IBM

The GNU Linear Programming Kit, Part 2: Intermediate problems in linear programming

Cleaning up the post office and making the most of a bad diet

Rodrigo Ceron (rceron@br.ibm.com), Staff Software Engineer, IBM

Summary: This article continues the series on using the GNU Linear Programming Kit and the glpsol client utility with the GNU MathProg language. In this installment, a diet problem shows you how to formulate a simple multi-variable and declare bidimensional parameters. A post office resource allocation problem then introduces MathProg expressions and integer-only decision variables.

Date: 07 Sep 2006
Level: Intermediate
Activity: 4588 views
Comments:

This article is the second in a three-part series on the GNU Linear Programming Kit (GLPK). For an introduction to GLPK, read the first installment in the series, "The GNU Linear Programming Kit, Part 1: Introduction to linear optimization."

The diet problem

This problem comes from *Operations Research: Applications and Algorithms, 4th Edition*, by Wayne L. Winston (Thomson, 2004); see Resources below for a link.

> My diet requires that all the food I eat come from one of the four "basic food groups": chocolate cake, ice cream, soda, and cheesecake. At present, the following four foods are available for consumption: brownies, chocolate ice cream, cola, and pineapple cheesecake. Each brownie costs $0.50, each scoop of chocolate ice cream costs $0.20, each bottle of cola costs $0.30, and each piece of pineapple cheesecake $0.80. Each day, I must ingest at least 500 calories, 6 oz of chocolate, 10 oz of sugar, and 8 oz of fat. The nutrition content per unit of each food is shown in the table below. Satisfy my daily nutritional requirements at minimum cost.

To summarize the important information about the problem:

Food cost per unit

- Brownie: $0.50
- Chocolate ice cream: $0.20 (scoop)
- Soda: $0.30 (bottle)

- Pineapple cheesecake: $0.80 (piece)

Daily food needs

- 500 calories
- 6 oz chocolate
- 10 oz sugar
- 8 oz fat

Food nutritional content

- Brownie: 400 calories, 3 oz chocolate, 2 oz sugar, 2 oz fat
- Chocolate ice cream (scoop): 200 calories, 2 oz chocolate, 2 oz sugar, 4 oz fat
- Soda (1 bottle): 150 calories, 0 oz chocolate, 4 oz sugar, 1 oz fat
- Pineapple cheesecake (1 piece): 500 calories, 0 oz chocolate, 4 oz sugar, 5 oz fat

---

Modeling

Modeling this diet problem should be easier after solving Giapetto's problem in Part 1. Let's determine the decision variables first.

The goal is to satisfy the daily nutritional needs at a minimum cost. Therefore, the decision variables are the amount of each food type to buy:

Food variables

- x_1: brownies
- x_2: scoops of chocolate ice cream
- x_3: bottles of cola
- x_4: pieces of pineapple cheesecake

Now the objective (target) function can be written in terms of the decision variables. The cost of the diet z needs to be minimized:

$$z = 0.5x_1 + 0.2x_2 + 0.3x_3 + 0.8x_4$$

The next step is to write the inequalities for the constraints. Note that there's a minimum amount of calories, chocolate, sugar, and fat that the food in the diet should provide. Each of these four requirements is a constraint, so the constraints can be written like this:

$$400x_1 + 200x_2 + 150x_3 + 500x_4 \geq 500 \ (Calorie\ constraint)$$

$$3x_1 + 2x_2 \geq 6 \ (Chocolate\ constraint)$$

Note that pineapple cheesecake and soda do not contain chocolate.

$$2x_1 + 2x_2 + 4x_3 + 4x_4 \geq 10 \ (Sugar\ constraint)$$

$$2x_1 + 4x_2 + x_3 + 5x_4 \geq 8 \ (Fat\ constraint)$$

And finally, the sign constraints:

$$x_i \geq 0, \ \forall i \in \{1, .., 4\}$$

Try to imagine the feasible region for this problem. The problem has four variables. So, the universe has four axes. That's hard to plot, so use your imagination. At first the solution could be anywhere in the four-dimensional space. With each constraint, the solution space shrinks to what looks like a polyhedron.

GNU MathProg solution for the diet problem

Note: The line numbers in Listing 1 are for reference only.

Listing 1. GNU MathProg solution of the diet problem

```
 1      #
 2      # Diet problem
 3      #
 4      # This finds the optimal solution for minimizing the cost of my diet
 5      #
 6
 7      /* sets */
 8      set FOOD;
 9      set NEED;
10
11      /* parameters */
12      param NutrTable {i in FOOD, j in NEED};
13      param Cost {i in FOOD};
14      param Need {j in NEED};
15
16      /* decision variables: x1: Brownie, x2: Ice cream, x3: soda, x4: cake*/
17      var x {i in FOOD} >= 0;
18
19      /* objective function */
20      minimize z: sum{i in FOOD} Cost[i]*x[i];
21
22      /* Constraints */
23      s.t. const{j in NEED} : sum{i in FOOD} NutrTable[i,j]*x[i] >= Need[j];
24
25      /* data section */
26      data;
27
28      set FOOD := Brownie "Ice cream" soda cake;
29      set NEED := Calorie Chocolate Sugar Fat;
30
31      param NutrTable: Calorie        Chocolate       Sugar   Fat:=
32      Brownie          400            3               2       2
33      "Ice cream"      200            2               2       4
34      soda             150            0               4       1
35      cake             500            0               4       5;
36
37      param Cost:=
```

```
38      Brownie         0.5
39      "Ice cream"     0.2
40      soda            0.3
41      cake            0.8;
42
43      param Need:=
44      Calorie         500
45      Chocolate       6
46      Sugar           10
47      Fat             8;
48
49      end;
```

Lines 8 and 9 define two sets: `FOOD` and `NEED`, but the elements of those two sets are declared in the data section on lines 28 and 29. The `FOOD` set contains the four food types given.

Because the space character separates the elements of a set, the `Ice cream` element needs double quotes around the name (instead of the less attractive `icecream`, for example). If you want to use non-ASCII characters in MathProg names, you should use double quotes as well even if there are no spaces.

Back to the model section. The `NEED` set holds the four dietary needs. Lines 12, 13, and 14 define the problem parameters: `Need`, `Cost`, and `NutrTable` (the nutrition table). There is a cost for each `FOOD` item. There is an amount for each nutritional need in the `NEED` set. Try to use a differently named index variable for each set consistently; it's a good idea, especially when debugging. In this case, the `FOOD` set uses `i` and the `NEED` set uses `j`. The `Cost` and `Need` parameters in the data section are defined in lines 37 through 47.

The nutritional table defined on line 12 and filled with data in lines 31 through 35 has two dimensions, since each food provides multiple nutritional values. Therefore, the nutrition table `NutrTable` is a mapping between the *FOOD* and *NEED* sets. Note that the row and column order is the same as the element order in each set, and the index variable names are consistent between lines 12, 13, and 14.

Throughout this exercise, the `i` axes are the rows, and the `j` axes are the columns as expected by most mathematicians in the audience. The syntax for two-dimensional parameter declaration (up to N columns and M rows) is:

Listing 2. Syntax for two-dimensional parameter declaration

```
param label :   J_SET_VAL_1     J_SET_VAL_2     ...     J_SET_VAL_N :=
I_SET_VAL_1     param_1_1       param_1_2       ...     param_1_N
I_SET_VAL_2     param_2_1       param_2_2       ...     param_2_N
...             ...             ...             ...     ...
I_SET_VAL_M     param_M_1       param_M_2       ...     param_M_N;
```

Don't forget the `:=` at the end of the first line, and the `;` at the end of the last line.

Line 17 declares the decision variables and states that each of them can't be a negative value. It's a very simple definition, because the array `x` is defined automatically with the elements of the `FOOD` set.

The objective function on line 20 minimizes the total food cost as the total of each decision variable (amount of food) multiplied by that food's cost per unit. Note that the `i` index is on the `FOOD` set.

Line 23 declares all the four dietary constraints. It's written in a very compact and elegant style, so

pay attention! The definition on the left of the colon : tells GLPK to create a constraint for each need in the `NEED` set: `const[Calorie]`, `const[Chocolate]`, `const[Sugar]`, and `const[Fat]`. Each of these constraints takes every nutritional need, and adds up the amount of each food type multiplied by how much of that need it can satisfy per food unit; this total must be equal to or greater than the minimal need for that nutrient for a balanced diet.

Expanded, this declaration would look like this (`i` covers the `FOOD` set):

$$const[Calorie] : \sum_i NutrTable[i, Calorie] >= Need[Calorie]$$

$$const[Chocolate] : \sum_i NutrTable[i, Chocolate] >= Need[Chocolate]$$

$$const[Sugar] : \sum_i NutrTable[i, Sugar] >= Need[Sugar]$$

$$const[Fat] : \sum_i NutrTable[i, Fat] >= Need[Fat]$$

Listing 3. The output of glpsol for this problem

```
Problem:    diet
Rows:       5
Columns:    4
Non-zeros:  18
Status:     OPTIMAL
Objective:  z = 0.9 (MINimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------ ------------- ------------- -------------
     1 z            B            0.9
     2 const[Calorie]
                    B            750           500
     3 const[Chocolate]
                    NL             6             6                      0.025
     4 const[Sugar] NL            10            10                      0.075
     5 const[Fat]   B             13             8

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------ ------------- ------------- -------------
     1 x[Brownie]   NL             0             0                      0.275
     2 x['Ice cream']
                    B              3             0
     3 x[soda]      B              1             0
     4 x[cake]      NL             0             0                        0.5

Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err. = 1.82e-13 on row 2
        max.rel.err. = 2.43e-16 on row 2
        High quality

KKT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

KKT.DE: max.abs.err. = 5.55e-17 on column 3
        max.rel.err. = 4.27e-17 on column 3
```

```
        High quality

KKT.DB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

End of output
```

These results show that the minimum cost and optimal value of the diet is $0.90. Which constraints have bound this solution?

The second section of the report says that both the chocolate and sugar constraints are lower-bounded, so this diet uses the minimum amount of chocolate and sugar required. The marginal tells us that if we could relax the chocolate constraint by one unit (5 oz instead of 6 oz), the objective function would improve by 0.025 (and would go from 0.9 to 0.875). Analogously, if we could relax the sugar constraint by one unit, the objective function would improve by 0.075. This makes sense: eat less, pay less. It's important to do these common sense checks on marginals and amounts. For example, if you are told it's optimal to eat 200 lbs. of chocolate and no calories, you should be a little suspicious (and grateful, maybe).

The third section reports the optimal values of the decision variables: 3 scoops of ice cream and one bottle of soda. The brownie and pineapple cheesecake have a marginal value, because their value is at the limit of their sign constraints. This marginal says that if the value of the brownie variable could ever be -1, the objective function would improve by 0.275, but of course that is not useful for this problem's setup.

Post office problem

From "*Operations Research*":

> A post office requires a different number of full-time employees working on different days of the week [summarized below]. Union rules state that each full-time employee must work for 5 consecutive days and then receive two days off. For example, an employee who works on Monday to Friday must be off on Saturday and Sunday. The post office wants to meet its daily requirements using only full-time employees. Minimize the number of employees that must be hired.

To summarize the important information about the problem:

- Every full-time worker works for 5 consecutive days and takes 2 days off
- Day 1 (Monday): 17 workers needed
- Day 2 : 13 workers needed
- Day 3 : 15 workers needed
- Day 4 : 19 workers needed
- Day 5 : 14 workers needed
- Day 6 : 16 workers needed
- Day 7 (Sunday) : 11 workers needed

The post office needs to minimize the number of employees it needs to hire to meet its demand.

Modeling

Let's start with the decision variables for this problem. You could try with seven variables, one for each day of the week, with a value equal to the number of employees that work on each day of the week. Although this seems to solve the problem at first, it can't enforce the constraint that an employee will work five days per week, because working on a given day can't require working on the next day.

The correct approach should guarantee that an employee that starts working on day `i` will also work on the four following days, so the correct approach is to use x_i as the number of employees that start working on day `i`. With this approach, it's much simpler to enforce the consecutiveness constraint. The decision variables are then:

- x_1: number of employees who start working on Monday
- x_2: number of employees who start working on Tuesday
- x_3: number of employees who start working on Wednesday
- x_4: number of employees who start working on Thursday
- x_5: number of employees who start working on Friday
- x_6: number of employees who start working on Saturday
- x_7: number of employees who start working on Sunday

The objective function that needs to be minimized is the amount of employees hired, which is given by:

$$z = \sum_{i=1}^{7} x_i$$

Now, what are the constraints? Each day of the week has a constraint to ensure the minimum amount of workers for that day. Let's take Monday as an example. Who will be working on Mondays? The first (partial) answer that comes to mind is "the people who start working on Mondays." But who else? Well, considering that employees must work for five consecutive days, it's expected that the employees that started working on Sunday will also be working on Monday (remember the problem definition). This same reasoning is used to conclude that the employees who start working on Saturday, Friday, and Thursday will also be working on Monday.

This constraint ensures that there will be at least 17 employees working on Monday.

$$x_4 + x_5 + x_6 + x_7 + x_1 \geq 17 \ \ (Monday\ constraint)$$

Similarly:

$$x_1 + x_2 + x_5 + x_6 + x_7 \geq 13 \ \ (Tuesday\ constraint)$$

$$x_1 + x_2 + x_3 + x_6 + x_7 \geq 15 \ \ (Wednesday\ constraint)$$

$$x_1 + x_2 + x_3 + x_4 + x_7 \geq 19 \ \ (Thursday\ constraint)$$

$$x_1 + x_2 + x_3 + x_4 + x_5 \geq 14 \ \ (Friday\ constraint)$$

$$x_2 + x_3 + x_4 + x_5 + x_6 \geq 16 \ (Saturday\ constraint)$$

$$x_3 + x_4 + x_5 + x_6 + x_7 \geq 11 \ (Sunday\ constraint)$$

Don't forget the sign constraints, of course:

$$x_i \geq 0, \ \forall i \in \{1,..,7\}$$

GNU MathProg solution for the post office problem

Note: The line numbers in Listing 4 are for reference only.

Listing 4. Post office problem solution

```
 1      #
 2      # Post office problem
 3      #
 4      # This finds the optimal solution for minimizing the number of full-time
 5      # employees to the post office problem
 6      #
 7
 8      /* sets */
 9      set DAYS;
10
11      /* parameters */
12      param Need {i in DAYS};
13
14      /* Decision variables. x[i]: No. of workers starting at day i */
15      var x {i in DAYS} >= 0;
16
17      /* objective function */
18      minimize z: sum{i in DAYS} x[i];
19
20      /* Constraints */
21
22      s.t. mon: sum{i in DAYS: i<>'Tue' and i<>'Wed'} x[i] >= Need['Mon'];
23      s.t. tue: sum{i in DAYS: i<>'Wed' and i<>'Thu'} x[i] >= Need['Tue'];
24      s.t. wed: sum{i in DAYS: i<>'Thu' and i<>'Fri'} x[i] >= Need['Wed'];
25      s.t. thu: sum{i in DAYS: i<>'Fri' and i<>'Sat'} x[i] >= Need['Thu'];
26      s.t. fri: sum{i in DAYS: i<>'Sat' and i<>'Sun'} x[i] >= Need['Fri'];
27      s.t. sat: sum{i in DAYS: i<>'Sun' and i<>'Mon'} x[i] >= Need['Sat'];
28      s.t. sun: sum{i in DAYS: i<>'Mon' and i<>'Tue'} x[i] >= Need['Sun'];
29
30      data;
31
32      set DAYS:= Mon Tue Wed Thu Fri Sat Sun;
33
34      param Need:=
35      Mon             17
36      Tue             13
37      Wed             15
38      Thu             19
39      Fri             14
40      Sat             16
41      Sun             11;
42
43      end;
```

Line 9 declares a set named `DAYS`, and its elements (just the days of the week, starting with Monday) are declared on line 32 in the data section.

Line 12 declares the parameter `Need` for each day in the `DAYS` set. Lines 34 through 41 define the values for this parameter: the minimum employees needed on each day of the week.

Line 15 declares the decision variables as an array of seven variables defined on the `DAYS` set, representing the number of people that start work that day.

Lines 22 through 28 define one constraint for each day of the week. Note that it would be too boring to write seven inequalities as a sum of five decision variables that are not necessarily in order, because in some constraints, the index may overlap the index 7. GNU MathProg offers *expressions* for the program writer to make this easier.

Each constraint is the total of all the decision variables, except for those two that should not take part in that specific day (instead of including the five others specifically). This expression is used inside the braces `{}` that define the index of the summation. The syntax for an expression is:

```
{index_variable in your_set: your_expression}
```

The expression can use logical comparisons. In this case, the Monday constraint uses: `i<>'Tue' and i<>'Wed'`, which means "when `i` is different from `Tue` and also when `i` is different from `Wed`." The same happens analogously for all the other constraints.

The `==` logical comparison can also be used in expressions.

Listing 5. Solution of this problem in glpsol

```
Problem:    post
Rows:       8
Columns:    7
Non-zeros:  42
Status:     OPTIMAL
Objective:  z = 22.33333333 (MINimum)

   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------ ------------- ------------- -------------
     1 z            B        22.3333
     2 mon          NL            17            17                 0.333333
     3 tue          B             15            13
     4 wed          NL            15            15                 0.333333
     5 thu          NL            19            19                 0.333333
     6 fri          NL            14            14                    < eps
     7 sat          NL            16            16                 0.333333
     8 sun          B        15.6667            11

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------ ------------- ------------- -------------
     1 x[Mon]       B        1.33333             0
     2 x[Tue]       B        5.33333             0
     3 x[Wed]       NL             0             0                    < eps
     4 x[Thu]       B        7.33333             0
     5 x[Fri]       NL             0             0                 0.333333
     6 x[Sat]       B        3.33333             0
     7 x[Sun]       B              5             0

Karush-Kuhn-Tucker optimality conditions:
```

```
KKT.PE: max.abs.err. = 3.55e-15 on row 6
        max.rel.err. = 2.37e-16 on row 6
        High quality

KKT.PB: max.abs.err. = 0.00e+00 on row 0
        max.rel.err. = 0.00e+00 on row 0
        High quality

KKT.DE: max.abs.err. = 5.55e-17 on column 1
        max.rel.err. = 2.78e-17 on column 1
        High quality

KKT.DB: max.abs.err. = 5.55e-17 on row 6
        max.rel.err. = 5.55e-17 on row 6
        High quality

End of output
```

Hey, wait a minute! No one can make 1.33333 employees start working on Monday! Remember the comment earlier about common-sense checks. This is one of them.

GLPK has to consider the decision variables as integer variables. Fortunately, MathProg has a nice way of declaring integer variables. Just change line 15 like so:

```
var x {i in DAYS} >=0, integer;
```

That's pretty simple. The output `glpsol` prints out for an integer problem is a little bit different:

Listing 6. glpsol output for the integer-constrained post office problem

```
Reading model section from post-office-int.mod...
Reading data section from post-office-int.mod...
50 lines were read
Generating z...
Generating mon...
Generating tue...
Generating wed...
Generating thu...
Generating fri...
Generating sat...
Generating sun...
Model has been successfully generated
lpx_simplex: original LP has 8 rows, 7 columns, 42 non-zeros
lpx_simplex: presolved LP has 7 rows, 7 columns, 35 non-zeros
lpx_adv_basis: size of triangular part = 7
      0:   objval =   0.000000000e+00   infeas =   1.000000000e+00 (0)
      7:   objval =   2.600000000e+01   infeas =   0.000000000e+00 (0)
*     7:   objval =   2.600000000e+01   infeas =   0.000000000e+00 (0)
*    10:   objval =   2.233333333e+01   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND
Integer optimization begins...
Objective function is integral
+    10: mip =     not found yet >=              -inf        (1; 0)
+    19: mip =   2.300000000e+01 >=   2.300000000e+01   0.0% (9; 0)
+    19: mip =   2.300000000e+01 >=     tree is empty   0.0% (0; 17)
INTEGER OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.2M (175512 bytes)
lpx_print_mip: writing MIP problem solution to `post-office-int.sol'...
```

Note that the output now shows that an integer optimal solution is found, and that before that happens, GLPK has calculated the optimal solution for the relaxed problem (the problem that does not require the variables to be integer).

Listing 7. The integer solution of the post office problem

```
Problem:    post
Rows:       8
Columns:    7 (7 integer, 0 binary)
Non-zeros:  42
Status:     INTEGER OPTIMAL
Objective:  z = 23 (MINimum) 22.33333333 (LP)

   No.   Row name        Activity     Lower bound   Upper bound
------ ------------    ------------ ------------ ------------
     1 z                          23
     2 mon                        18           17
     3 tue                        13           13
     4 wed                        15           15
     5 thu                        19           19
     6 fri                        14           14
     7 sat                        16           16
     8 sun                        20           11

   No. Column name       Activity     Lower bound   Upper bound
------ ------------    ------------ ------------ ------------
     1 x[Mon]       *             1            0
     2 x[Tue]       *             2            0
     3 x[Wed]       *             3            0
     4 x[Thu]       *             7            0
     5 x[Fri]       *             1            0
     6 x[Sat]       *             3            0
     7 x[Sun]       *             6            0

End of output
```

The first section says that the solution found is integer optimal, and that the value found for the objective function is 23, a minimum. The post office will have to hire 23 full-time employee to satisfy its goals. The optimal value of the relaxed objective function (the one that didn't consider the decision variables as integers) is also printed here.

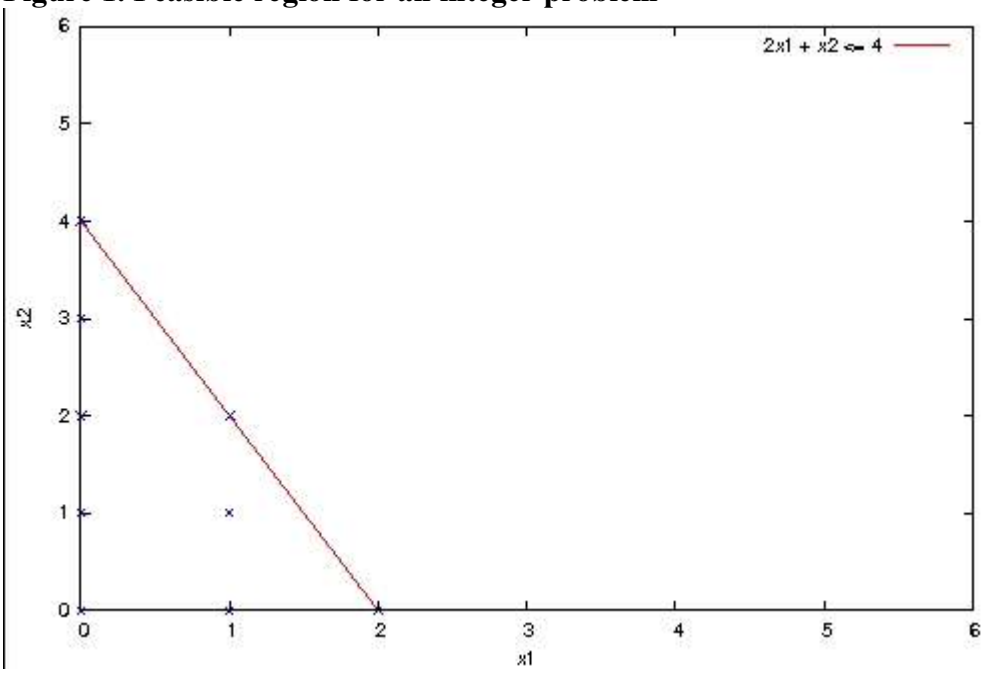
Skip the second section of the report for a second. The third section shows the values of the decision variables. Those values are integers that minimize the problem's objective function.

Now, let's talk about the second section. It shows the activity of the constraints. Some of them are lower bounded, and you probably expect a marginal value or shadow price by now. It doesn't make sense, though, to talk about the marginal value in integer problems. The feasible region of integer problems is not a continuous region. In other words, the feasible region is not a polyhedron; it's composed only of the integer ($x_1, x_2, \ldots, x_n$) pairs inside or at the boundaries of the actual polyhedron of the relaxed problem. This means that the feasible region consists of discrete points in space and, therefore, a relaxation in one of the constraints may or may not yield a better solution inside the new polyhedron.

A little bit of theory

To understand better what is being discussed here, examine this simple feasible region:

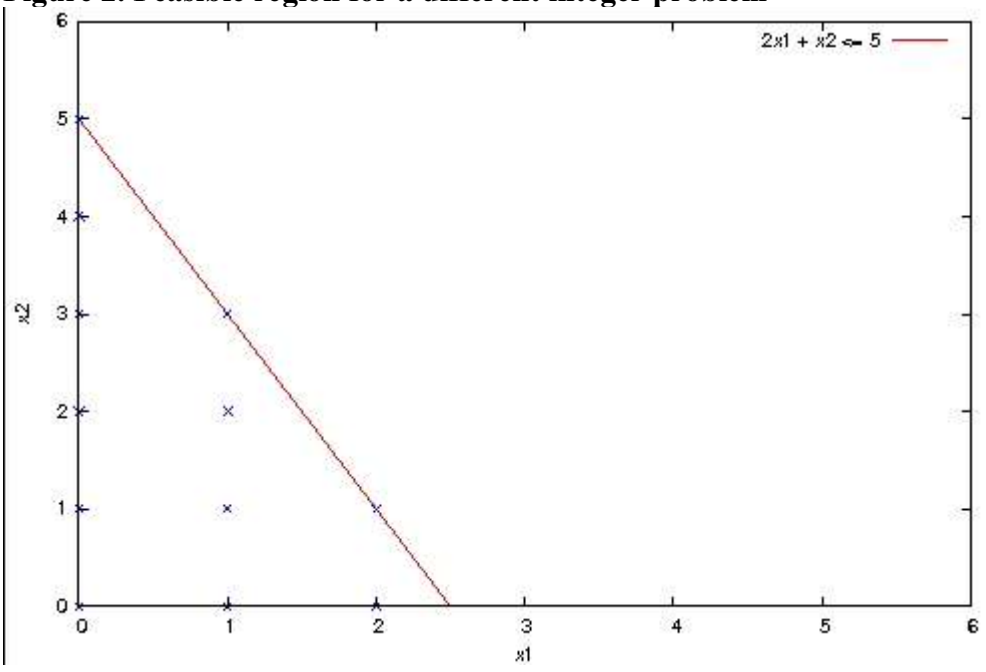
Figure 1. Feasible region for an integer problem



The blue points marked as `x` are integer (x_1, x_2) pairs. This is the integer-feasible region of a x_1 X x_2 two-dimensional universe constrained by x_1 >= 0, x_2 >= 0, and x_1 + x_2 <= 4. If the objective function for this simple case was maximize z = x_1 - x_2, it's clear that the optimal solution would be (2,0), and the constraint would be bounded (because the optimal solution lies on the line ruled by the constraint).

If the constraint was relaxed by one unit, x_1 + x_2 <= 5, the feasible region would now be different.

Figure 2. Feasible region for a different integer problem



The optimal solution would still be the integer point (2,0), though more integer points are feasible now.

Thus, a relaxation of the constraint of an integer problem does not necessarily improve the solution, because the feasible region is discrete and not continuous.

Another question you might ask is: "Is the optimal solution of an integer somehow related to its relaxed problem's optimal solution?" The answer is in the algebra behind the simplex algorithm, but explaining how it works is beyond the scope of this article. Just know that the optimal solution for a non-integer problem is always one of the polyhedron vertices. Always!

In the first feasible region above, the optimal solution is the right vertex of the solution space, which is a triangle made by all the constraints. The objective function increases in the direction of the directional derivative of the objective function. In the simple case, the directional derivative of $x_1 - x_2$ is `(1,-1)`. Therefore, the optimal solution is the farthest point on the polyhedron boundary from the axis' origin using the direction `(1,-1)` as a direction vector. Because the integer solution may lie only on a boundary or within the polyhedron, the best-case scenario is to have an integer solution on a vertex. In this particular case, the optimal solution for both the integer and non-integer problems is the same, but that's not always the case. Why? Because this integer point may not be as far from the origin as the relaxed solution point. For the other non-best-case scenarios, the best integer point would lie within the polyhedron, and the objective function would naturally have worse performance than the one for the relaxed problem, which you might note from the post-office problem.

Remember that this analysis used a simple two-variable maximization problem. The same analysis for a minimization problem looks for the point that minimizes the objective function, which is the one closest to the origin (in the opposite direction of the directional derivative of the objective function).

Conclusion

With the diet problem, you saw how to formulate a simple, multi-variable problem, how to declare bidimensional parameters in GNU MathProg, and how to interpret the results of a minimization problem.

The post office problem introduced MathProg expressions and integer-only decision variables. You saw how to analyze `glpsol` output for the integer problem.

Finally, I discussed feasible region visualization for an integer problem and the way it relates to the objective function for an integer and for the corresponding relaxed problem.

The third and final installment discusses a problem to maximize the profits of a perfume manufacturer, and includes an example that illustrates binary decision variables using a basketball lineup problem.

Resources

Learn

- Get an RSS feed for this GLPK series. (Find out more about RSS feeds of developerWorks content.)
- Read all installments in this GLPK series (developerWorks, August and September, 2006)

- The problems in this article are taken with permission from *Operations Research: Applications and Algorithms, 4th Edition*, by Wayne L. Winston (Thomson, 2004).
- The online documentation for GLPK gives more information about GLPK, how to get the software, and how to join the GLPK community.
- Subscribe to the GLPK help mailing list or bug reports mailing list.
- In the developerWorks Linux zone, find more resources for Linux developers.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Rodrigo Ceron Ferreira de Castro is a Staff Software Engineer at the IBM Linux Technology Center. He graduated from the State University of Campinas (UNICAMP) in 2004. He received the State Engineering Institute prize and the Engineering Council Certification of Honor when he graduated. He's given speeches in open source conferences in Brazil and other countries.