COCONUT Deliverable D14

Set of combination algorithms for state of the art modules

EPFL – *Swiss Federal Institue of Technology – Lausanne*
IRIN – *University of Nantes*
University of Vienna

July 30, 2003

Chapter 3

External Converters

3.1 ampl2dag

3.1.1 Copyright

This information is copyright by Dr. Oleg Shcherbina and Dr. Hermann Schichl from the Mathematics Department of the University of Vienna.

3.1.2 Module name

Module name is `ampl2dag`

3.1.3 Type of module

`ampl2dag` is an external converter.

3.1.4 Description of the module

`ampl2dag` is a module which produces a DAG (directed acyclic graph) from an AMPL file. The result is a text file `<filename>.dag`, containing the problem description as a directed acyclic graph (input format of the COCONUT environment).

3.1.5 Dependencies

1. ampl
2. nl2dag
3. info2dag

3.1.6 Limits

There are no hardcoded limits. Cannot handle: function `if, errorf`.

3.1.7 Input

Any AMPL file.

3.1.8 Output

A text file `<filename>.dag`, containing the problem description as a directed acyclic graph (input format of the COCONUT environment) and parameters like bounds on the variables, AMPL names of variables, constraints, fixed variables, constants added to objective values.

3.1.9 Control parameters

None.

3.1.10 Statistics

None.

3.1.11 Example

Let us consider the following problem:

$$\min x^3 - x \tag{3.1}$$
$$\text{s.t. } x \in [-2, 2]. \tag{3.2}$$

This problem written in the AMPL modeling language looks like:

```
var x <=2, >= -2;
minimize obj: x^3 - x;
```

To obtain the DAG file of the above problem we run the program `ampl2dag`. The DAG file will contain:

```
<N> V 1
<0> b [-2,2]: V 0
<1> +
<2> V 0
<E> 1 2 -1
<3> ^
<4> V 0
<E> 3 4 1
<5> C 3
<E> 3 5 1
<E> 1 3 1
```

```
<N> M 1 min
<N> N 0 'x'
<N> S 'st'
<N> O 'obj'
```

3.2 dag2c (C write)

3.2.1 Copyright

This information is copyright by Dr. Oleg Shcherbina and Dr. Hermann Schichl from the Mathematics Department of the University of Vienna.

3.2.2 Module name

Module name is `dag2c(C write)`

3.2.3 Type of module

`dag2c (C write)` is a conversion (report) module.

3.2.4 Description of the module

`dag2c` is a module which converts DAG files to `C` format. The result is one file: `<filename>.c` containing the description of the problem as `C` program.

3.2.5 Dependencies

1. coconut-environment-1.32, coconut-environment-2.11

3.2.6 Limits

There are no hardcoded limits. Cannot handle: function `log10, tan, atan, asin, acos, cosh, tan, if`

3.2.7 Service *print*

`dag2c (C write)` produces one file: `<filename>.c` containing the description of the problem as `C` program.

3.2.8 Kind of input

`<filename>.dag`, where `<filename>.dag` is the DAG file to be converted to `C` format.

3.2.9 Output

`<filename>.c`

3.2.10 Command line switches

-s do not pass the DAG through the DAG simplifier before producing `C` code.

-y print multipliers as y and k, respectively.

3.2.11 Statistics

None.

3.2.12 Example

Convert the DAG file from Subsection 3.1.11 to `C` format:

```
#include <math.h>

double objective(double *x)
{
  return -x[0] + pow(x[0], 3);
}

double constraint(int i, double *x)
{
  switch(i)
  {
    case 0: /* bound: [-2, 2], node: 0 */
      return x[0];
  }
}
```

3.3 dag2gams

3.3.1 Copyright

This information is copyright by Dr. Oleg Shcherbina and Dr. Hermann Schichl from the Mathematics Department of the University of Vienna.

3.3.2 Module name

Module name is `dag2gams`

3.3.3 Type of module

`dag2gams` is a conversion (report) module.

3.3.4 Description of the module

`dag2gams` is a module which converts DAG files to `GAMS` format. The result is one file: `<filename>.gms`. This file contains the problem description, bounds on the variables and the parameters needed by the solver.

3.3.5 Dependencies

1. coconut-environment-1.32, coconut-environment-2.11

3.3.6 Limits

There are no hardcoded limits. Cannot handle: function `log10, tan, atan, asin, acos, cosh, tan, if`.

3.3.7 Service *print*

`dag2gams` produces a text file `<filename>.dag`, containing the problem description in `GAMS` format.

3.3.8 Kind of input

`<filename>.dag` where `<filename>.dag` is the DAG file to be converted to GAMS format.

3.3.9 Output

`<filename>.gms` Converted GAMS file.

3.3.10 Command line switches

`-s` do not pass the DAG through the DAG simplifier before producing output.

3.3.11 Statistics

None.

3.3.12 Example

Convert the DAG file from Subsection 3.1.11 to `GAMS` format:

```
* GAMS-model ex.gms written by DAG-GAMS Converter at 21/07/2003 11:25:25
* University of Vienna
\$offdigit;
File res/ex.res/;
Put res;
*res - name of output file ex.res into which a solution is written
```

```
options limrow=0,
        limcol=0,
        solprint=off,
        sysout=on,
        nlp=baron,
        reslim=1000
*running time limit in seconds
        ;
 Set j/1*1/;
 Equations objcon
;
Variables x(j), obj;
* Objective function (to be minimized)
objcon.. obj =e= -x('1') +
power(x('1'), 3);


x.lo('1')=-2;
x.up('1')=2;

x.l(j)=1/(ORD(j)+1);
*nonzero initial guess
Model m/All/;
m.workspace = 32;

Solve m using nlp minimizing obj;

PUT "modelstatus = ", m.MODELSTAT; PUT /;
loop(j,PUT "x(", j.te(j), ") = ", x.l(j):16:10/);
PUT "obj = ", obj.l:16:10/;
PUT "infeas = ", m.NUMINFES; PUT /;
PUT "nonopt = ", m.NUMNOPT; PUT /;
display x.l, obj.l;
```

3.4 dag2lgo

3.4.1 Copyright

3.4.2 Module name

Module name is `dag2lgo`

3.4.3 Type of module

`dag2lgo` is a conversion (report) module.

3.4.4 Description of the module

`dag2lgo` is a module which converts DAG files to Windows version `LGO` solver format. The result consists of two files: `lgo.in` and `user_fct.c`, both are needed to run the problem with `LGO`. The first one is the configuration file, containing the dimension of the problem, bounds for the variables and default parameters (suggested by the author of the solver). The second one is the problem coded in `C` but within a special function needed by the `LGO` solver.

3.4.5 Dependencies

1. coconut-environment-1.32, coconut-environment-2.11

3.4.6 Limits

There are no hardcoded limits. Cannot handle: function `log10, tan, atan, asin, acos, cosh, tan, if`

3.4.7 Service *print*

`dag2lgo` produces two files: `lgo.in` and `user_fct.c`, both are needed to run the problem with `LGO`.

3.4.8 Kind of input

`<filename>.dag` where `<filename>.dag` is the DAG file to be converted to `LGO` format.

3.4.9 Output

`lgo.in` and `user_fct.c`.

3.4.10 Control parameters

`-s` do not pass the DAG through the DAG simplifier before producing output.

3.4.11 Statistics

None.

3.4.12 Example

Convert the DAG file from Subsection 3.1.11 to WinLGO format:

```
user_fct.c

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int __declspec(dllexport) _stdcall USER_FCT( double x[],
                                             double fox[1],
                                             double gox[])

{
/* Objective function (to be minimized) */

fox[0] = -x[0] +
pow(x[0], 3);


/* Constraints */
gox;

return 0;
}
```

The file lgo.in contains:

```
--- lgo.in input file, used with lgo.exe text I/O interface implementation ---
 Driver file lgo.in for LGO which is constructing from DAG for model ex
 written by DAG-LGO Converter at 21/07/2003 11:26:40
 University of Vienna


--- Model Descriptors ---
ex                      ModelName
1                    Number Of Variables
0                    Number of Constraints
Variable names       Lower Bounds        Nomimal Values      Upper Bounds
x1                    -2                      0                   2
Obj_Funct               Objective function name
! Constraint names      Constraint types: equations 0, <= inequalities -1
!--- SOLVER OPTIONS AND PARAMETERS ---
```

```
3                      ! Operational modes 0: LS; 1: BB+LS; 2: GAR+LS; 3: MS+LS
50000                  ! Maximal no. of fct evals in global search phase
20000                  ! Maximal no. of fct evals in global search phase w/o
                         improvement
1000                   ! Constraint penalty multiplier
-10                    ! Target objective function value in global search phase
-10                    ! Target objective function value in local search phase
0.000001               ! Merit function precision improvement threshold in
                         local search phase
0.000001               ! Constraint violation tolerance in local search phase
0.000001               ! Kuhn-Tucker local optimality conditions tolerance in
                         local search phase
0                      ! Built-in random number generator seed value
300                    ! Program execution time limit (seconds)
```

3.5 dag2globsol

3.5.1 Copyright

This information is copyright by Dr. Oleg Shcherbina and Dr. Hermann Schichl from the Mathematics Department of the University of Vienna.

3.5.2 Module name

Module name is `dag2globsol`

3.5.3 Type of module

`dag2globsol` is a conversion (report) module.

3.5.4 Description of the module

`dag2globsol` is a module which converts DAG files to `GlobSol` format. The result consists of two files: `<problem>.f90` and `<problem>.DT1`. The first file contains the description of the problem in `Fortran 90` with a special structure needed by the solver. The second one contains the initializing data: initial interval of the variables, stopping criterion and the information whether the boundaries of the initial interval are strict or not.

3.5.5 Dependencies

1. coconut-environment-1.32, coconut-environment-2.11

3.5.6 Limits

There are no hardcoded limits. Cannot handle: function `log10, tan, atan, asin, acos, cosh, tan, if`

3.5.7 Service *print*

`dag2globsol` produces two files:`<problem>.f90` and `<problem>.DT1`. The first file contains the description of the problem in `Fortran 90` with a special structure needed by the solver. The second one contains the initializing data: initial interval of the variables, stopping criterion and the information whether the boundaries of the initial interval are strict or not.

3.5.8 Kind of input

`<filename>.dag` where `<filename>.dag` is the DAG file to be converted to `GlobSol` format.

3.5.9 Output

`<problem>.f90` and `<problem>.DT1`.

3.5.10 Control parameters

`-s` do not pass the DAG through the DAG simplifier before producing output.

3.5.11 Statistics

None.

3.5.12 Example

Convert the DAG file from subsection 3.1.11 to `GlobSol` format:

```
PROGRAM ex
  USE CODELIST_CREATION

  PARAMETER (NN=1)
  PARAMETER (NSLACK = 0)
  TYPE(CDLVAR), DIMENSION (NN+NSLACK):: X
  TYPE(CDLLHS), DIMENSION(1):: PHI
  OUTPUT_FILE_NAME= 'ex.CDL'
  CALL INITIALIZE_CODELIST(X)

  PHI(1) = -X(1) + (X(1)**3)
```

```
  CALL FINISH_CODELIST
END PROGRAM ex
```

The file ex.DT1 contains:

```
1.0D-8                ! File: ex.DT1
-2.000000   2.000000
T    T
```

3.6 c2dag form

`c2dag` is a set of programs for the conversion of an optimization problem specified in a C++ program to an equivalent (unsimplified) DAG for use in the COCONUT environment.

3.6.1 C++ specification of optimization problems

Let us consider the global optimization problem

$$\begin{array}{ll} \min & (x_0 + x_1) \cdot x_2 \\ \text{s.t.} & x_0 + x_1 = 0 \\ & x_0 \in [0,1],\ x_1 \in [1,2],\ x_2 \in [2,3] \end{array} \tag{3.3}$$

We shall describe this problem in the C++ language as follows.

```
symbolic x0;
symbolic x1;
symbolic x2;
x0 = "[0,1]";
x1 = "[1,2]";
x2 = "[2,3]";
constraint(x0+x1,"[0,0]");
optimize((x0+x1)*x2);
```

We can see that each variable has to be of "symbolic" type. After declaration we have to give bounds for each variable `x`

```
x = "[x_min,x_max]";
```

where `x_min`, `x_max` are upper and lower bound of the variable `x`. If we would like to define infinity we have to use the symbol `I`. The constraints can be defined by using the procedure `constraints(expression,bounds)`. The first argument of this procedure is an arithmetical expression representing a particular constraint. The second argument is a string which gives us information about the upper and lower bound of the arithmetical expression, where

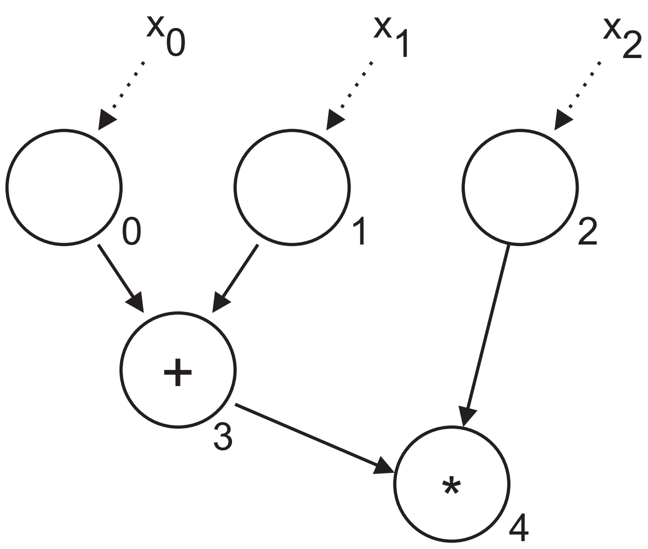
$$\texttt{lower\_bound} \leq \texttt{expression} \leq \texttt{upper\_bound}.$$

If we would like to define equality constraints we can use the following bounds `"[0,0]"`. In order to define inequality constraints, we use for example `"[0,I]"` for ≥ 0 or `"[-I,1]"` for ≤ 0. The command `optimize(expression)` means that `expression` should be minimized.

This program supports the following standard functions: sqr, sqrt, log, exp, sin, cos.

3.6.2 DAG description of optimization problems

Figure 3.1: Graph corresponding to $(x_0 + x_1) \cdot x_2$



Each arithmetical expression can be described using some DAG. For example, the expression $(x_0+x_1)\cdot x_2$ occurring in (7.1) is described by the DAG given in Fig. 1. Each graph contains the following elements:

- nodes (circles in Fig. 3.1),
- edges (arrows in Fig. 3.1),
- variables (x_0, x_1, x_2 in Fig. 3.1),
- labels or operator identifiers (*, + in Fig. 3.1).

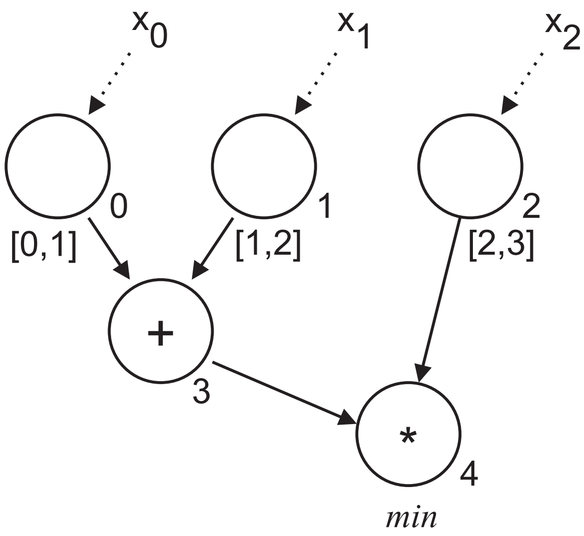
In order to describe a DAG in the COCONUT environment, a special input format is required. For example, the graph from Fig. 3.1 corresponds with the following line-oriented description (in a file `<name>.dag`):

```
<0> b [0,1]: V 0
<1> b [1,2]: V 1
<2> b [2,3]: V 2
<3> +
<E> 3 0
<E> 3 1
<4> *
<E> 4 3
<E> 4 2
```

DAGs also describe complete optimization problems. For example, the DAG for the problem (7.1) is shown in Fig. 3.2 and has the following line-oriented description:

```
<0> b [0,1]: V 0
<1> b [1,2]: V 1
<2> b [2,3]: V 2
<3> +
<E> 3 0
<E> 3 1
<4> *
<E> 4 3
<E> 4 2
<N> M 4 min
```

Figure 3.2: Graph of the global optimization problem (7.1)



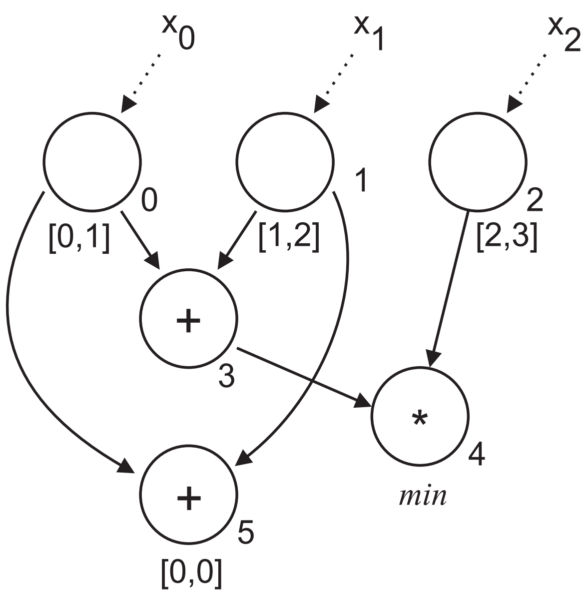
We can also define additional constraints. The DAG corresponding to the optimization problem

$$
\begin{array}{ll} \min & (x_0 + x_1) \cdot x_2 \\ \text{s.t.} & x_0 + x_1 = 0 \\ & x_0 \in [0,1],\ x_1 \in [1,2],\ x_2 \in [2,3] \end{array} \tag{3.4}
$$

is drawn in Fig. 3.3 and has the following line-oriented description:

```
<0> b [0,1]: V 0
<1> b [1,2]: V 1
<2> b [2,3]: V 2
<3> +
<E> 3 0
<E> 3 1
<4> *
<E> 4 3
<E> 4 2
<N> M 4 min
<5> b [0,0]: +
<E> 5 0
<E> 5 1
```

Figure 3.3: DAG of the problem (7.5)



3.7 c2dag manual

`c2dag` allows the conversion of any optimization problem specified as in Section 3.6.1 to an equivalent (unsimplified) DAG.

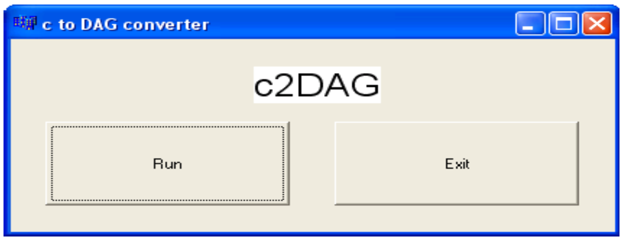
3.7.1 How to use the C to DAG converter with the g++ compiler (Linux OS)

1) Write the C code to the file “C_code.cpp".
2) Type “make".
3) The DAG code is in the file “output.dag".

3.7.2 How to use the C to DAG converter with the Kylix 3 compiler (Linux OS)

In order to convert C code to a DAG it is necessary to do the following steps:

1) Run the Borland C++ compiler.
2) Open the project “c2DAG_project.bpr".
3) Write the C code to the file “C_code.cpp".
4) Compile the program (Run->Run or press the button <F9>).



5) Press the button <Run>

6) If there is no error press the button <Solution done>
7) The DAG code is in the file “output.dag".

3.7.3 How to use the C to DAG converter with the Borland C++ Builder 6.0 compiler (Windows OS)

The steps are exactly the same as in the case of the Kylix 3 compiler (Linux OS).

3.7.4 Conversion of a DAG file to C language (c_write)

Each DAG can be transformed into a C program using the `c_write` program. If the DAG is written in a file "test.dag" then we can convert it using the following command:

```
./c_write test.dag
```

In this case we assume that the program c_write is in the current directory. The results are shown in the screen. In order to write the results to a file (for example test.c) we have to use the following command.

```
./c_write test.dag > test.c
```

3.7.5 Example

By applying `c2dag` to the C++ code in Section 3.6.1, we obtain the DAG file

```
<N> V 0
<N> c 0 0
<0> b [0,1]: V 0
<N> c 1 1
<1> b [1,2]: V 1
<N> c 2 2
<2> b [2,3]: V 2
<3> +
<E> 3 0
<E> 3 1
<N> c 3 4
<4> b [0,0]: +
<E> 4 3
<6> +
<E> 6 0
<E> 6 1
<7> *
<E> 7 6
<E> 7 2
<N> M 7 min
```

If we apply the `c_write` program to this DAG code then we get the following C program.

```
#include <math.h>

double objective(double* x)
{
  return x[2] * (x[0] + x[1]);
}
```

```
double constraint(int i, double* x)
{
  switch(i)
  {
    case 0: /* bound: [0, 1], node: 0 */
      return x[0];
    case 1: /* bound: [1, 2], node: 1 */
      return x[1];
    case 2: /* bound: [2, 3], node: 2 */
      return x[2];
    case 3: /* linear: [0, 0], node: 4 */
      return x[0] + x[1];
  }
}
```