DIMACS Reconnect Conference on MIP

Branch & Cut &M INTO

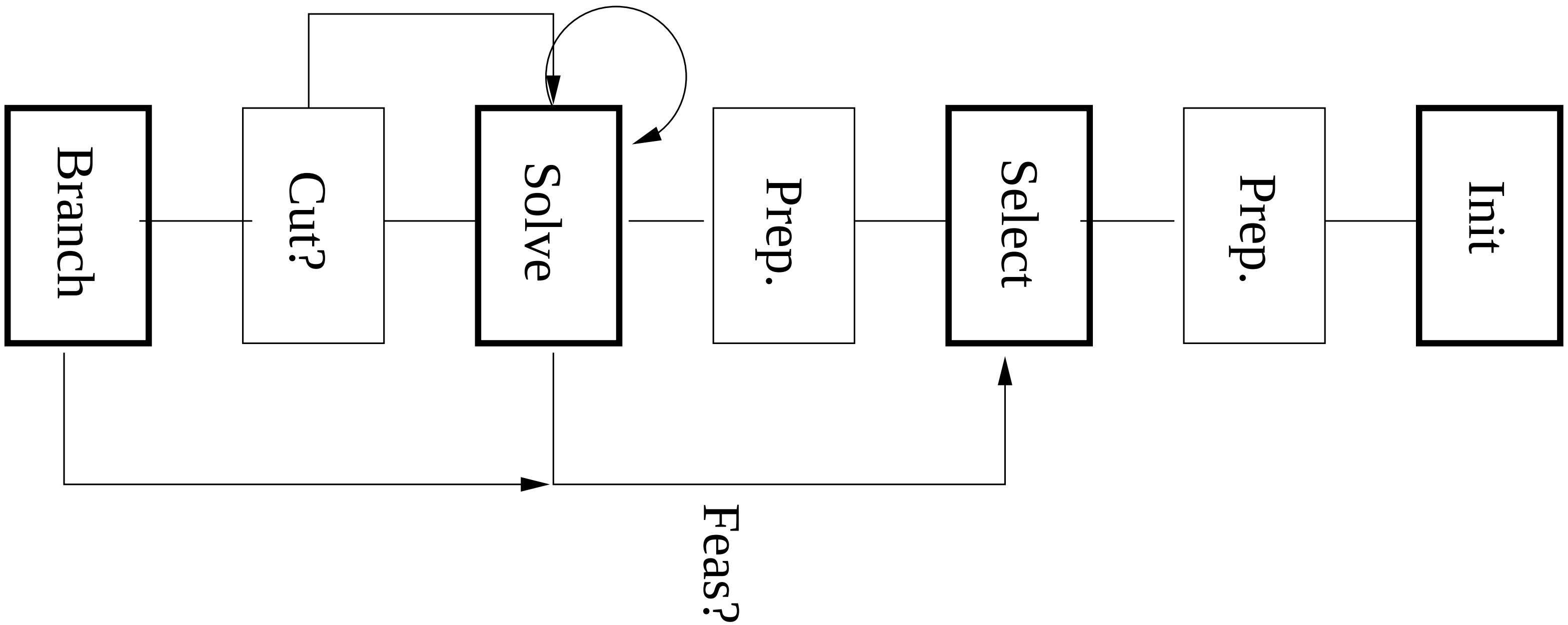
Jeff Linderoth

June 22, 2004

Outline

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

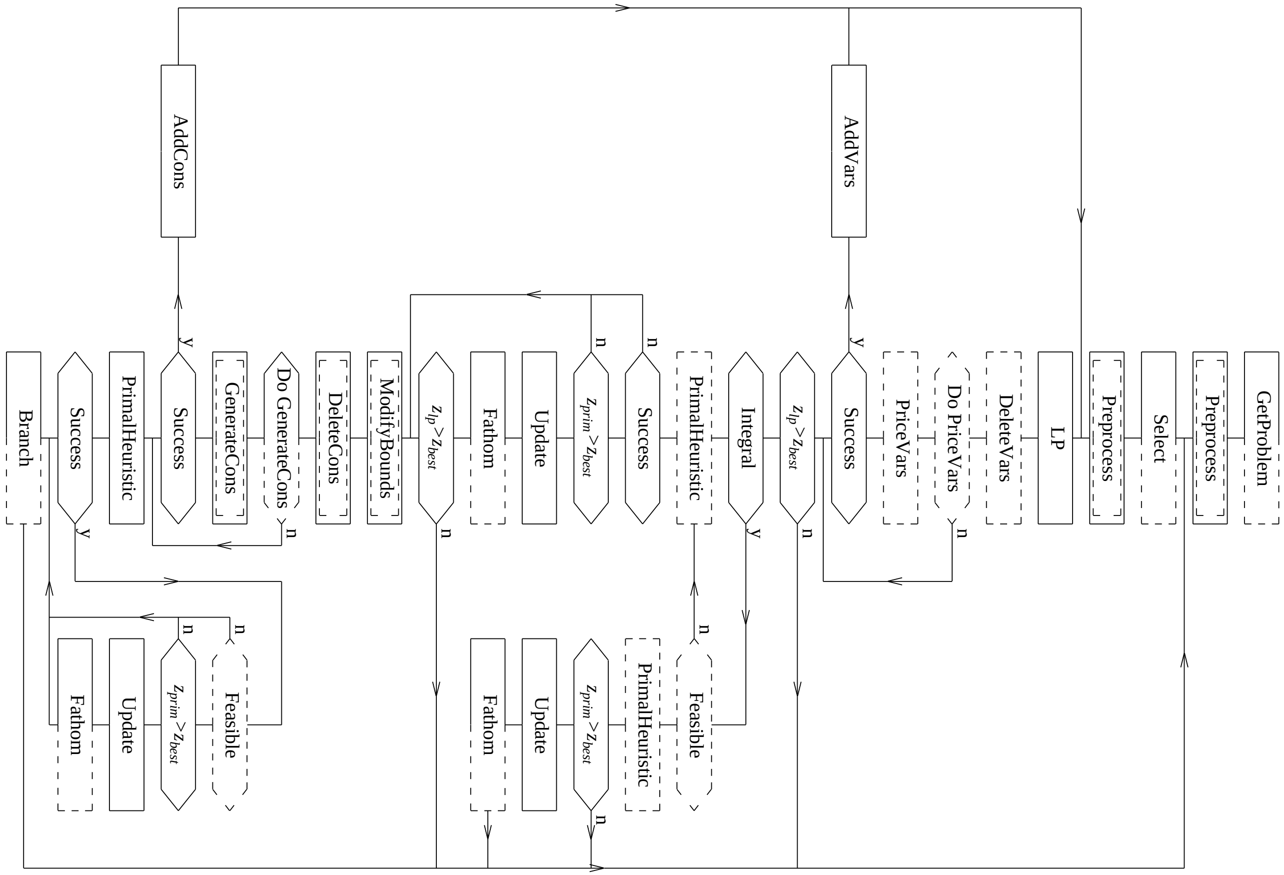
- A Simplified version of the Branch and Cut Algorithm

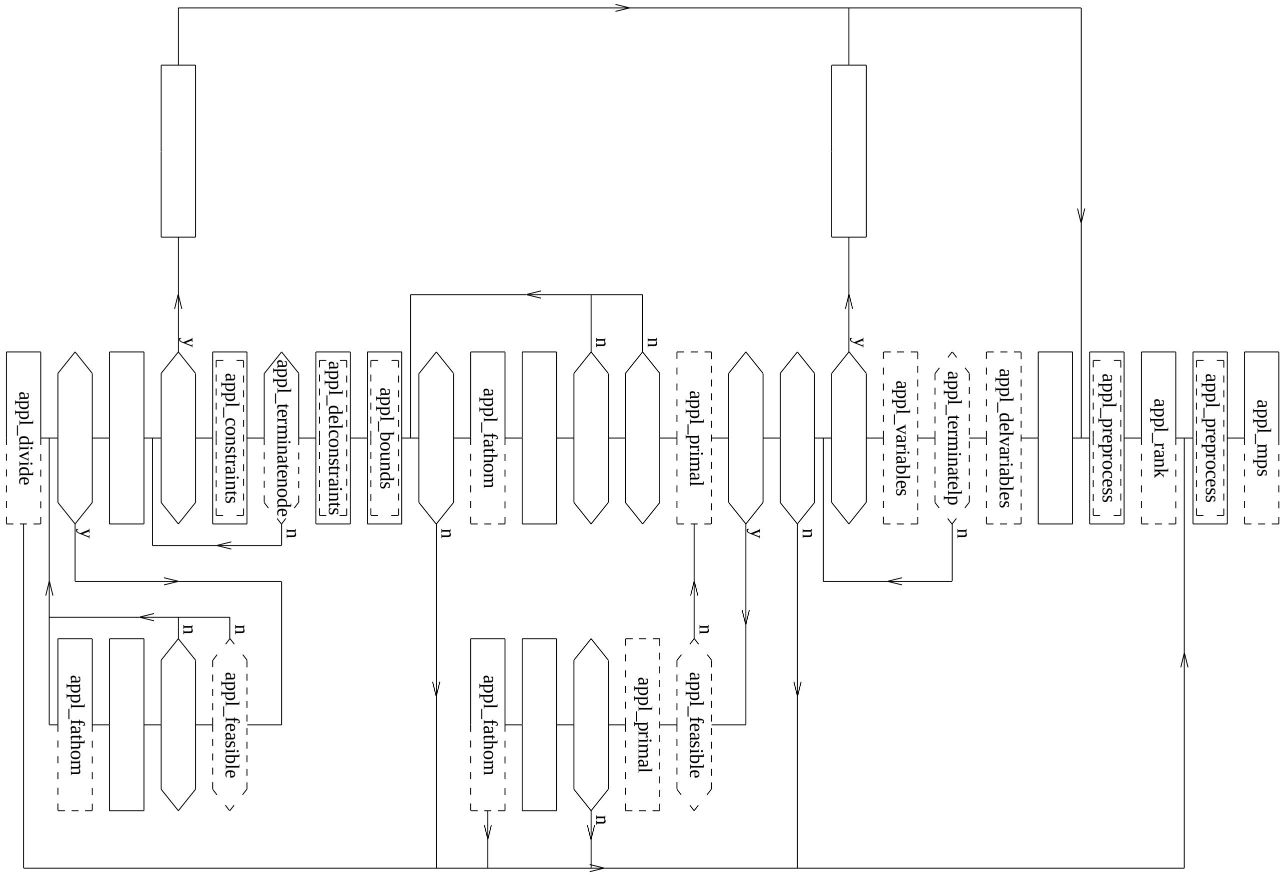

Branch
Cut?
Solve
Prep.
Select
Prep.
Init
Feas?

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


AddCons
AddVars
Branch
Success
PrimalHeuristic
Success
GenerateCons
Do GenerateCons
DeleteCons
ModifyBounds
$z_{lp} > z_{best}$
Fathom
Update
$z_{prim} > z_{best}$
Success
PrimalHeuristic
Integral
$z_{lp} > z_{best}$
Success
PriceVars
Do PriceVars
DeleteVars
LP
Preprocess
Select
Preprocess
GetProblem
Fathom
Update
$z_{prim} > z_{best}$
Feasible
Fathom
Update
$z_{prim} > z_{best}$
PrimalHeuristic
Feasible
y
n


appl_mps
appl_preprocess
appl_rank
appl_preprocess
appl_delvariables
appl_terminatelp
appl_variables
appl_primal
appl_fathom
appl_bounds
appl_delconstraints
appl_terminatenode
appl_constraints
appl_divide
appl_feasible
appl_primal
appl_fathom
appl_feasible
appl_fathom
y
n

MINTO options

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

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

`inq_form()`

- A call to **`inq_form()`** initializes the variable *info_form* that has the following structure:

```
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).

inq_form() example

```
#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.

inq_var()

```
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;
```

inq_var() Cont.

- If $y_j \leq u_j x_j, (x_j \in \{0,1\})$, y_j is said to have a *variable upper bound.*
- These are used to generate some classes of strong valid inequalities

```
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()

```
#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);
      }
    }
}
```

inq_constr

```
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;
```

inq_constr() Example

```
#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

class	constraint
MIXUB	$\sum_{j\in B} a_j x_j + \sum_{j\in I\cup C} a_j y_j \leq a_0$
MIXEQ	$\sum_{j\in B} a_j x_j + \sum_{j\in I\cup C} a_j y_j = a_0$
NOBINUB	$\sum_{j\in I\cup C} a_j y_j \leq a_0$
NOBINEQ	$\sum_{j\in I\cup C} a_j y_j = a_0$
ALLBINUB	$\sum_{j\in B} a_j x_j \leq a_0$
ALLBINEQ	$\sum_{j\in B} a_j x_j = a_0$
SUMVARUB	$\sum_{j\in I^+\cup C^+} a_j y_j - a_k x_k \leq 0$
SUMVAREQ	$\sum_{j\in I^+\cup C^+} a_j y_j - a_k x_k = 0$
VARUB	$a_j y_j - a_k x_k \leq 0$
VAREQ	$a_j y_j - a_k x_k = 0$
VARLB	$a_j y_j - a_k x_k \geq 0$
BINSUMVARUB	$\sum_{j\in B\setminus\{k\}} a_j x_j - a_k x_k \leq 0$
BINSUMVAREQ	$\sum_{j\in B\setminus\{k\}} a_j x_j - a_k x_k = 0$
BINSUM1VARUB	$\sum_{j\in B\setminus\{k\}} x_j - a_k x_k \leq 0$
BINSUM1VAREQ	$\sum_{j\in B\setminus\{k\}} x_j - a_k x_k = 0$
BINSUM1UB	$\sum_{j\in B} x_j \leq 1$
BINSUM1EQ	$\sum_{j\in B} x_j = 1$

Adapting MINTO. appl_constraints()

```
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.

$$
\begin{aligned}
x_1 + 2x_2 &\leq 7 \\
x_1 + x_2 - x_3 &\leq 2 \\
-7x_1 + x_4 &\geq 0
\end{aligned}
$$

C Code Example in appl_constraints()

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

/* Number of nonzeroes */
*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;

ctype[0] = GLOBAL;
ctype[1] = GLOBAL;
ctype[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

- 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.

	p_1	p_2	p_3	p_4	p_5
Grinding	12	20	0	25	15
Drilling	10	8	16	0	0
Profit	55	60	35	40	20

PPP – More Info

- Each unit of each product take 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.

Constraints

- 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.

$$12x_1 + 20x_2 + 0x_3 + 25x_4 + 15x_5 \le 288$$

- LHS : Grinding hours in production plan per week
- RHS : Total grinding hours available per week.

More Constraints...

- Drilling
  - $10x_1 + 8x_2 + 16x_3 + 0x_4 + 0x_5 \leq 2 * 16 * 6 = 192$
- Finishing Labor
  - 8 Assembly workers, each working 48 hours/week.
  - $20x_1 + 20x_2 + 20x_3 + 20x_4 + 20x_5 \leq 8 * 48 = 384$
- 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

$$55x_1 + 60x_2 + 350x_3 + 40x_4 + 20x_5 \qquad \text{(Profit/week)}$$

subject to

$$\begin{aligned} 12x_1 + 20x_2 + 0x_3 + 25x_4 + 15x_5 &\leq 288 \\ 10x_1 + 8x_2 + 16x_3 + 0x_4 + 0x_5 &\leq 192 \\ 0x_1 + 20x_2 + 20x_3 + 20x_4 + 20x_5 &\leq 384 \\ x_i &\geq 0 \qquad \forall i = 1, 2, \ldots 5 \end{aligned}$$

⋆ 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 brough a couple AMPL books for us to use