Branch & Cut & M INTO

Jeff Linderoth

June 22, 2004
Outline

- Review
- Branch and Cut
- A hopefully gentle introduction to MINTO
- An even gentler introduction to AMPL (Time permitting)

- A Simplified version of the Branch and Cut Algorithm
Branch \rightarrow Cut? \rightarrow Solve \rightarrow Prep. \rightarrow Select \rightarrow Prep. \rightarrow Init

Feas? \rightarrow \rightarrow \rightarrow \rightarrow
MINTO

- MINTO is a flexible (relatively) powerful solver for general mixed integer programs

- `minto [-xo> m<>t<>be<>E<>p<>hcikgfrRB<>sn<>a] <name>`

- MINTO has many “advanced” features to solve MIP problems “right out of the box”, but also allows users to customize portions of the branch and cut (and price) algorithms

- Written in C.

- User application functions also written in C
## MINTO options

<table>
<thead>
<tr>
<th>option</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>assume maximization problem</td>
</tr>
<tr>
<td>o &lt; 0, 1, 2, 3 &gt;</td>
<td>level of output</td>
</tr>
<tr>
<td>m &lt; ... &gt;</td>
<td>maximum number of nodes to be evaluated</td>
</tr>
<tr>
<td>t &lt; ... &gt;</td>
<td>maximum cpu time in seconds</td>
</tr>
<tr>
<td>b</td>
<td>deactivate bound improvement (reduced cost fixing)</td>
</tr>
<tr>
<td>e &lt; 0, 1, 2, 3, 4, 5 &gt;</td>
<td>type of branching</td>
</tr>
<tr>
<td>E &lt; 0, 1, 2, 3, 4 &gt;</td>
<td>type of node selection</td>
</tr>
<tr>
<td>p &lt; 0, 1, 2, 3 &gt;</td>
<td>level of preprocessing and probing</td>
</tr>
<tr>
<td>h</td>
<td>deactivate primal heuristic</td>
</tr>
<tr>
<td>c</td>
<td>deactivate clique generation</td>
</tr>
<tr>
<td>i</td>
<td>deactivate implication generation</td>
</tr>
<tr>
<td>k</td>
<td>deactivate knapsack cover generation</td>
</tr>
<tr>
<td>g</td>
<td>deactivate GUB cover generation</td>
</tr>
<tr>
<td>f</td>
<td>deactivate flow cover generation</td>
</tr>
<tr>
<td>r</td>
<td>deactivate row management</td>
</tr>
<tr>
<td>R</td>
<td>deactivate restarts</td>
</tr>
<tr>
<td>B</td>
<td>&lt; 0, 1, 2 &gt; type of forced branching</td>
</tr>
<tr>
<td>s</td>
<td>deactivate all system functions</td>
</tr>
<tr>
<td>n &lt; 1, 2, 3 &gt;</td>
<td>activate a names mode</td>
</tr>
<tr>
<td>a</td>
<td>activate use of advance basis</td>
</tr>
</tbody>
</table>
Branching and Node Selection

- $e < 0, 1, 2, 3, 4, 5 >$
  - maximum infeasibility (0),
  - penalty based (1),
  - strong branching (2),
  - pseudocost based (3),
  - adaptive (4),
  - SOS branching (5).

- $E < 0, 1, 2, 3, 4 >$
  - best bound (0),
  - depth first (1),
  - best projection (2),
  - best estimate (3), and
  - adaptive (4).
Building and Running MINTO

cd APPL
make -f Makefile.Linux.OsiClp
cp p0033.mps .
./minto -o2 p0033 > minto-default.out
./minto -s -o2 p0033 > minto-naive.out

- OK, great, but now I want to use MINTO to customize the branch and cut procedure for my problem.
- To do that, we need to learn a few functions
A call to `inq_form()` initializes the variable `info_form` that has the following structure:

```c
typedef struct info_form {
    int form_vcnt; /* number of variables in the formulation */
    int form_ccnt; /* number of constraints in the formulation */
} INFO_FORM;
```

So obviously it used used to determine the current size of the formulation (of the relaxation being solved).
#include <stdio.h>
#include "minto.h"

void
WriteSize ()
{
    inq_form ();
    printf("Number of variables: %d\n", info_form.form_vcnt);
    printf("Number of constraints: %d\n", info_form.form_ccnt);
}

• To inquire about the entities making up the formulation, you use the function inq_var(j, NO) and inq_constr(i)
  ◦ The i,j are the index of the variable or constraint about which you are inquiring, and the NO means that you do not which to retrive the column information (in the matrix) for this variable.
typedef struct info_var {
    char    *var_name;    /* name, if any */
    char    var_class;    /* class: CONTINUOUS, INTEGER, or BINARY */
    double  var_obj;     /* objective function coefficient */
    int     var_nz;      /* number of constraints with nonzero coefficients */
    int     *var_ind;    /* indices of constraints with nonzero coefficients */
    double  *var_coef;   /* actual coefficients */
    int     var_status;  /* ACTIVE, INACTIVE, or DELETED */
    double  var_lb;      /* lower bound */
    double  var_ub;      /* upper bound */
    VLB     *var_vlb;     /* associated variable lower bound */
    VUB     *var_vub;     /* associated variable upper bound */
    int     var_lb_info; /* ORIGINAL, MODIFIED_BY_MINTO, MODIFIED_BY_BRANCHING, or MODIFIED_BY_APPL */
    int     var_ub_info; /* ORIGINAL, MODIFIED_BY_MINTO, MODIFIED_BY_BRANCHING, or MODIFIED_BY_APPL */
} INFO_VAR;
• If \( y_j \leq u_j x_j, (x_j \in \{0, 1\}) \), \( y_j \) is said to have a \textit{variable upper bound}.

• These are used to generate some classes of strong valid inequalities

```c
typedef struct {
    int vlb_var; /* index of associated 0-1 variable */
    double vlb_val; /* value of associated bound */
} VLB;

typedef struct {
    int vub_var; /* index of associated 0-1 variable */
    double vub_val; /* value of associated bound */
} VUB;
```
Example of inq_var()

```c
#include <stdio.h>
#include "minto.h"

void
WriteFixed ()
{
    int j;
    int nvar;

    inq_form();
    nvar = info_form.form_vcnt;
    for (j = 0; j < nvar; j++) {
        inq_var (j, NO);
        if (info_var.var_lb > info_var.var_ub - 1.0e-6) {
            printf ("Variable %d is fixed at %f\n", j, info_var.var_lb);
        }
    }
}
```
typedef struct info_constr {
    char    *constr_name;  /* name, if any */
    int     constr_class; /* classification: ... */
    int     constr_nz;   /* number of variables with nonzero coefficients */
    int     *constr_ind; /* indices of variables with nonzero coefficients */
    double  *constr_coef; /* actual coefficients */
    char    constr_sense; /* sense */
    double  constr_rhs;  /* right hand side */
    int     constr_status; /* ACTIVE, INACTIVE, or DELETED */
    int     constr_type;  /* LOCAL or GLOBAL */
    int     constr_info;  /* ORIGINAL, GENERATED_BY_MINTO, 
                           GENERATED_BY_BRANCHING, or GENERATED_BY_APPL */
} INFO_CONSTR;
#include <stdio.h>
#include "minto.h"

void
WriteType ()
{
  int i;

  for (inq_form (), i = 0; i < info_form.form_ccnt; i++) {
    inq_constr (i);
    printf ("Constraint %d is of type %s\n",
            i, info_constr.constr_type == GLOBAL ? "GLOBAL" : "LOCAL");
  }
}
### Constraint Classes in MINTO

<table>
<thead>
<tr>
<th>class</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIXUB</td>
<td>$\sum_{j \in B} a_j x_j + \sum_{j \in I \cup C} a_j y_j \leq a_0$</td>
</tr>
<tr>
<td>MIXEQ</td>
<td>$\sum_{j \in B} a_j x_j + \sum_{j \in I \cup C} a_j y_j = a_0$</td>
</tr>
<tr>
<td>NOBINUB</td>
<td>$\sum_{j \in I \cup C} a_j y_j \leq a_0$</td>
</tr>
<tr>
<td>NOBINEQ</td>
<td>$\sum_{j \in I \cup C} a_j y_j = a_0$</td>
</tr>
<tr>
<td>ALLBINUB</td>
<td>$\sum_{j \in B} a_j x_j \leq a_0$</td>
</tr>
<tr>
<td>ALLBINEQ</td>
<td>$\sum_{j \in B} a_j x_j = a_0$</td>
</tr>
<tr>
<td>SUMVARUB</td>
<td>$\sum_{j \in I \cup C^+} a_j y_j - a_k x_k \leq 0$</td>
</tr>
<tr>
<td>SUMVAREQ</td>
<td>$\sum_{j \in I \cup C^+} a_j y_j - a_k x_k = 0$</td>
</tr>
<tr>
<td>VARUB</td>
<td>$a_j y_j - a_k x_k \leq 0$</td>
</tr>
<tr>
<td>VAREQ</td>
<td>$a_j y_j - a_k x_k = 0$</td>
</tr>
<tr>
<td>VARLB</td>
<td>$a_j y_j - a_k x_k \geq 0$</td>
</tr>
<tr>
<td>BINSUMVARUB</td>
<td>$\sum_{j \in B \setminus {k}} a_j x_j - a_k x_k \leq 0$</td>
</tr>
<tr>
<td>BINSUMVAREQ</td>
<td>$\sum_{j \in B \setminus {k}} a_j x_j - a_k x_k = 0$</td>
</tr>
<tr>
<td>BINSUM1VARUB</td>
<td>$\sum_{j \in B \setminus {k}} x_j - a_k x_k \leq 0$</td>
</tr>
<tr>
<td>BINSUM1VAREQ</td>
<td>$\sum_{j \in B \setminus {k}} x_j - a_k x_k = 0$</td>
</tr>
<tr>
<td>BINSUM1UB</td>
<td>$\sum_{j \in B} x_j \leq 1$</td>
</tr>
<tr>
<td>BINSUM1EQ</td>
<td>$\sum_{j \in B} x_j = 1$</td>
</tr>
</tbody>
</table>
Adapting MINTO. **appl_constraints()**

```c
unsigned
appl_constraints (id, zlp, xlp, zprimal, xprimal, nzcnt, ccnt, cfirst,
                   cind, ccoef, csense, crhs, ctype, cname, sdim, ldim)

  int id;    /* identification of active minto */
  double zlp; /* value of the LP solution */
  double *xlp; /* values of the variables */
  double zprimal; /* value of the primal solution */
  double *xprimal; /* values of the variables */
  int *nzcnt; /* variable for number of nonzero coefficients */
  int *ccnt; /* variable for number of constraints */
  int *cfirst; /* array for positions of first nonzero coefficients */
  int *cind; /* array for indices of nonzero coefficients */
  double *ccoef; /* array for values of nonzero coefficients */
  char *csense; /* array for senses */
  double *crhs; /* array for right hand sides */
  int *ctype; /* array for the constraint types: LOCAL or GLOBAL */
  int **cname; /* array for the names */
  int sdim; /* length of small arrays */
  int ldim; /* length of large arrays */
{
}
```
Using `appl_constraints()`

- Suppose after some processing, I realize that I would like to add three cutting planes to the global formulation of my IP instance.

\[
\begin{align*}
  x_1 + 2x_2 & \leq 7 \\
  x_1 + x_2 - x_3 & \leq 2 \\
  -7x_1 + x_4 & \geq 0
\end{align*}
\]
C Code Example in appl_constraints()

/* Number of constraints */
*ccnt = 3;

/* Number of nonzeros */
*nzcnt = 7;

cfirst[0] = 0;
cfirst[1] = 2;
cfirst[2] = 5;
cfirst[3] = 7;

cind[0] = 0;
cind[1] = 1;
cind[2] = 0;
cind[3] = 1;
cind[4] = 2;
cind[5] = 0;
cind[6] = 3;

ccoef[0] = 1.0;
ccoef[1] = 2.0;
ccoef[2] = 1.0;
ccoef[3] = 1.0;
ccoef[4] = -1.0;
ccoef[5] = -7.0;
ccoef[6] = 1.0;

csense[0] = 'L';
csense[1] = 'L';
csense[2] = 'G';

crhs[0] = 7.0;
crhs[1] = 2.0;
crhs[2] = 0.0;

cctype[0] = GLOBAL;
cctype[1] = GLOBAL;
cctype[2] = GLOBAL;

cname[0] = '\0';
cname[1] = '\0';
cname[2] = '\0';

return(SUCCESS);

- Don’t worry – we’ll see more examples, and we’ll be here to help!
• AMPL is an Algebraic Modeling Language

• In many ways, AMPL is like any other programming language.
  ◊ It just has special syntax that helps us create an
    optimization instance and interact with optimization
    solvers.

• AMPL is a very useful tool for building and solving
  optimization instances, but it is not too user friendly!
PPP – A Production Planning Problem

- An engineering plant can produce five types of products: $p_1, p_2, \ldots, p_5$ by using two production processes: grinding and drilling. Each product requires the following number of hours of each process, and contributes the following amount (in hundreds of dollars) to the net total profit.

<table>
<thead>
<tr>
<th></th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
<th>$p_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Grinding</strong></td>
<td>12</td>
<td>20</td>
<td>0</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td><strong>Drilling</strong></td>
<td>10</td>
<td>8</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Profit</strong></td>
<td>55</td>
<td>60</td>
<td>35</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>
• Each unit of each product takes 20 manhours for “final assembly”.

• The factory has three grinding machines and two drilling machines.

• The factory works a six day week with two shifts of 8 hours/day. Eight workers are employed in assembly, each working one shift per day.

• $x_i$: The number of product $p_i$ to make in a week.
• Grinding...
  ◊ 3 machines. 16 hours/day. 6 days/week.

⋆ Get the Units right!

• 288 grinding hours available per week.
  ◊ 3 machines * 16 grinding hours/(machine*day) * 6 days/week = 288 grinding hours/week.

\[12x_1 + 20x_2 + 0x_3 + 25x_4 + 15x_5 \leq 288\]

• LHS : Grinding hours in production plan per week
• RHS : Total grinding hours available per week.
More Constraints...

- Drilling
  - $10x_1 + 8x_2 + 16x_3 + 0x_4 + 0x_5 \leq 2 \times 16 \times 6 = 192$

- Finishing Labor
  - 8 Assembly workers, each working 48 hours/week.
    - $20x_1 + 20x_2 + 20x_3 + 20x_4 + 20x_5 \leq 8 \times 48 = 384$

- The Laws of Nature
  - $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0, x_5 \geq 0$. 
Final Problem

maximize

\[ 55x_1 + 60x_2 + 350x_3 + 40x_4 + 20x_5 \] (Profit/week)

subject to

\[ 12x_1 + 20x_2 + 0x_3 + 25x_4 + 15x_5 \leq 288 \]
\[ 10x_1 + 8x_2 + 16x_3 + 0x_4 + 0x_5 \leq 192 \]
\[ 0x_1 + 20x_2 + 20x_3 + 20x_4 + 20x_5 \leq 384 \]

\[ x_i \geq 0 \quad \forall i = 1, 2, \ldots 5 \]

* AMPL Interactive Portion*
Generalizing the Model

• Suppose we want to generalize the model to more (or less) than five products.

• Suppose we wanted to have more than three resources constraining us?

• Suppose we wanted to change certain parameters associated with the model?
  ∗ AMPL (and all “real” modeling environments) allow the model to be separated from the data.
  ∗ This is IMPORTANT!!!
General PPP Model

- Sets
  - $P$: Set of products to be made
  - $R$: Set of resources available (constraining our production)

- Parameters
  - $c_p$: Net profit of producing one unit of product $p$ ($\forall p \in P$)
  - $b_r$: Amount of resource $r$ available ($\forall r \in R$)

- Variables
  - $x_p$: Amount of product $p$ to produce ($\forall p \in P$)
AMPL Entities

- Data
  - Sets: lists of products, materials, etc.
  - Parameters: numerical inputs such as costs, etc.

- Model
  - Variables: The values to be decided upon.
  - Objective Function.
  - Constraints.

- These are usually stored in different files.

★ AMPL Interactive Portion
An AMPL Template

- Define Sets
- Define Parameters
- Define Variables
  - Also can define variable bound constraints in this section
- Define Objective
- Define Constraints
Important AMPL Keywords/Syntax

- model file.mod;
- data file.mod;
- reset;
- quit;

- set
- param
- var
- maximize (minimize)
- subject to
Important AMPL Notes

- The `#` character starts a comment.
- All statements must end in a semi-colon;
- Names must be unique!
  - A variable and a constraint cannot have the same name.
- AMPL is case sensitive. Keywords must be in lower case.
- Even if the AMPL error message is cryptic, look at the location where it shows an error – this will often help you deduce what is wrong.
- See `papers/ampl1.pdf` for a short introduction to AMPL.
- I also have brought a couple AMPL books for us to use.