Linear and Integer LinearProgramming

Enrico Malaguti

DEIS - University of Bologna

ECI 2012

Outline

1. Optimization Problems
2. Linear Programming
    - Complexity
    - Algorithms for LP
3. Integer Linear Programming
4. Strong formulations for ILP
    - Separation
    - Column Generation
5. MIP solvers

Optimization Problems

Optimization Problems

- $x = (x_1, \ldots, x_n) \in R^n$: vector of decision variables
- $F \in R^n$: set of feasible solutions
- $\phi : R^n \rightarrow R$: objective function

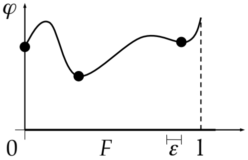
Optimization Problem $[P]$

$$\min \phi(x)$$
$$x \in F$$

Find $x^* \in F$ (global optimum) such that: $\phi(x^*) \leq \phi(x), \ \forall x \in F$

Optimization Problems

- in general, ϕ and F can be whatever (non-continuos, non-differentiable, etc.)
- x^* may not exists ($F = \emptyset$) or may not be unique
- there might be local and global minima



In general, an algorithm has only a "local" view of ϕ and F

Optimization Problems

y is a local optimum if it exists a neighborhood $N \subseteq F$:
$\phi(y) \leq \phi(x), \ \forall x \in N$

e.g. $N_\epsilon(y) = \{x \in F : ||y - x|| \leq \epsilon, \epsilon > 0\}$

- $[P]$ requires to find at least one global optimum
- We call a *feasible* solution any $z \in F$
- a heuristic algorithm produces feasible solutions z of "good" quality, the quality of z is given by $\phi(z)$

Problems classification

Problems are classified depending on ϕ and F.

- Generic ϕ and F: Non Linear Programming (NLP). Great capacity of modeling real-world, we do not know algorithms for general problems. Algorithms for specific problems converge to local optima.
- ϕ and F convex: Convex Programming. We know algorithms for general problems, we can compute global optima.

Convex Programming

- F is a convex *set* if: $\forall x, y \in F,\ 0 \leq \theta \leq 1,\ z = \theta x + (1-\theta)y$, we have $z \in F$.
- $\phi : R^n \rightarrow R$ is a convex *function* if: $dom(\phi)$ is a convex set, $\forall x, y \in dom(\phi),\ \phi(\theta x + (1-\theta)y) \leq \theta\phi(x) + (1-\theta)\phi(y)$.

Theorem - If $[P]$ is a Convex Programming Problem, any local optimum is a global optimum.

Linear Programming

Linear Programming

- $\phi : R^n \rightarrow R$ is a linear function: $\forall x, y \in dom(\phi),\ \ a, b \in R$, we have $\phi(ax + by) = a\phi(x) + b\phi(y)$.
- $F \in R^n$ is defined by a set of *linear inequalities*: $g_i(x) \geq 0,\ \ i = 1, \dots, m$ (note that a linear inequality $g_i(x) \geq 0$ defines a half-space in R^n, whose support is the hyper-plane $g_i(x) = 0$).

Linear programming is a special case of convex programming.

Notation: $g_i(x) = a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n - b_i$

LP in *Standard Form*:

$$\begin{aligned} \min c'x \\ Ax = b \\ x \geq 0 \end{aligned}$$

where $A \in R^{m \times n},\ \ c, x \in R^n,\ \ b \in R^m$.

LP: assumptions

$$\min\; z = \ldots + c_h x_h + \ldots$$
$$\ldots + a_{ih} x_h + \ldots \leq b_i$$

1. **Proportionality**. The effect of the h-th resource is proportional to its amount x_h:
2. **Additivity**. There is no interaction among the resources in the objective function and constraints.
3. **Divisibility**. Decision variables can get *non integer* values.
4. **Determinism**. All the model parameters are given constants, while, in real-world, parameters are affected by incertitude.
    - Sensitivity analysis
    - Stochastic Optimization
    - Robust Optimization

Computational Complexity

Complexity of an *algorithm*: denotes the number of *elementary operations* needed to run an algorithm.

- always referred to the size of the input (number of *bits* needed to code the input);
- always referred to the worst case;
- measured in term of order of magnitude through the notation $O(.)$.

Polynomial Algorithm: its running time is bounded above ($O(.)$) by a polynomial expression in the size of the input;
Exponential Algorithm: its running time is bounded below ($\Omega(.)$) by an exponential expression in the size of the input;

Computational Complexity

Complexity of a *problem* (decision):

- *Class P - polynomial*. Problems for which a polynomial time algorithm exists belong to the complexity class P.
- *Class NP - Non deterministic polynomial* - Intuitively, NP is the set of all decision problems for which the instances where the answer is "yes" have efficiently verifiable proofs of the fact that the answer is indeed "yes". More precisely, these proofs have to be verifiable in polynomial time by a deterministic Turing machine.
- Class of *NP complete problems*. A problem $[P]$ is NP-complete if any problem in NP can be transformed to $[P]$ in polynomial time.
- P = NP??

P=NP?

The Simpsons, *Treehouse of Horror VI*, serie 7, episode 6.

Algorithms for LP

Theorem - LP is polynomially solvable.

- Simplex Algorithm, non-polynomial, excellent practical behavior;
- Ellipsoid Algorithm, used to prove that LP is polynomial, useless in practice;
- Interior Point Methods, polynomial, can solve convex problems in general;

Simplex Algorithm (G. Dantzig, 1947)

Consider a generic Linear Program LP. We have that:

- Linear Programming is convex $\Rightarrow$ any local optimum is a global optimum;
- The feasible region F is a polyhedron (intersection of half-spaces);
- (*Theorem*) - If F has at least one vertex, then either LP is unbounded or there is an optimal solutions which is a vertex of the polyhedron F;

So we have a problem (LP) with infinitely many solutions but, to solve it, it is enough to check a finite number of vertices, and to find a local optimum (which is global).

Algorithm (geometric): start from a vertex of the polyhedron and iteratively select an improving (w.r.t. the objective function) adjacent vertex. When no adjacent vertex improves on the current one, this is optimum.

Simplex Algorithm - Algebraic viewpoint

Consider a *standard form* LP ($\min c'x,\ Ax = b,\ x \geq 0$) with $A \in R^{m \times n}$ and $x \in R^n$ ($m + n$ constraints including non-negativity). It can be shown that:

- A vertex of a polyhedron corresponds to a *feasible* solution of the LP where n constraints are tight ($=$). We have m tight constraints $Ax = b$, so we need $n - m$ tight constraints of the form $x_f = 0$ to define a vertex. The m variables x_b not forced at 0 are called *basic variables* (w.l.o.g., let be basic the first m).
- The m columns of A corresponding to the x_B variables are called a *base* B: $A = [B|F]$.
- Any vertex corresponds to a solution $x = [x_B|x_F]$, where $x_F = [0, \ldots, 0]$. The value of the *basic variables* is $x_B = B^{-1}b$.
- Two adjacent vertices x and y differ for 1 column in the bases: $B = [A_1, \ldots, \mathbf{A_l}, \ldots, A_m]$; $B' = [A_1, \ldots, A_{l-1}, A_{l+1}, \ldots, A_m, \mathbf{A_j}]$

Simplex Algorithm - Algebraic viewpoint

When moving from a feasible solution x towards a feasible direction d of a quantity $\theta > 0$, we get to $x + \theta d$. For feasibility, it must be $Ad = 0$.

Now consider to move from a vertex x to an *adjacent* vertex y. We have that:

- all the non basic variables remain at 0, with the exception of x_j (which enters the basis), so $d_i = 0,\ i \notin B,\ d_j = 1$;
- the basic variables x_B change to $x_B + \theta d_B$:

$$0 = Ad = Bd_b + A_j d_j = Bd_B + A_j$$

  i.e., $d_B = -B^{-1}A_j$.

The cost change along direction d is $c'd = c_j + c'_B d_B = c_j - c'_B B^{-1} A_j$. This coefficient is called reduce cost $\tilde{c}_j$ of the variable x_j (and has value 0 for basic variables).

Simplex Algorithm - Algebraic viewpoint

Starting from a feasible solution x with associated base B_x, the Simplex Algorithm moves to an *adjacent* solution y with associated base B_y such that $c'y < c'x$ (more feasible basis can correspond to the same solution in case of degeneracy). This happens when $\exists j: \ \tilde{c}_j < 0$ ($j \notin B_x$, $\tilde{c}_i = 0$, $i \in B_x$)

Of course, optimality of a solution x means that there is no variable j with *negative* reduced cost c_j.

In the worst case, the Simplex Algorithm may visit all the exponentially many vertices of the polyhedron before getting to the optimal one, thus it is *not* a polynomial algorithm.

interior point methods

Class of algorithms for the solution of convex problems (including LP).

- *Idea:* consider a modified objective function ϕ', including a penalty which grows when the solution is close the the boundary of the feasible set F. Let $\delta(x)$ denote the distance of x from the boundary of F.
$$\phi'(x) = \phi(x) + \mu \ln \delta(x)$$
- The algorithm starts from a solution (point) x in the interior of F. At each iteration, the algorithm moves along a direction of maximum decrease of $\phi'(x)$ (using Newton methods), and a new solution within F is obtained. The algorithm converges to an optimal solution on the boundary of F.

Simplex VS Interior Point Methods

- Interior Point Methods are on average faster in solving LPs from scratch;
- The Simplex Algorithm allows a "warm start", so one can:
  - modify the objective function coefficients;
  - modify the right-hand-side;
  - add variables;
  - add constraints;

  and re-optimize by starting from a previously computed solution.
- The Simplex Algorithm provides several useful information in addition to the optimal solution, like reduced costs, dual variables, etc.

Integer Linear Programming

Integer Linear Programming - ILP

- $\phi : R^n \to R$ is a linear function.
- the feasible region F is the intersection of a polyhedron $P \in R^n$ defined by a set of *linear inequalities*: $P = \{x \mid g_i(x) \geq 0,\ i = 1, \dots, m\}$ and the set of integer numbers $\mathbb{Z}$.

$$\min c'x$$
$$Ax \geq b$$
$$x \in \mathbb{Z}^n$$

We talk about Mixed-Integer Linear Programming (MILP) when only a subset of the variables are integer.

Theorem - Integer Linear Programming is NP-complete.

ILP - Lower and Upper Bound

Relaxation of problem $[P]$: problem obtained by "relaxing" some of the requirements of $[P]$. Relaxed problems are in general easier to solve (e.g., original problem is NP-complete, relaxed problem is polynomial).

Continuous Relaxation: obtained by relaxing the integrality requirement for the variables, provides a lower bound on the optimal solution value of $[P]$.

Minimization Problem

- upper bound (integer feasible solution)

- optimal solution (integer optimal)

- lower bound (e.g., optimal solution of the continuous relaxation)

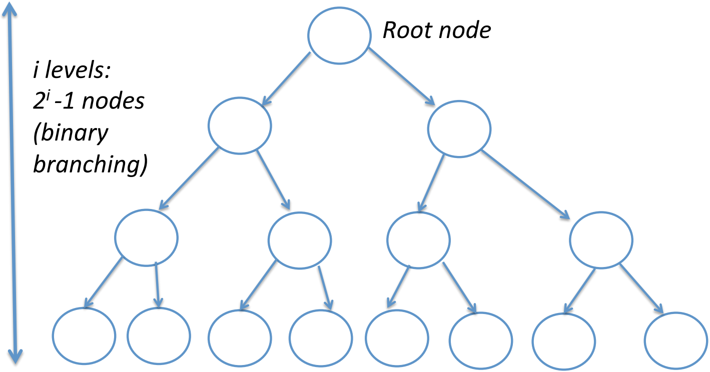
Branch-and-Bound

Branch-and-Bound is a *general method* for solving optimization problems with a finite number of solutions, which can be applied to combinatorial problems and ILP. It consists in the implicit enumeration of all the possible solutions to the problem.
Branch-and-Bound has two main components:

1. A splitting procedure (branching) that, given a set of candidate solutions S, returns two or more subsets S_1 and S_2 such that $S = S_1 \cup S_2$. Each set S_j is associated with a problem $[P_j]$, that is, $[P]$ solved on S_j.
2. An evaluating procedure (bounding) which relaxes the problem and computes a lower bound on any $[P_j]$, that is, a bound on the optimal value of the objective function ϕ on S_j.

The original problem $[P]$ is iteratively split into smaller problems, and the search is represented through a decision tree.

Branch-and-Bound



Root node

i levels:
2^i -1 nodes
(binary
branching)

Branch-and-Bound has *exponential* complexity.

Branch-and-Bound: Algorithm

Initialization: $LB = 0$ (or $LB = -\infty$), $UB = +\infty$;
Add $[P]$ to the list of active problems;

1. Select an active problem P_j and compute a lower bound $LB(P_j)$ by solving a relaxation of P_j (**bounding**);
2. If $LB(P_j) \geq UB$ return (**pruning**). Includes the case when P_j is infeasible ($LB(P_j) = +\infty$);
3. If the solution of the relaxation is *feasible* we have a feasible solution (UB) to $[P]$. If the solution is better than UB, update UB, return;
4. If the solution of the relaxation is *feasible*, return;
5. Split the set S_j of candidate solutions for P_j in S_{j1} and S_{j2} (**branching**), add the associated problems P_{j1} and P_{j2} to the list of active problems (and remove P_j);
6. Go to 1.

Branching

Branching is the split of the current problem (defined by the set of candidate solutions) in two or more subproblems.

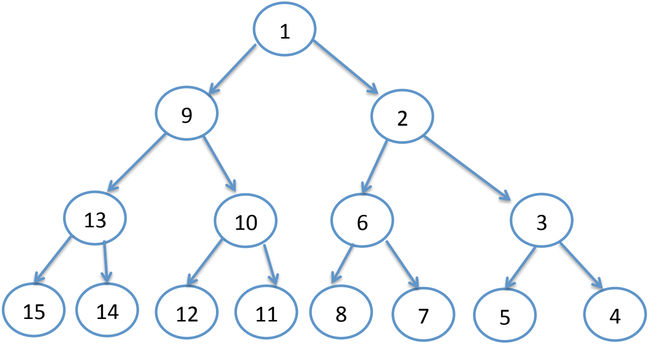
- combinatorial algorithm: select/forbid an item, arc, choice (e.g., KP01);
- ILP problem P: select the branching variable, e.g., the most fractional variable (let it be x_1);
- create the subproblems P_1 and P_2:

$$[P_1] := [P], x_1 \leq \lfloor x_1 \rfloor \qquad\qquad [P_1] := [P], x_1 \geq \lceil x_1 \rceil$$

Search Strategy

Defines the way the next active problem (node) is selected during the search.

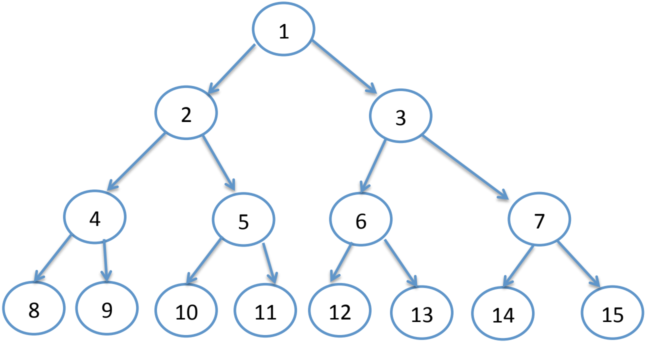
Depth first



Search Strategy

Defines the way the next active problem (node) is selected during the search.

Breadth first



Search Strategy

Best first

Select the problem (node) having the best (i.e., smallest) lower bound. It requires to solve the relaxation of all the active nodes (computationally expensive). It reduces the number of explored nodes (on average).

Cutting planes

Solve the continuous relaxation of the problem by iteratively adding additional constraints (valid inequalities or *cuts*) such that:

- the current fractional solution violates the added cuts;
- no integer solution violates the added cuts (which are thus *valid*)

By adding cuts, the feasible region is reduced: the obtained lower bound is improved at each iteration.

- upper bound (integer feasible solution)

- optimal solution (integer optimal)

- lower bound strengthened by valid inequalities
(e.g., optimal solution of the continuous relaxation+inequalities)

- lower bound (e.g., optimal solution of the continuous relaxation)

Cutting planes

Example:

$$\max x1 + x2 + x3 \tag{1}$$
$$x1 + x2 \leq 1 \tag{2}$$
$$x2 + x3 \leq 1 \tag{3}$$
$$x3 + x1 \leq 1 \tag{4}$$
$$x1, x2, x3 \in \mathbb{Z} \quad (ILP) \tag{5}$$

consider the relaxation $x1, x2, x3 \geq 0$
and the valid inequality $x1 + x2 + x3 \leq 1$ (clique inequality).

Strong formulations for ILP

Strong formulations for ILP

We will see in the following lectures that different ILP formulations provide different continuous relaxations, with a strong impact on the practical solvability of the problem.

Theorem - For any ILP, there exists an LP formulation whose optimal solution is integer (thus *feasible* and *optimal* for the ILP).

- Unfortunately, this ideal formulation has in general exponentially many inequalities, and it is NP-complete to generate one of these inequalities.
- However, producing some "strong" inequalities which improve the LB provided by the continuous relaxation can be very useful for the solution of generic ILPs.

LP Models with "too many" constraints

Solving an ILP with exponentially many constraints (like the TSP model) asks to solve an LP with the same constraints (and then apply B&B). Consider the generic LP:

$$\begin{aligned} \min c'x \\ Ax \geq b \\ x \geq 0 \end{aligned}$$

where A is too large to be explicitly given to an LP solver.

Key observation: not all the constraints are *active* at an optimal solution. Thus, consider a *subset* of them which are *enough*.

Separation Algorithm

1. Initialize $\tilde{A}, \tilde{b}$ with a subset of the constrains defined by A and b
2. Solve the corresponding LP and get x^*:

$$\begin{aligned} \min c'x \\ \tilde{A}x \geq \tilde{b} \\ x \geq 0 \end{aligned}$$

3. If x^* satisfies all the constraints $Ax \geq b$ then it is optimal for the complete problem, otherwise add the violated constrains and go to 2)

The crucial point is checking the condition (3) (*separation*).
Theorem - (Grötschel, Lovász, Schrijver, 1981) - If (3) can be solved in polynomial time, the whole procedure converges in polynomial time.

Observation If we stop the Separation Algorithm before convergence, we have a lower bound on the optimal solution value of the LP.

LP Models with "too many" variables

Solving an ILP with exponentially many variables asks to solve an LP with the same variables (and then apply B&B).
Consider the generic LP:

$$\begin{aligned} \min c'x \\ Ax \geq b \\ x \geq 0 \end{aligned}$$

where A has too many *columns* to be explicitly given to an LP solver.

Key observation: At most *m basic* variables have a value > 0 at an optimal LP solution. The simplex algorithm explores a (hopefully) small subset of the polytope vertices to find the optimal solution. Thus, there is no need to explicitly consider all the problem variables.

Column Generation Algorithm

1. Initialize $\tilde{A}, \tilde{b}$ and $\tilde{x}$ with a subset of the columns containing a feasible solution.
2. Solve the corresponding LP and get x^* and an associated base B (or dual solution $y^* = c_B B^{-1}$):

$$\begin{aligned} \min c'x \\ \tilde{A}x \geq \tilde{b} \\ x \geq 0 \end{aligned}$$

3. If (x^*, y^*) is *optimal* for the original problem, stop. Otherwise, compute a variable which would reduce the solution cost, add it to $\tilde{A}$ and go to (2)

The crucial point is checking optimality in (3) (*column generation*).
Theorem - (Grötschel, Lovász, Schrijver, 1981) - If (3) can be solved in polynomial time, the whole procedure converges in polynomial time.

Observation - If we stop the Column Generation before convergence, we have an upper bound on the optimal solution value of the LP (useless).

Column Generation - Optimality Condition

Checking whether an optimal solution (x^*, π^*) to the *reduced problem* is optimal for the *complete problem*, asks to check if there exists a variable x_j with negative reduced cost: $\tilde{c}_j = c_j - c'_B B^{-1} A_j < 0$. Here $A_j \notin \tilde{A}$, otherwise it would enter the base.

$$\exists A_j : c_j - c'_B B^{-1} A_j < 0?$$

i.e., since $c'_B B^{-1} = \pi^*$ (dual optimal solution),

$$\exists A_j : c_j - \pi'^* A_j < 0?$$

Equivalently, does it exist a violated dual constraint?
Note that here π^* is a parameter.

MIP solvers

MIP solvers

MIP solvers

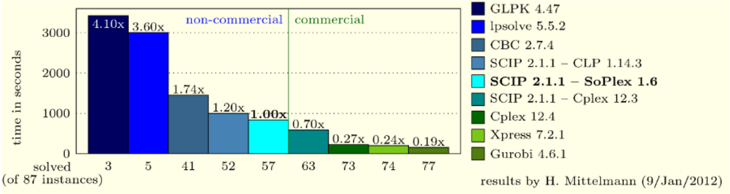
Modern MIP solvers are all based on a Branch-and-Cut framework: at each node of the B&B, the solver generates some inequalities (cuts) to improve the LB associated with the node.
In addition, MIP solvers include:

- preprocessing;
- primal heuristics;
- improved search strategies;
- etc.

MIP solvers

There are several available MIP solvers, with different capabilities and licensing. When comparing MIP solvers, consider the following:

- Capability: what kind of problems can be solved;
- Capability: what kind of models can be solved;
- Performance (profiling problem);
- Licensing;
- Documentation, friendliness, debug level.



results by H. Mittelmann (9/Jan/2012)