Financial Optimization
ISE 347/447

Lecture 13

Dr. Ted Ralphs

Reading for This Lecture

- C&T Chapter 11

Integer Linear Optimization

- An *integer linear optimization problem* (ILP) is the same as a linear optimization problem except that the variables can take on only integer values.
- If only some of the variables are constrained to take on integer values, then we call the program a *mixed integer linear optimization problem* (MILP).
- The general form of an MILP is

$$
\begin{aligned}
min \quad & c^\top x + d^\top y \\
s.t. \quad & Ax + By = b \\
& x, y \geq 0 \\
& x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}
\end{aligned}
$$

Mixed Integer Nonlinear Optimization Problem

- A *mixed integer nonlinear optimization problem* (MINLP) is the same as a nonlinear optimization problem except that the variables can take on only integer values.
- The general form of a MINLP is

$$\begin{aligned} \min\ & f(x) \\ \text{s.t.}\ & g(x) \le 0 \\ & h(x) = 0 \\ & x \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \end{aligned}$$

Modeling with Integer Variables

- Why do we need integer variables?
- If the variable is associated with a physical entity that is indivisible, then it must be integer.
  - Shares of a stock.
  - Investments that can only be made in fixed amounts.
- *0-1 (binary) variables* can be used to model logical conditions or combinatorial structure.
  - Modeling yes/no decisions.
  - Enforcing disjunctions.
  - Enforcing logical constraints.
  - Modeling fixed costs.
  - Modeling piecewise linear functions.

Conjunction versus Disjunction

- A more general mathematical view that ties integer programming to logic is to think of integer variables as expressing *disjunction*.
- The constraints of a standard mathematical program are *conjunctive*.
  - All constraints must be satisfied.
  - In terms of logic, we have

$$g_1(x) \leq b_1 \text{ AND } g_2(x) \leq b_2 \text{ AND } \cdots \text{ AND } g_m(x) \leq b_m \tag{1}$$

  - This corresponds to *intersection* of the regions associated with each constraint.
- Integer variables introduce the possibility to model *disjunction*.
  - At least one constraint must be satisfied.
  - In terms of logic, we have

$$g_1(x) \leq b_1 \text{ OR } g_2(x) \leq b_2 \text{ OR } \cdots \text{ OR } g_m(x) \leq b_m \tag{2}$$

  - This corresponds to *union* of the regions associated with each constraint.

How Hard is Integer Programming?

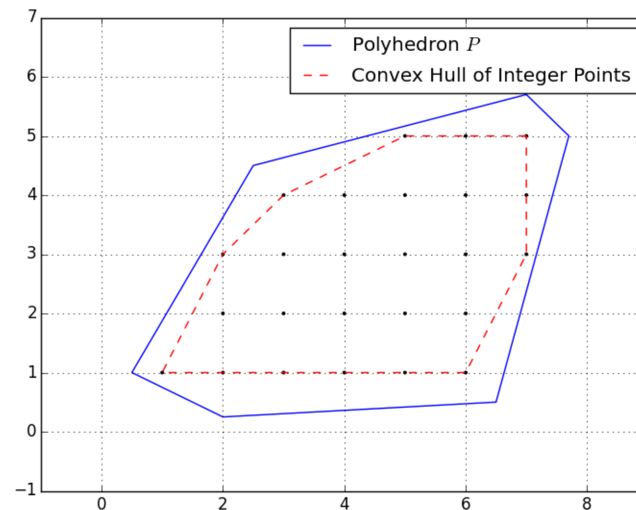
- Solving general integer programs can be much more difficult than solving linear programs.
- There is no known *polynomial-time* algorithm for solving general MILPs.
- Solving the associated *linear programming relaxation* provides a lower bound on the optimal solution value of a given MILP.
- In general, an optimal solution to the LP relaxation may not tell us much about an optimal solution to the MILP.
  - Rounding to a feasible integer solution may be difficult.
  - The optimal solution to the LP relaxation can be arbitrarily far away from the optimal solution to the MILP.
  - Rounding may result in a solution far from optimal.
  - We can sometimes bound the difference between the optimal solution to the LP and the optimal solution to the MILP (how?).

The Geometry of Integer Programming

- Let's consider again an integer linear program

$$\begin{aligned} min \quad & c^\top x \\ s.t. \quad & Ax = b \\ & x \geq 0 \\ & x \text{ integer} \end{aligned}$$

- The feasible region is the integer points inside a polyhedron.



- It is not difficult to see why solving the LP relaxation does not necessarily yield a solution near an integer optimum.

Easy Integer Programs

- Certain integer programs are "easy".
- What makes an integer program "easy"?
  - All of the extreme points of the LP relaxation are integral.
  - Every square submatrix of A has determinant $+1$, -1, or 0.
  - We know a *complete description* of the convex hull of feasible solutions.
  - We have an efficient algorithm for finding an optimal integer solution (not based on linear programming).
  - There is no duality gap (more on this later).
- Examples of "easy" integer programs.
  - Minimum cost network flow problem.
  - Maximum flow problem.
  - Assignment problem.

Modeling Binary Choice

- We use binary variables to model yes/no decisions.
- Example: Integer knapsack problem
  - We are given a set of items with associated values and weights.
  - We wish to select a subset of maximum value such that the total weight is less than a constant K.
  - We associate a 0-1 variable with each item indicating whether it is selected or not.

$$
\begin{aligned}
max \ & \sum_{j=1}^{m} c_j x_j \\
s.t. \ & \sum_{j=1}^{m} w_j x_j \le K \\
& x \ge 0 \\
& x \quad integer
\end{aligned}
$$

- Knapsack problems arise as subproblems in many financial applications.

Modeling Dependent Decisions

- We can also use binary variables to enforce the condition that a certain action can only be taken if some other action is also taken.
- Suppose x and y are variables representing whether or not to take certain actions.
- The constraint $x \leq y$ says "only take action x if action y is also taken".

Example: Portfolio Optimization

- Consider a portfolio optimization problem and suppose we want to avoid positions that are "too small."
- As before, let x_i be the size of the investment in asset i.
- As a first ideas, we could impose a constraint that says something like $x_i > 0 \Rightarrow x_i \geq l_i$.
- Possible implementations
  - Require investments in asset i to be multiples of l_i (by scaling variable x_i and requiring it to be integer).
  - Add a binary variable y_i that takes value 1 if the asset is purchased and 0 otherwise and use it enforce the constraint.
  - Use a branching disjunction (more on this later).

Variable Upper and Lower Bounds

- Variable bounds are bounds whose value is either 0 or some other constant, depending on the value of an associated binary variable.
- To impose a variable upper bound on variable x_i, we add an associated a binary variable y_i and the constraint

$$x_i \leq y_i u_i$$

- This constraint (along with nonnegativity) means that x_i must either take value 0 or have an upper bound of u_i.
- We can have both upper and lower bounds variable with the constraint

$$y_i l_i \leq x_i \leq y_i u_i$$

- We could use variable bounds to impose the minimum transaction level constraint.

Modeling Disjunctive Constraints

- More generally, we may be given two constraints $a^\top x \geq b$ and $c^\top x \geq d$ with nonnegative coefficients.
- We want to impose that at least one of the two constraints to be satisfied.
- To model this, we define a binary variable y and impose

$$\begin{aligned} a^\top x &\geq yb, \\ c^\top x &\geq (1-y)d, \\ y &\in \{0,1\}. \end{aligned}$$

- Further generalizing, we can impose that at least k out of m constraints be satisfied with

$$\begin{aligned} (a_i)^\top x \geq b_i y_i, \quad i \in [1..m] \\ \sum_{i=1}^{m} y_i \geq k, \\ y_i \in \{0,1\} \end{aligned}$$

Cardinality Constraints

- Another approach to ensuring that a portfolio is not composed of many small positions is to impose an upper bound of K on the number of positions.
- This can be done using the same aforementioned indicator variables along with a constraint of the form

$$\sum_{i=1}^{n} y_i \leq K$$

- Alternatively, this constraint could also be imposed using branching disjunctions without the indicator variables (more on this later).

Example: Simple Marwowitz Portfolio Model

```
model.assets = Set()
model.T = Set(initialize = range(1994, 2014))
model.max_risk = Param(initialize = .00305)
model.R = Param(model.T, model.assets)
def mean_init(model, j):
    return sum(model.R[i, j] for i in model.T)/len(model.T)
model.mean = Param(model.assets, initialize = mean_init)
def Cov_init(model, i, j):
    return sum((model.R[k, i] - model.mean[i])*(model.R[k, j] - model.mean[j])
            for k in model.T)
model.Cov = Param(model.assets, model.assets, initialize = Cov_init)
model.alloc = Var(model.assets, within=NonNegativeReals)
def risk_bound_rule(model):
    return (sum(sum(model.Cov[i, j] * model.alloc[i] * model.alloc[j]
                 for i in model.assets) for j in model.assets)
                 <= model.max_risk)
model.risk_bound = Constraint(rule=risk_bound_rule)
def tot_mass_rule(model):
    return (sum(model.alloc[j] for j in model.assets) == 1)
model.tot_mass = Constraint(rule=tot_mass_rule)
def objective_rule(model):
    return sum(model.alloc[j]*model.mean[j] for j in model.assets)
model.objective = Objective(sense=maximize, rule=objective_rule)
```

Example: Adding Cardinality Constraints

```
model.K = Param()
model.buy = Var(model.assets, within=NonNegativeIntegers)
def selection_rule(model, i):
    return (model.alloc[i] <= model.buy[i])
model.selection = Constraint(model.assets, rule=selection_rule)

def cardinality_rule(model):
    return (summation(model.buy) == model.K)
model.cardinality = Constraint(rule=cardinality_rule)
```

Example: Capital Budgeting

- Suppose we have \$4 million to invest in projects over the next three years.
- Each project has an associated cost and profit (in present value dollars) as follows:

	Year 1		Year 2		Year 3	
Project	Cost	Profit	Cost	Profit	Cost	Profit
1	0	0	0	0	0	0
2	1	2	1	3	1	2
3	2	4	3	9	2	5
4	4	10	-	-	-	-

Modeling a Restricted Set of Values

- Note that in each year, our decision is really just how much to invest in that year.
- One approach is therefore to have a single variable for each year and to restrict the value to be equal to one of the possible investment levels.
- More generally, we may want variable x to only take on values in the set $\{a_1, \ldots, a_m\}$.
- We introduce m binary variables $y_j, j = 1, \ldots, m$ and the constraints

$$x = \sum_{j=1}^{m} a_j y_j,$$

$$\sum_{j=1}^{m} y_j = 1,$$

$$y_j \in \{0, 1\}$$

- In fact, in this case, we don't actually need the variable x.

Set Partitioning, Packing, and Covering Problems

- Constraints of the form $\sum_{j \in T} x_j = 1$ can be used to enforce that exactly one item should be chosen from a set T.
- Similarly, we can also require that at most one or at least one item should be chosen.
- Example: Set partitioning problem
  - A set partitioning problem is any problem of the form

$$\begin{aligned} & min\, c^\top x \\ & s.t. \;\; Ax = 1 \\ & \qquad x_j \in \{0,1\}\, \forall j \end{aligned}$$

    where A is a 0-1 matrix.
  - Each row of A represents an item from a set S.
  - Each column A_j represents a subset S_j of S.
  - Each variable x_j represents selecting subset S_j.
  - The constraints say that $\cup_{\{j | x_j = 1\}} S_j = S$.
  - In other words, each item must appear in at least one selected subset.

Example: Combinatorial Auctions

- The winner determination problem for a *combinatorial auction* is a set packing problem.
- The rows represent items or services that a buyer is trying to acquire.
- The columns represent subsets of the items that a particular supplier can provide for a specified cost.
- The object is to select a subset of the bidders such that
  - cost is minimized, and
  - every item is provided by at least one bidder.
- This is a set covering problem.

Piecewise Linear Cost Functions

- We can use binary variables to model arbitrary piecewise linear cost functions.
- We could use such a model to solve a version of the capital budgeting problem in which we are allowed to invest in multiple projects, in whole or in part.
- The function is specified by ordered pairs $(a_i, f(a_i))$ and we wish to evaluate it at a point x.
- We have a binary variable y_i, which indicates whether $a_i \le x \le a_{i+1}$.
- To evaluate the function, we will take linear combinations $\sum_{i=1}^{k} \lambda_i f(a_i)$ of the given functions values.
- This works only if the only two nonzero $\lambda_i's$ are the ones corresponding to the endpoints of the interval in which x lies.

Minimizing Piecewise Linear Cost Functions

- The following formulation minimizes the function.

$$
\begin{aligned}
min \sum_{i=1}^{k} \lambda_i f(a_i) & \\
s.t. \ \sum_{i=1}^{k} \lambda_i = 1, & \\
\lambda_1 \leq y_1, & \\
\lambda_i \leq y_{i-1} + y_i, & \quad i \in [2..k-1], \\
\lambda_k \leq y_{k-1}, & \\
\sum_{i=1}^{k-1} y_i = 1, & \\
\lambda_i \geq 0, & \\
y_i \in \{0, 1\}. &
\end{aligned}
$$

- The key is that if $y_j = 1$, then $\lambda_i = 0, \ \forall i \neq j, j+1$.

Fixed-charge Problems

- In many instances, there is a fixed cost and a variable cost associated with a particular decision.
- For example, there might be a fixed cost to certain financial transactions, regardless of the amount transacted.
- Consider the problem of converting B units of currency 1 into currency N through a sequence of intermediate transactions in currencies 2 through $N-1$.
  - To convert current i into a set of other currencies, there is a fixed cost of c_i (in terms of currency N).
  - There is also an associated exchange rate r_{ij}.
  - There is a cap u_i on the total amount of currency i that can be converted.
  - The goal is to end up with as much of currency N as possible.

Modeling the Currency Exchange Problem

- The decision to be made is how much of each currency to exchange for each other currency. So variables in this case are

$$y_i = \text{whether any of currency } i \text{ is exchanged for other currencies}$$
$$x_{ij} = \text{amount of currency } i \text{ exchanged for currency } j$$

- Note that the amount of currency j we end up with after exchanging from i is $r_{ij}x_{ij}$.

- Ultimately, we want to end up with as much of currency N as possible, so our objective function is the amount of all other currencies exchanged into currency N:

$$\max \sum_{i=1}^{N-1} r_{iN}x_{iN} - \sum_{i=1}^{n} c_i y_i.$$

Modeling the Currency Exchange Problem (cont.)

- For notational convenience, we assume that $x_{ii} = 0 \; \forall i \in [1..N]$.
- For every currency $j \neq 1$, the amount available for exchange is $\sum_{i=1}^{N-1} r_{ij} x_{ij}$ and the amount actually exchanged is $\sum_{j=2}^{N} x_{ij}$.
- The constraints are then

$$
\begin{aligned}
\sum_{j=2}^{N} x_{ij} &\leq y_i u_i, \quad \forall\, i \in [1..N], \\
\sum_{i=1}^{N-1} r_{ij} x_{ij} &\geq \sum_{k=2}^{N} x_{jk}, \forall\, j \in [2..N-1], \\
\sum_{j=2}^{N} x_{1j} &\leq B, \text{ and} \\
x_{ij} &\geq 0, \quad \forall\, i \in [1..N-1], j \in [2..N]. \\
y_i &\in \{0,1\}, \forall\, i \in [1..N-1]
\end{aligned}
$$

Modeling the Currency Exchange Problem (cont.)

This gives us a integer programming formulation that looks like

$$
\begin{aligned}
\max \sum_{i=1}^{N} r_{iN} x_{iN} - c_i y_i & & \\
s.t. \qquad \sum_{j=1}^{N} x_{ij} &\le y_i u_i, & \forall\, i \in [1..N], \\
\sum_{i=1}^{N} r_{ij} x_{ij} &\le \sum_{k=1}^{N} x_{jk}, & \forall\, j \in [2..N-1], \\
\sum_{j=1}^{N} x_{1j} &\le B, & \\
x_{ij} &\ge 0, & \forall\, i \in [1..N-1], j \in [2..N], \\
y_i &\in \{0,1\}, & \forall\, i \in [1..N-1].
\end{aligned}
$$

Distinguishing "Formulations" and "Models"

- The modeling process consists generally of the following steps.
  - Determine the "real-world" state variables, system constraints, and goal(s) or objective(s) for operating the system.
  - Translate these variables and constraints into the form of a mathematical optimization problem (the "formulation").
  - Solve the mathematical optimization problem.
  - Interpret the solution in terms of the real-world system.
- This process presents many challenges.
  - Simplifications may be required in order to ensure the eventual mathematical program is "tractable".
  - The mappings from the real-world system to the model and back are sometimes not very obvious.
  - There may be more than one valid "formulation".
- All in all, an intimate knowledge of mathematical optimization definitely helps during the modeling process.

The Importance of Fomulation

- Different formulations for the same problem can result in dramatically different in terms of tractability.
- Simple example: two ways of modeling binary variables x.
  - $x \in \{0, 1\}$
  - $x = x^2$
- The first formulation is integer linear, while the second formulation is nonlinear continuous.
- These would be solved with two entirely different classes of algorithms.
- As a rulse of thumb, the first formulation is preferred.