IP Reference guide for integer programming formulations.

by James B. Orlin

for 15.053 and 15.058

This document is intended as a compact (or relatively compact) guide to the formulation of integer programs. For more detailed explanations, see the PowerPoint tutorial on integer programming.

The following are techniques for transforming a problem described logically or in words into an integer program. In most cases, the transformation is the simplest to describe. Unfortunately, simplest is not the same as "best." It is widely accepted that the best integer programming formulations are those that result in fast solutions by integer programming solvers. In general, these are the integer programs for which the linear programming relaxation provides a good bound.

Section 1. Subset selection problems.

Often, models are based on selecting a subset of elements. For example, in the knapsack problem, one wants to select a subset of items to put into a knapsack so as to maximize the value while not going over a specified weight. Or one wants to select a subset of potential products in which to invest. Or, one has a set of different integers, and one wants to select a subset that sums to a value K. In these cases, it is typical for the integer variables to be as follows:

$$x_i = \begin{cases} 1 & \text{if element } i \text{ is selected} \\ 0 & \text{otherwise.} \end{cases}$$

Example: knapsack/capital budgeting. In this example, there are six items to select from.

Item	1	2	3	4	5	6
Cost	5	7	4	3	4	6
Value	16	22	12	8	11	19

Problem: choose items whose cost sums to at most 14 so as to maximize the utility.

Formulation:

$$\begin{array}{ll} \text{maximize} & 16x_1 + 22x_2 + 12x_3 + 8x_4 + 11x_5 + 19x_6 \\ \text{subject to} & 5x_1 + 7x_2 + 4x_3 + 3x_4 + 4x_5 + 6x_6 \le 14 \\ & x_i \in \{0,1\} \quad \text{for } i = 1 \text{ to } 6. \end{array}$$

In general: Maximize the value of the selected items such that the weight is at most *b*.
c_i = value of item *i* for i = 1 to *n*.
a_i = weight of item *i* for i = 1 to *n*.
b = bound on total weight.

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} & \sum_{i=1}^{n} a_i x_i \le b \\ & x_i \in \{0,1\} \quad \text{for } i = 1 \text{ to } n. \end{array}$$

Covering and packing problems.

In some selection problems, each item is associated with a subset of a larger set. The larger set is usually referred to as the *ground set*. For example, suppose that there is a collection of *n* sets S_1, ..., S_n where for i = 1 to n S_i is a subset of the ground set {1, 2, 3, ..., *m*}. Associated with each set S_i is a cost c_i.

$$\text{Let } a_{ij} = \begin{cases} 1 & \text{if } i \in S_j \\ 0 & \text{otherwise.} \end{cases} \qquad \text{Let } x_j = \begin{cases} 1 & \text{if set } S_j \text{ is selected} \\ 0 & \text{otherwise.} \end{cases}$$

The set packing problem is the problem of selecting the maximum cost subcollection of sets, no two of which share a common element. *The set covering problem* is the problem of selecting the minimum cost subcollection of sets, so that each element $i \in \{1, 2, ..., m\}$ is in one of the sets.

$$\begin{aligned} &\text{Maximize} && \textstyle\sum_{j=1}^{n} c_j x_j \\ &\text{subject to} && \textstyle\sum_{j=1}^{n} a_{ij} x_j \le 1 \quad \text{for each } i \in \{1,\dots,m\} && \textit{Set Packing Problem} \\ & && x_j \in \{0,1\} \qquad \text{for each } j \in \{1,\dots,n\}. \end{aligned}$$

$$\begin{aligned} &\text{Minimize} && \textstyle\sum_{j=1}^{n} c_j x_j \\ &\text{subject to} && \textstyle\sum_{j=1}^{n} a_{ij} x_j \ge 1 \quad \text{for each } i \in \{1,\dots,m\} && \textit{Set Covering Problem} \\ & && x_j \in \{0,1\} \qquad \text{for each } j \in \{1,\dots,n\}. \end{aligned}$$

For example, consider the following map of the counties of Colorado.

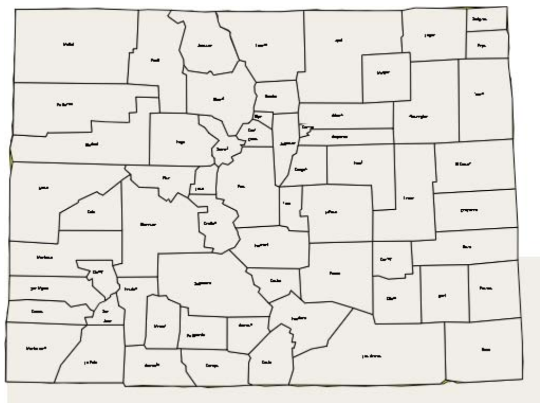


Figure 1. The counties of Colorado.

Suppose that we want to select a minimum cardinality subset C of counties such that each county either was in C or shared a common boundary with a county in C. In this case, we would let

$S_j = \{j\} \cup \{i$: county i shares a boundary with county $j\}$. We let $c_j = 1$ for each j. We would then solve the min cost covering problem.

Here is a related problem. Select the largest subset C' of counties such that no two counties of C' share a common border. We refer to this problem as P1. Although P1 is a set packing problem (as you shall soon see) it is not the set packing problem defined on the sets $S_1, S_2, \ldots, S_n$. To see why not, consider the following example. Suppose that county 1 bordered on county 2, which bordered on county 3, but that counties 1 and 3 had no common border. Then one can include both county 1 and county 3 in C'. But $2 \in S_1 \cap S_3$, and so it would not be possible to select S_1 and S_3 for the set packing problem.

Problem P1 can be formulated as follows:

$$\begin{array}{lll} \text{Maximize} & \sum_{j=1}^{n} x_j & \\ \text{subject to} & x_i + x_j \le 1 & \text{whenever } a_{ij} = 1 \text{ (i.e., } i \in S_j) \qquad \textit{Problem P1} \\ & x_j \in \{0,1\} & \text{for each } j \in \{1,\ldots,n\}. \end{array}$$

Section 2. Modular arithmetic.

In this very brief section, we show how to constrain a variable x to be odd or even, and we show how to constraint x to be a mod(b). (That is, there is an integer q such that $a + qb = x$.) In each case, we need to add a new variable w, where $w \ge 0$ and integer.

Constraint.	**IP Constraint**
x is odd.	$x - 2w = 1$.
x is even.	$x - 2w = 0$.
$x \equiv a \pmod{b}$	$x - bw = a$.

Table 1. Modular arithmetic formulations.

Section 3. Simple logical constraints.

Here we address different logical constraints that can be transformed into integer programming constraints.

In the first set, we describe the logical constraints in terms of selection of items from a subset.

Logical Constraint.	**IP Constraint**
If item *i* is selected, then item *j* is also selected.	$x_i - x_j \le 0$
Either item *i* is selected or item *j* is selected, but not both.	$x_i + x_j = 1$
Item *i* is selected or item *j* is selected or both.	$x_i + x_j \quad 1$
If item *i* is selected, then item *j* is not selected.	$x_i + x_j \le 1$
If item *i* is not selected, then item *j* is not selected.	$-x_i + x_j \le 0$
At most one of items *i*, *j*, and *k* are selected.	$x_i + x_j + x_k \le 1$
At most two of items *i*, *j*, and *k* are selected.	$x_i + x_j + x_k \le 2$
Exactly one of items *i*, *j,* and *k* are selected.	$x_i + x_j + x_k = 1$
At least one of items *i*, *j* and *k* are selected.	$x_i + x_j + x_k \quad 1$

Table 2. Simple constraints involving two or three binary variables.

Restricting a variable to take on one of several values.

Suppose that we wanted to restrict *x* to be one of the elements {4, 8, 13}. This is accomplished as follows.

$$x = 4\,w_1 + 8\,w_2 + 13\,w_3$$
$$w_1 + w_2 + w_3 = 1$$
$$w_i \in \{0, 1\} \text{ for } i = 1 \text{ to } 4.$$

If we wanted to restrict *x* to be one of the elements {0, 4, 8, 13}, it suffices to use the above formulation with the equality constraint changed to "$w_1 + w_2 + w_3 \le 1$."

Section 4. Other logical constraints, and the big M method.

Binary variables that are 1 when a constraint is satisfied.

We next consider binary variables that are defined to be 1 if a constraint is satisfied, and 0 otherwise. In each case, we need bounds on how much the constraint could be violated in a solution that satisfies every other constraint of the problem.

Example 1.

$$w = \begin{cases} 1 & \text{if } x \ge 1 \\ 0 & \text{if } x = 0. \end{cases}$$

In this example, *x* is an integer variable. And suppose that we know *x* is bounded above by 100. (We may know the bound on x because it is one of the constraints of the problem. We may also know the bound of 100 on x because it is implied by one or more of the other constraints. For example, suppose that one of the constraints was "3x + 4y + w ≤ 300." We could infer from this constraint that 3x ≤ 300 and thus x ≤ 100.

Equivalent constraint: $w \le x \le 100w$.
$w \in \{0,1\}$.

In any feasible solution, the definition of w is correct. If x 1, then the first constraint is satisfied whether w = 0 or w = 1, and the second constraint forces w to be 1. If x = 0, then the first constraint forces w to be 0, and the second constraint is satisfied.

Example 2.
$$w = \begin{cases} 1 & \text{if } x \ge 7 \\ 0 & \text{otherwise.} \end{cases}$$

Again, we assume here that x is an integer variable, and that x is bounded above by 100.

Equivalent constraints: $x \ge 7 - 7(1-w)$
$x \le 6 + 94\,w$
$w \in \{0,1\}$.

In any feasible solution, the definition of w is correct. If x 7, then the first constraint is satisfied whether w = 0 or w = 1, and the second constraint forces w to be 1. If $x \le 6$, then the first constraint forces w to be 0, and the second constraint is satisfied.

Big *M*: example 1.
$$w = \begin{cases} 1 & \text{if } x \ge 7 \\ 0 & \text{otherwise.} \end{cases}$$

Here we assume that x is an integer variable that is bounded from above, but we don't specify the bound.

Equivalent constraints: $x \ge 7 - M(1-w)$
$x \le 6 + Mw$
$w \in \{0,1\}$,

where M is chosen sufficiently large. In this way, we don't concern ourselves with the value of the bound. We just write M. In fact, we could have written 7 instead of M in the first constraint, and it would have also been valid. But writing big M means not having to think about the best bound.

The disadvantage of this approach is that the running time of the integer programming algorithm may depend on the choice of M. Choosing M very large (e.g., M = 1 trillion) will lead to valid formulations, but the overly large value of M may slow down the solution procedure.

Big M: example 2.
$$w = \begin{cases} 1 & \text{if } x \ge a \\ 0 & \text{otherwise.} \end{cases}$$

Here we assume that x is an integer variable that is bounded from above, but we don't specify the bound.

Equivalent constraints: $x \ge a - M(1-w)$
$x \le (a-1) + Mw$
$w \in \{0,1\}$,

where M is chosen sufficiently large. In any feasible solution, the definition of w is correct. If x a, then the first constraint is satisfied whether $w = 0$ or $w = 1$, and the second constraint forces w to be 1. If $x \leq a$-1, then the first constraint forces w to be 0, and the second constraint is satisfied.

Big M: example 3.

$$w = \begin{cases} 1 & \text{if } x \leq a \\ 0 & \text{otherwise.} \end{cases}$$

Here we assume that x is an integer variable that is bounded from above, but we don't specify the bound.

Equivalent constraints:

$$\begin{aligned} &x \leq a - M(1\text{-}w) \\ &x \geq (a+1) + M\text{w} \\ &w \in \{0,1\}, \end{aligned}$$

where M is chosen sufficiently large.

In the case that w depends on an inequality constraint involving more than one variable, the previous two transformations can modified in a straightforward manner.

Big M: example 4.

$$w = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} a_i x_i \leq b \\ 0 & \text{otherwise.} \end{cases}$$

Here we assume that $\sum_{i=1}^{n} a_i x_i$ is integer valued and is bounded from above, but we don't specify the bound.

Equivalent constraints:

$$\begin{aligned} &\sum_{i=1}^{n} a_i x_i \leq b + M(1-w). \\ &\sum_{i=1}^{n} a_i x_i \geq b + 1 - Mw. \\ &w \in \{0,1\}. \end{aligned}$$

In any feasible solution, the definition of w is correct. If $\sum_{i=1}^{n} a_i x_i \leq b,$, then the first constraint is satisfied whether $w = 0$ or $w = 1$, and the second constraint forces w to be 1. If $\sum_{i=1}^{n} a_i x_i \geq b+1,$, then the first constraint forces w to be 0, and the second constraint is
satisfied.

At least one of three inequalities is satisfied.

Suppose that we wanted to model the logical constraint that at least one of three inequalities is satisfied. For example,

$x_1 + 4x_2 + 2x_4 \geq 7$ or $3x_1 - 5x_2 \leq 12$ or $2x_2 + x_3 \geq 6$.

We then create three new binary variables w_1, w_2, and w_3 and reformulate the above constraint as the following system of logical, linear, and integer constraints.

If $w_1 = 1$, then $x_1 + 4x_2 + 2x_4 \geq 7$.
If $w_2 = 1$, then $3x_1 - 5x_2 \leq 12$.
If $w_3 = 1$, then $2x_2 + x_3 \geq 6$.
$w_1 + w_2 + w_3 \geq 1$.
$w_i \in \{0,1\}$ for $i = 1$ to 3.

This above system of constraints is equivalent to the following.

$$\begin{aligned}
&x_1 + 4x_2 + 2x_4 && \geq 7 - M(1-w_1)\\
&3x_1 - 5x_2 && \leq 12 + M(1-w_2)\\
&2x_2 + x_3 && \geq 6 - M(1-w_3)\\
&w_1 + w_2 + w_3 && \geq 1. \quad \text{(Using an equality would also be valid.)}\\
&w_i \in \{0,1\} \text{ for } i = 1 \text{ to } 3.
\end{aligned}$$

At least one of two inequalities is satisfied.

In the case of two inequalities, it suffices to create a single new variable w. For example, suppose that we want to satisfy the following logical condition. At least one of the following two constraints is satisfied: $x_1 + 4x_2 + 2x_4 \geq 7$ or $3x_1 - 5x_2 \leq 12$.

This logical condition is equivalent to:

$$\begin{aligned}
&x_1 + 4x_2 + 2x_4 && \geq 7 - Mw\\
&3x_1 - 5x_2 && \leq 12 + M(1-w)\\
&w \in \{0,1\}
\end{aligned}$$

Section 5. Fixed costs

Here we consider an integer program in which there are fixed costs on variables. For example, consider the following integer program:

$$\begin{aligned}
\text{Maximize} \quad & f(x_1) + f_2(x_2) + f_3(x_3)\\
\text{subject to} \quad & 2x_1 + 4x_2 + 5x_3 \leq 100\\
& x_1 + x_2 + x_3 \leq 30\\
& 10x_1 + 5x_2 + 2x_3 \leq 204\\
& x_i \geq 0 \text{ and integer for } i = 1 \text{ to } 3.
\end{aligned}$$

where

$$f_1(x_1) = \begin{cases} 52x_1 - 500 & \text{if } x_1 \geq 1 \\ 0 & \text{if } x_1 = 0 \end{cases}, \quad f_2(x_2) = \begin{cases} 30x_2 - 400 & \text{if } x_2 \geq 1 \\ 0 & \text{if } x_2 = 0 \end{cases}, \quad f_3(x_3) = \begin{cases} 20x_3 - 300 & \text{if } x_3 \geq 1 \\ 0 & \text{if } x_3 = 0 \end{cases}.$$

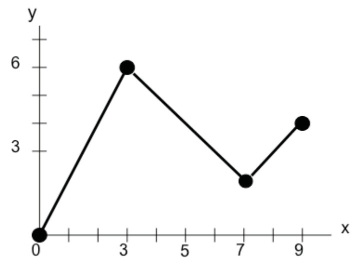
The IP formulation is as follows:

$$\begin{array}{ll} \text{Maximize} & 52x_1 - 500w_1 + 30x_2 - 400w_2 + 20x_3 - 300w_3 \\ \text{subject to} & 2x_1 + 4x_2 + 5x_3 \le 100 \\ & x_1 + x_2 + x_3 \le 30 \\ & 10x_1 + 5x_2 + 2x_3 \le 204 \\ & x_i \le Mw_i \quad \text{for } i = 1 \text{ to } 3 \\ & x_i \ge 0 \text{ and integer for } i = 1 \text{ to } 3. \end{array}$$

The constraint $x_i \le Mw_i$ forces $w_i = 1$ whenever $x_i > 0$. The model may look incorrect because it permits the possibility that $x_i = 0$ and $w_i = 1$. It is true that the IP model allows more feasible solutions than it should. However, if $x_i = 0$ in an optimal solution, then $w_i = 0$ as well because its objective value coefficient is less than 0. Because the integer program gives optimal solutions, if $x_i = 0$, then $w_i = 0$.

Section 6. Piecewise linear functions.

Integer programming can be used to model functions that are piecewise linear. For example, consider the following function.



$$y = \begin{cases} 2x & \text{if } 0 \le x \le 3 \\ 9 - x & \text{if } 4 \le x \le 7 \\ -5 + x & \text{if } 8 \le x \le 9. \end{cases}$$

One can model y in several different ways. Here is one of them. We first define two new variables for every piece of the curve.

$$w_1 = \begin{cases} 1 & \text{if } 0 \le x \le 3 \\ 0 & \text{otherwise.} \end{cases} \qquad x_1 = \begin{cases} x & \text{if } 0 \le x \le 3 \\ 0 & \text{otherwise.} \end{cases}$$

$$w_2 = \begin{cases} 1 & \text{if } 4 \le x \le 7 \\ 0 & \text{otherwise.} \end{cases} \qquad x_2 = \begin{cases} x & \text{if } 4 \le x \le 7 \\ 0 & \text{otherwise.} \end{cases}$$

$$w_3 = \begin{cases} 1 & \text{if } 8 \le x \le 9 \\ 0 & \text{otherwise.} \end{cases} \qquad x_3 = \begin{cases} x & \text{if } 8 \le x \le 9 \\ 0 & \text{otherwise.} \end{cases}$$

We complete the model as follows.

IP formulation

$y = 2x_1 + 9w_2 - x_2 - 5w_3 + x_3$
$0 \le x_1 \le 3w_1$
$4w_2 \le x_2 \le 7w_2$
$8w_3 \le x_3 \le 9w_3$
$w_1 + w_2 + w_3 = 1$
$x = x_1 + x_2 + x_3$
$w_i \in \{0, 1\}$ for $i = 1$ to 3.

For any choice of w_1, w_2, and w_3, the variables x and y are correctly defined.

Section 7. The traveling salesman problem

In this section, we give the standard model for the traveling salesman problem. It has an exponential number of constraints, which may seem quite unusual for an integer programming model. We explain how it can be implemented so as to be practical.

We assume that there are n cities, and that c_{ij} denotes the distance from city i to city j.

In this model, the following are the variables:

$$x_{ij} = \begin{cases} 1 & \text{if city } i \text{ is immediately followed by city } j \text{ on the tour} \\ 0 & \text{otherwise.} \end{cases}$$

The formulation is as follows:

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & \\ \text{subject to} & \sum_{j=1}^{n} x_{ij} = 1 & \text{for all } i = 1 \text{ to } n \\ & \sum_{i=1}^{n} x_{ij} = 1 & \text{for all j} = 1 \text{ to } n \\ & \sum_{i\in S, j\in N\setminus S} x_{ij} \ge 1 & \text{for all } S \subset \{1,2,\ldots,n\} \text{ with } 1 \le |S| \le n-1 \\ & x_{ij} \in \{0,1\}, & \text{for all } i, j = 1 \text{ to } n. \end{array}$$

The first set of constraints ensures that there is some city that follows city i in the solution. The second set of constraints ensures that there is some city that precedes city j in the solution.

Unfortunately, there are not enough constraints to ensure that the solution is a tour. For example, a feasible solution to the first two sets of constraints for the six city problem consists of $x_{12} = x_{23} = x_{31} = 1$, and $x_{45} = x_{56} = x_{64} = 1$. This "solution" to the first two sets of constraints corresponds to the cycles 1-2-3-1 and 4-5-6-4. A tour should be one cycle only.

The third set of constraints guarantees the following. For any non-empty subset S of cities with $|S| < n$, there must be some city not in S that follows some city that is in S. This set of constraints is known as the *subtour elimination constraints*.

With all of the constraints listed above, every feasible solution corresponds to a tour.

Implementation details.

Suppose that one wanted to solve the linear programming relaxation of the TSP; that is, we solve the problem obtained from the above constraints if we drop the integrality constraints, and merely require that x_{ij} 0 for each i and j. (We won't deal with solving the integer program here.) We refer to this linear program as LP*.

At first it appears that there is no way of solving LP* for any large values of n, say $n > 100$. For any TSP instance with more than 100 cities, there are more than 2^{100} different subtour elimination constraints. Listing all of the constraints would take more than all of the computer memory in the entire world.

Instead, LP* is solved using a "constraint generation approach." LP(0) is obtained from LP* by dropping all of the subtour elimination constraints. An optimal solution x_0 is obtained for LP(0). If x_0 is feasible for LP*, then it is also optimal for LP*. If not, then some subtour elimination constraint that x_0 violates is identified and added to LP(0), resulting in LP(1). It's not obvious how one can determine a subtour elimination constraint that is violated, but there are efficient approaches for finding a violated subtour elimination constraint.

An optimal solution x_1 is obtained for LP(1). If x_1 is feasible for LP*, then it is also optimal for LP*. If not, then some subtour elimination constraint that x_1 violates is identified and added to LP(1), resulting in LP(2).

This process continues until there is an optimal solution for LP(k) for some k that is feasible for LP* and hence optimal for LP*. Or else, the solver takes so much time, that it is stopped before reaching optimality. In practice, the technique works quite well, and finds an optimal solution for LP* in a short amount of time.