefile to include the new source code.

Important – Directory Structure: In order to keep things clean, there is a directory **solvers** in the **osDip** folder. We suggest using the **solvers** directory for all of the solvers that inherit from **OSDipBlockSolver**.

5 Simple Plant/Lockbox Location Example

The problem minimizing the sum of the cost of capital due to float and the cost of operating the lock boxes is the problem.
Parameters:

$m-$ number of customers to be assigned a lock box

$n-$ number of potential lock box sites

$c_{ij}-$ annual cost of capital associated with serving customer j from lock box i

f_i- annual fixed cost of operating a lock box at location i

Variables:

$x_{ij}-$ a binary variable which is equal to 1 if customer j is assigned to lock box i and 0 if not

y_i- a binary variable which is equal to 1 if the lock box at location i is opened and 0 if not

The integer linear program for the lock box location problem is

$$\min \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} + \sum_{i=1}^{n} f_i y_i \tag{5}$$

$$(LB) \qquad x_{ij} - y_i \leq 0, \qquad i = 1,\ldots,n,\ j = 1,\ldots,m \tag{6}$$

$$\text{s.t.} \qquad \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1,\ldots,m \tag{7}$$

$$x_{ij},\ y_i \in \{0,1\}, \quad i = 1,\ldots,n,\ j = 1,\ldots,m. \tag{8}$$

The objective (5) is to minimize the sum of the cost of capital plus the fixed cost of operating the lock boxes. Constraints (6) are forcing constraints and require that a lock box be open if a customer is served by that lock box. For now, we consider these the $A'x \geq b'$ constraints. The requirement that every customer be assigned a lock box is modeled by constraints (7). For now, we consider these the $A''x \geq b''$ constraints.

Location Example 1: A three plant, five customer model.

		CUSTOMER					
		1	2	3	4	5	FIXED COSTS
	1	2	3	4	5	7	2
PLANT	2	4	3	1	2	6	3
	3	5	4	2	1	3	3

Table 1: Data for a 3 plant, 5 customer problem

$$\begin{aligned}\min \quad & 2x_{11} + 3x_{12} + 4x_{13} + 5x_{14} + 7x_{15} + 2y_1 + \\ & 4x_{21} + 3x_{22} + x_{23} + 2x_{24} + 6x_{25} + 3y_2 + \\ & 5x_{31} + 4x_{32} + 2x_{33} + x_{34} + 3x_{35} + 3y_3\end{aligned}$$

$$
\begin{array}{ll}
x_{11} \leq y_1 \leq 1 & \\
x_{12} \leq y_1 \leq 1 & \\
x_{13} \leq y_1 \leq 1 & \\
x_{14} \leq y_1 \leq 1 & \\
x_{15} \leq y_1 \leq 1 & \\
x_{21} \leq y_2 \leq 1 & \\
x_{22} \leq y_2 \leq 1 & \\
x_{23} \leq y_2 \leq 1 & A'x \geq b' \text{ constraints} \\
x_{24} \leq y_2 \leq 1 & \\
x_{25} \leq y_2 \leq 1 & \\
x_{31} \leq y_3 \leq 1 & \\
x_{32} \leq y_3 \leq 1 & \\
x_{33} \leq y_3 \leq 1 & \\
x_{33} \leq y_3 \leq 1 & \\
x_{33} \leq y_3 \leq 1 & \\
x_{ij}, y_i \geq 0, \ i = 1, \ldots, n, \ j = 1, \ldots, m. &
\end{array}
$$

$$
\begin{array}{lll}
\text{s.t.} & x_{11} + x_{21} + x_{31} = 1 & \\
& x_{12} + x_{22} + x_{32} = 1 & \\
& x_{13} + x_{23} + x_{33} = 1 & A''x \geq b'' \text{ constraints} \\
& x_{14} + x_{24} + x_{34} = 1 & \\
& x_{15} + x_{25} + x_{35} = 1 &
\end{array}
$$

Location Example 2 (SPL2): A three plant, three customer model.

		CUSTOMER			
		1	2	3	FIXED COSTS
	1	2	1	1	1
PLANT	2	1	2	1	1
	3	1	1	2	1

Table 2: Data for a three plant, three customer problem

$$
\begin{array}{ll}
\min & 2x_{11} + x_{12} + x_{13} + y_1 + \\
& x_{21} + 2x_{22} + x_{23} + y_2 + \\
& x_{31} + x_{32} + 1x_{33} + +y_3
\end{array}
$$

$$
\begin{array}{ll}
x_{11} \le y_1 \le 1 & \\
x_{12} \le y_1 \le 1 & \\
x_{13} \le y_1 \le 1 & \\
x_{21} \le y_2 \le 1 & \\
x_{22} \le y_2 \le 1 & A'x \ge b' \text{ constraints} \\
x_{23} \le y_2 \le 1 & \\
x_{31} \le y_3 \le 1 & \\
x_{32} \le y_3 \le 1 & \\
x_{33} \le y_3 \le 1 & \\
x_{ij}, y_i \ge 0,\ i = 1, \ldots, n,\ j = 1, \ldots, m. &
\end{array}
$$

$$
\begin{array}{lll}
\text{s.t.} & x_{11} + x_{21} + x_{31} = 1 & \\
& x_{12} + x_{22} + x_{32} = 1 & A''x \ge b'' \text{ constraints} \\
& x_{13} + x_{23} + x_{33} = 1 &
\end{array}
$$

6 Generalized Assignment Problem Example

A problem that plays a prominent role in vehicle routing is the *generalized assignment problem.* The problem is to assign each of n tasks to m servers without exceeding the resource capacity of the servers.

Parameters:

$n-$ number of required tasks

$m-$ number of servers

$f_{ij}-$ cost of assigning task i to server j

b_j- units of resource available to server j

$a_{ij}-$ units of server j resource required to perform task i

Variables:

$x_{ij}-$ a binary variable which is equal to 1 if task i is assigned to server j and 0 if not

The integer linear program for the generalized assignment problem is

$$
(GAP) \qquad
\begin{array}{lll}
\min & \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{m} f_{ij}x_{ij} & (9) \\
\text{s.t.} & \displaystyle\sum_{j=1}^{m} x_{ij} = 1, \qquad i = 1, \ldots, n & (10) \\
& \displaystyle\sum_{i=1}^{n} a_{ij}x_{ij} \le b_j, \qquad j = 1, \ldots, m & (11) \\
& x_{ij} \in \{0, 1\},\ i = 1, \ldots, n,\ j = 1, \ldots, m. & (12)
\end{array}
$$

The objective function (9) is to minimize the total assignment cost. Constraint (10) requires that each task is assigned a server. These constraints correspond to the $A''x \ge b''$ constraints. The

requirement that the server capacity not be exceeded is given in (11). These corresspond to the $A'x \geq b'$ constraints that are used to define $\mathcal{P}$. The test problem used in the file `genAssign.osil` is:

$$\begin{aligned} \min \quad & 2x_{11} + 11x_{12} + 7x_{21} + 7x_{22} & & \\ & +20x_{31} + 2x_{32} + 5x_{41} + 5x_{42} & & \\ & x_{11} + x_{12} & = & 1 \\ & x_{21} + x_{22} & = & 1 \\ & x_{31} + x_{32} & = & 1 \\ & x_{41} + x_{42} & = & 1 \\ & 3x_{11} + 6x_{21} + 5x_{31} + 7x_{41} & \leq & 13 \\ & 2x_{12} + 4x_{22} + 10x_{32} + 4x_{42} & \leq & 10 \end{aligned}$$

7 Defining the Problem Instance and Blocks

Here we describe how to use the OSoption stuff and OSInstance. We illustrate with a simple plant location problem. Refer back to the example in Table 1 for a three-plant, five-customer problem. We treat the fixed charge constraints as the block constraints, i.e. we treat constraint set 6 as the set $A'x \geq b'$ constraints. These constraints naturally break into a block for each plant, i.e. there is a block of constraints:

$$x_{ij} \leq y_i \tag{13}$$

In order to use the OS Dip solver it is necessary to: 1) define the set of variables in each block and 2) define the set of constraints that constitute the core or coupling constraints. This information is communicated to the OS Dip solver using Optimization Services option Language (OSoL). The OSoL input file for the example in Table 1 appears in Figures 1 and 2. See lines 32-55. There is an `<other>` option with `name="variableBlockSet"` for each block. Each block then lists the variables in the block. For example, the first block consists of variable indexed by 0, 1, 2, 3, 4, and 15. These correspond to variables x_{11}, x_{12}, x_{13}, x_{13}, x_{14}, and y_1. Likewise the second block corresponds to the variable for the second plant and the third block corresponds to variables for the third plant.

It is also necessary to convey which constraints constitute the core constraints. This is done in lines 58-64. The core constraints are indexed by 15, 16, 17, 18, 19. These constitute the demand constraints given in Equation (7).

Notice also that in lines 32, 40, and 48 there is an attribute `value` in the `<other>` variable element with the attribute `name` equal to `variableBlockSet`. The attribute `value` should be the name of the solver factory that should be assigned to solve that block. For example, if the optimization problem that results from solving a linear objective over the constraints defining the first block is solved using `MySolver1` then this must correspond to a

```
OSDipBlockSolverFactory::factories["MySolver1"] = new
MySolver1::Factory;
```

in the file **OSDipFactoryInitializer.h**. In the test file, `spl1.osol` for the first block we set the solver to a specialized solver for the simple plant location problem (`OSDipBlockSplSolver`) and for the other two blocks we use the generic solver (`OSDipBlockCoinSolver`).

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <osol>
3      <general>
4         <instanceName>spl1 -- setup constraints are the blocks</instanceName>
5      </general>
6      <optimization>
7         <variables numberOfOtherVariableOptions="6">
8            <other name="initialCol" solver="Dip" numberOfVar="6" value="0">
9               <var idx="0" value="1"/>
10              <var idx="1" value="1"/>
11              <var idx="2" value="1"/>
12              <var idx="3" value="1"/>
13              <var idx="4" value="1"/>
14              <var idx="15" value="1"/>
15           </other>
16           <other name="initialCol" solver="Dip" numberOfVar="6" value="1">
17              <var idx="5" value="1"/>
18              <var idx="6" value="1"/>
19              <var idx="7" value="1"/>
20              <var idx="8" value="1"/>
21              <var idx="9" value="1"/>
22              <var idx="16" value="1"/>
23           </other>
24           <other name="initialCol" solver="Dip" numberOfVar="6" value="2">
25              <var idx="10" value="1"/>
26              <var idx="11" value="1"/>
27              <var idx="12" value="1"/>
28              <var idx="13" value="1"/>
29              <var idx="14" value="1"/>
30              <var idx="17" value="1"/>
31           </other>
32           <other name="variableBlockSet" solver="Dip" numberOfVar="6" value="MySolver1">
33              <var idx="0"/>
34              <var idx="1"/>
35              <var idx="2"/>
36              <var idx="3"/>
37              <var idx="4"/>
38              <var idx="15"/>
39           </other>
40           <other name="variableBlockSet" solver="Dip" numberOfVar="6" value="MySolver2">
41              <var idx="5"/>
42              <var idx="6"/>
43              <var idx="7"/>
44              <var idx="8"/>
45              <var idx="9"/>
46              <var idx="16"/>
47           </other>
```

Figure 1: A sample OSoL file – SPL1.osol

One can use the OSoL file to specify a set of starting columns for the initial restricted master. In Figure 1 see lines 8-31. In and OS option file (OSoL) there is `<variables>` element that has `<other>` children. Initial columns are specified using the `<other>` elements. This is done by using the `name` attribute and setting its value to `initialCol`. Then the children of the tag contain index-value pairs that specify the column. For example, the first initial column corresponds to setting:

$$x_{11} = 1, \quad x_{12} = 1, \quad x_{13} = 1, \quad x_{14} = 1, \quad x_{15} = 1, \quad y_1 = 1$$

Finally note that in all of this discussion we know to apply the options to **Dip** because the attribute `solver` always had value `Dip`. It is critical to set this attribute in all of the option tags.

8 The Dip Parameter File

The **Dip** solver has a utility class **UtilParameters**, for parsing a parameter file. The **UtilParameters** class constructor takes a parameter file as an argument. In the case of the OS Dip solver the name of the parameter file is **osdip.parm** and the parameter is read in at the command line with the command

```
./osdip -param osdip.parm
```

The **UtilParameters** class has a method **GetSetting()** for reading the paramater values. In the OS Dip implementation there is a class **OSDipParam** that has a data memmbers key paramaters such as the name of the input OSiL file and input OSoL file. The **OSDipParam** class has a method **getSettings()** that takes as an argument a pointer to an object in the **UtilParameters** and uses **GetSetting()** method to return the relevant paramater values. For example:

```
OSiLFile = utilParam.GetSetting("OSiLFile", "", common);
OSoLFile = utilParam.GetSetting("OSoLFile", "", common);
```

In the current **osdip.parm** file we have:

```
#first simple plant location problem
OSiLFile = spl1.osil
#setup constraints as blocks
OSoLFile = spl1.osol
#assignment constraints as blocks
#OSoLFile = spl1-b.osol

#second simple plant location problem
#OSiLFile = spl2.osil
#setup constraints as blocks
#OSoLFile = spl2.osol
#assignment constraints as blocks
#OSoLFile = spl2-b.osol

#third simple plant location problem -- block matrix data not used
```

```
#OSiLFile = spl3.osil
#setup constraints as blocks
#OSoLFile = spl3.osol

#generalized assignment problem
#OSiLFile = genAssign.osil
#OSoLFile = genAssign.osol

#Martin textbook example
#OSiLFile = smallIPBook.osil
#OSoLFile = smallIPBook.osol
```

By commenting and uncommenting you can run one of four problems that are in the **data** directory. The first example, **spl1.osil**, corresponds to the simple plant location model given in Table 1. Using the option file **spl1.osol** treats the setup forcing constraints 6 as the $A'x \geq b'$ constraints. Using the option file **spl1-b.osol** treats the demand constraints 7 as the $A'x \geq b'$ constraints. Likewise for the problem **spl2.osil** which correponds to the simple plant location data given in Table 2.

In both examples **spl1.osil** and **spl2.osil** the $A'x \geq b'$ constraints are explicitly represented in the OSiL file. However, this is not necessary. The solver Factory **OSDipBlockSlpSolver** is a special oracle that only needs the objective function coefficients and pegs variables based on the sign of the objective function coefficients. The **spl3.osil** is the example given in Table 1 but without the setup forcing constraints. Each block uses the **OSDipBlockSlpSolver** oracle.

The **genAssign.osil** file corresponds to the generalized assignment problem given in Section 6. The option file **genAssign.osol** treats the capacity constraints 11 as the $A'x \geq b'$ constraints.

The last problem defined in the file **smallIPBook.osil** is based on Example 16.3 on page 567 in *Large Scale Linear and Integer Optimization*. The option file treats the constraints

$$4x_1 + 9x_2 \leq 18, \quad -2x_1 + 4x_2 \leq 4$$

as the $A'x \geq b'$ constraints.

The user should also be aware of the parameter `solverFactory`. This parameter is the name of the default solver Factory. If a solver is not named for a block in the OSoL file this value is used. We have set the value of this string to be `OSDipBlockCoinSolver`.

9 Issues to Fix

- Enhance solveRelaxed to allow parallel processing of blocks. See ticket 30.
- Does not work when there are 0 integer variables. See ticket 31.
- Be able to set options in C++ code. See ticket 41. It would be nice to be able to read all the options from a generic options file. It seems like right now options for the **DecompAlgo** class cannot be set inside C++.
- Problem with Alps bounds at node 0. See ticket 43
- Figure out how to use BranchEnforceInMaster or BranchEnforceInSubProb so I don't get the large bonds on the variables. See ticket 47.

10 Miscellaneous Issues

If you want to terminate at the root node and just get the dual value under the `ALPS` option put:

```
[ALPS]
nodeLimit = 1
```

More from Matt:

```
Kipp - the example you sent finds the optimal solution after a
few passes of pricing and therefore never calls the cut generator.
By default, the PC solver, in the root node starts with pricing,
and does not stop until it prices out (or finds optimal, or within gap limits).

If it prices out and has not yet found optimal, then it will proceed to cuts.

This is parameter driven.


You'll see in the log file (LogDebugLevel = 3),
PRICE_AND_CUT  LimitRoundCutIters       2147483647
PRICE_AND_CUT  LimitRoundPriceIters     2147483647

This is the number of Price/Cut iterations to take before switching off (i.e., MAXINT).

To force it to cut before pricing out, change this parameter in the parm file.
For example, if you change to :

[DECOMP]
LimitRoundPriceIters = 1
LimitRoundCutIters   = 1

It will then go into your generateCuts after one pricing iteration.
```

If there is an integer solution at the root node, it may be the case that we are still not optimal. A perfect example is where you want to add tour-breaking constraints. There could be an integer solution, but you still violate a tour-breaking constraint. Here is what Matt says:

"By default, DIP assumes, that if problem is LP feasible to the linear system and IP feasible, then it is feasible. In the case where the user knows something that DIP does not (e.g., that the linear system does not define the entire valid constraint system, as in TSP), then they must provide a derivation of this function APPisUserFeasible. Then, DIP will check LP feasible, IP feasible and lastly, `APPisUserFeasible` before declaring a point a feasible solution."

For an example of using this see, `https://projects.coin-or.org/Dip/browser/trunk/Dip/examples/TSP/TSP_DecompApp.cpp`.

If you return false in `APPisUserFeasible` it will next call `solverRelaxed()` and then `generateCuts()` for you.

11 Setting Options

Kipp – put in information on how to set options for the `AlpsDecompModel`
Some options to get output

```
Turn up the log levels to see when that happens:
[DECOMP]
LogDebugLevel    = 3
LogLevel         = 3
LogDumpModel     = 2
LogLpLevel       = 1
```

```
Yes - it is also used to solve the master problem as an IP.
That is, it periodically changes the variables in the master
problem (lambda) to integers and solves the master as an
integer program. This is a heuristic to try and get incumbent
solutions for the entire problem.
This, in my thesis, is called Price-and-Branch.
```

```
48          <other name="variableBlockSet" solver="Dip" numberOfVar="6" value="MySolver3">
49             <var idx="10"/>
50             <var idx="11"/>
51             <var idx="12"/>
52             <var idx="13"/>
53             <var idx="14"/>
54             <var idx="17"/>
55          </other>
56       </variables>
57       <constraints numberOfOtherConstraintOptions="1">
58          <other name="constraintSet" solver="Dip" numberOfCon="5" type="Core">
59             <con idx="15"/>
60             <con idx="16"/>
61             <con idx="17"/>
62             <con idx="18"/>
63             <con idx="19"/>
64          </other>
65       </constraints>
66    </optimization>
67 </osol>
```

Figure 2: A sample OSoL file – SPL1.osol (Continued)