Column generation

Giovanni Righini
Università degli Studi di Milano

Complementi di RIcerca Operativa

UNIVERSITÀ DEGLI STUDI DI MILANO

Linear programming

Any linear programming problem can be written as:

$$\begin{aligned} \text{maximize } z' &= c'x \\ \text{s.t. } Ax &\le b \\ x &\ge 0 \end{aligned} \qquad\qquad \begin{aligned} \text{minimize } z &= cx \\ \text{s.t. } A'x &\ge b' \\ x &\ge 0 \end{aligned}$$

where $z = -z'$, $c = -c'$, $A = -A'$ and $b = -b'$.

In this form the LP problem only has:

- m inequality constraints
- n non-negative variables.

Standard form

To be solved with the simplex algorithm, the LP problem must be put in standard form:

$$\begin{aligned} \text{minimize } z &= cx \\ \text{s.t. } Ax + I\hat{x} &= b \\ x &\geq 0 \\ \hat{x} &\geq 0. \end{aligned}$$

- The objective function is rewritten in minimization form;
- the constraints are rewritten as linear equalities, by the use of m slack variables.

The tableau

The simplex algorithm works on a data-structure, called “tableau”, which is a matrix with:

- a row for each constraint $(1, \dots, m)$;
- a column for each variable, including slack variables $(1, \dots, n+m)$;
- an additional row (row 0) with the reduced costs;
- an additional column (column 0) with the values of the basic variables.

Initially we have this tableau (initial canonical form):

0	c_1	$\dots$	c_n	0	0	$\dots$	0
b_1	a_{11}	$\dots$	a_{1n}	1	0	$\dots$	0
b_2	a_{21}	$\dots$	a_{2n}	0	1	$\dots$	0
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	0
b_m	a_{m1}	$\dots$	a_{mn}	0	0	$\dots$	1

The canonical form

An LP problem is in canonical form when two conditions are satisfied:

- m basic columns form an $(m \times m)$ identity matrix;
- the reduced costs of the basic columns are null.

A canonical for is strong if and only if all entries in column 0 are non-negative, i.e. all basic variables have non-negative values:

$$(x, \hat{x}) \geq 0.$$

The simplex algorithm

The simplex algorithm works on LP problems in canonical form.

It is an iterative algorithm. At each iteration:

- one basic column leaves the basis;
- one non-basic column enters the basis;
- all entries in the tableau are modified to produce a canonical form corresponding with the new basis.

The stop criterion

For each LP problem instance three cases can occur:

- it is infeasible;
- it is unbounded;
- it has a finite optimal solution.

Correspondingly, after a finite number of iterations the simplex algorithm terminates in one of three different ways:

- it detects infeasibility;
- it detects unboundedness;
- it finds an optimal solution.

The optimality condition that stops the algorithm in the third case is:

All reduced cost coefficients are non-negative.

LP duality

For each primal LP problem instance we can define a dual LP problem instance:

P) minimize $z = cx$

$$\text{s.t. } Ax \geq b$$
$$x \geq 0$$

D) maximize $w = b\lambda$

$$\text{s.t. } A^T\lambda \leq c$$
$$\lambda \geq 0.$$

The dual problem has:

- as many dual variables λ as the primal constraints
- as many dual constraints as the primal variables x.

The dual tableau

Since the entries in the tableau are those of A, b and c in both cases, it is possible to interpret the same tableau as a description of the primal or the dual problem in standard form.

It is also possible to run the dual simplex algorithm on the tableau of the primal problem. This is equivalent to run the primal simplex algorithm on the tableau of the dual problem.

As we read the values of the primal basic variables on column 0, we also read the values of the dual basic variables on row 0.

The reduced cost coefficients of the primal problem are the values of the corresponding dual variables.

Complementary slackness

- Original primal variable x $\Rightarrow$ Dual constraint $\Rightarrow$ Dual slack/surplus variable $\hat{\lambda}$;
- Slack/surplus primal variable $\hat{x}$ $\Rightarrow$ Primal constraint $\Rightarrow$ Dual variable λ.

A one-to-one correspondence between primal and dual bases can be established through complementary slackness conditions.

$$x\hat{\lambda} = 0 \quad \hat{x}\lambda = 0.$$

The only primal-dual pair of solutions $((x, \hat{x}), (\lambda, \hat{\lambda}))$ satisfying the C.S.C. and such that $(x, \hat{x})$ is primal feasible and $(\lambda, \hat{\lambda})$ is dual feasible is made by the two optimal solutions.

Reduced costs

The expression of the reduced cost on column j (we read it on row 0 of the tableau) is:

$$r_j = c_j - \sum_{i=1}^{m} a_{ij}\lambda_i .$$

Whenever a column with negative reduced cost exists:

- the stop criterion of the simplex algorithm is not satisfied;
- that column is a candidate to enter the primal basis at the next iteration of the simplex algorithm.

The reduced costs of the primal variables (columns of the primal tableau) depend on the current values of the dual variables λ.

Column generation

The main idea of column generation is to solve a restricted LP, where not all columns are in the tableau, i.e. not all variables are allowed to be basic.

After reaching optimality, we know all values of the dual variables (λ and we can ask the question (pricing problem):

Is there a column j not currently in the tableau, such that its reduced cost r_j in the current basic solution is negative?

The answer depends on its coefficients a and c (i.e. the “structure” of the column), because $r_j = c_j - \sum_{i=1}^{m} a_{ij}\lambda_i$.

- If the answer is “No”, we are guaranteed that the current optimal solution of the restricted LP is also an optimal solution of the whole LP.
- If the answer is “Yes”, we insert the negative reduced cost column into the restricted LP and we go on pivoting.

Why should we use CG?

The main reason for applying column generation to LP problems is their excessive number of columns (variables).

LP problems may have a huge number of variables (for instance when variables have many indices).

The computing time spent by the simplex algorithm usually depends more on the number of constraints than on the number of variables: so, it may be convenient to solve the dual of an LP problem with many constraints.

LP problems may show a block-diagonal structure, so that they can be decomposed into independent sub-problems when linking constraints are considered separately.

LP problems may arise as linear relaxations of reformulations of combinatorial problems with an exponential number of variables.

Some remarks on CG (1)

Remark 1. Primal feasibility of the restricted LP problem must be guaranteed. For this purpose we can add one or more dummy columns with very high cost whose structure ensures feasibility.

Remark 2. We do not need ensuring primal boundedness: if the restricted LP is unbounded, the complete LP is also unbounded.

Remark 3. As negative reduced cost columns are inserted into the restricted LP, some non-basic columns with “large” positive reduced cost can be deleted from it. This allows us to save space when storing very large tableaux.

Some remarks on CG (2)

Remark 4. We can keep a "pool" of known columns in a suitable data-structure, where we search for negative reduced cost columns every time the pricing sub-problem must be solved. Deleted columns are stored into the pool for possible future re-use.

Remark 5. If we do not have an explicit list of the columns, we must have an implicit description of them. In this case we must actually solve a pricing sub-problem that is an optimization problem in itself, to generate them:

$$\begin{aligned} &\text{minimize } r = c(a) - a\lambda \\ &\text{s.t. } a \in \mathcal{A} \end{aligned}$$

where $\mathcal{A}$ is the set of feasible columns and the cost c depends on the structure of the column described by coefficients a. In the pricing sub-problem the coefficients a play the role of variables.

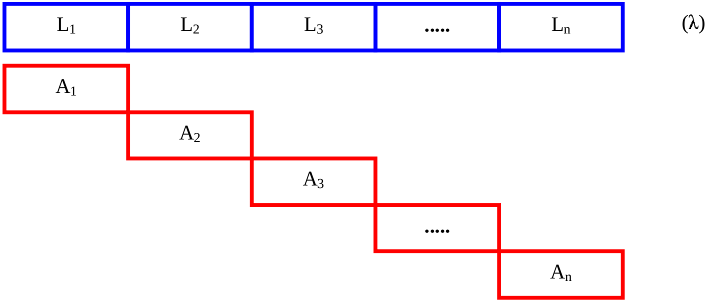
Pricing

Instead of generating only one column at a time, it may be convenient to generate several columns with negative reduced cost, in every pricing iteration. This is called multiple pricing.

To speed up CG, it is also convenient to generate columns with negative reduced cost in a heuristic way, as far as possible. However to be guaranteed that no columns with negative reduced cost exist, we must use an exact optimization algorithm. When heuristic pricing fails, we resort to exact pricing.

Dantzig-Wolfe decomposition (1960)

Some LP may have special structure, that can be exploited by suitable algorithms: for instance they may exhibit a block-diagonal structure, i.e. the non-zero coefficients in the constraint matrix are grouped into rectangular blocks, possibly with a few exceptions represented by linking constraints, i.e dense rows with non-zero coefficients in columns corresponding to different blocks.



Dantzig-Wolfe decomposition (1960)

If we remove the linking constraints, we are left with a decomposable problem; it can be solved as a set of smaller independent sub-problems.

We keep the linking constraints in a master problem and the other constraints in the independent sub-problems. Each sub-problem has only a subset of the variables. The master problem has all the variables and we solve it with CG.

Generation of new columns with negative reduced cost is done independently for each sub-problem. They all have in common the same current values of the dual variables of the master problem but possibly different structures in different subproblems.

Discrete optimization

Optimization problems often require using integer or binary variables.

Integer variables are used to represent quantities that must be integer multiples of elementary, non-divisible amounts: number of employees in a call center, number of pallets on a truck, number of beds in a hospital, etc...

Binary variables are used to represent logic conditions and yes/no decisions: assignments, orderings, selections, etc...

Even if problems with discrete variables are formulated with linear objective function and linear constraints, they are in general *NP*-hard, i.e. difficult to solve from a computational viewpoint.

There are some easy exceptions that we know to be solvable in polynomial time: shortest path, max flow, min cost matching, min cost spanning tree, etc...

Formulations

An integer linear programming problem is formulated as follows:

$$z^* = \min\{cx : x \in X\}$$

where X is a discrete set defined by linear constraints

$$X = \{x \in \mathcal{Z}_+^n : Ax \geq b\}.$$

For simplicity of notation we assume here that X has two properties:

- it is not empty;
- it is bounded.

In other words, X is a polytope.

Hence a finite optimal solution exists and its value z^* is finite.

The concepts can also be extended to the cases of empty and unbounded polyhedra, but they are less interesting in practice.

The convex hull

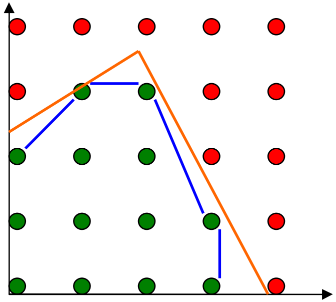
Given an ILP problem

$$z^* = \min\{cx : x \in X\}$$

its feasible region contains the same set of integer points as

$$z^* = \min\{cx : x \in conv(X)\}$$

where $conv(X)$ indicates the convex hull of the integer points in X.



Convexification

This is the ideal formulation: the extreme points of the convex hull $conv(X)$ are all integer points. This is very useful because we can drop the integrality requirements and solve the problem as an LP problem with continuous variables, still obtaining an integer optimal solution.

The knowledge of the facets of the convex hull of the feasible region, corresponds to our ability to solve the problem efficiently.

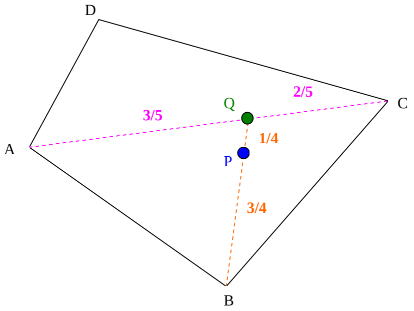
Usually we do not know the facets of the convex hulls of the ILP problems we must solve, but maybe we know for some sub-problems.

Every time we are able to replace a set with the convex hull of its integer points, we call this “convexification”.

Minkowsky and Weyl theorem

Every point in a polyhedron can be obtained as a convex combination of its extreme points.

$$\forall x \in X, \exists \lambda \geq 0 : x = \sum_{i=1}^{n} \lambda_i x_i, \sum_{i=1}^{n} \lambda_i = 1.$$



$Q = \frac{2}{5}A + \frac{3}{5}C$

$P = \frac{3}{4}Q + \frac{1}{4}B = \frac{3}{4}(\frac{2}{5}A + \frac{3}{5}C) + \frac{1}{4}B = \frac{6}{20}A + \frac{5}{20}B + \frac{9}{20}C.$

Reformulation

It may be convenient to reformulate an (I)LP as follows:

$$z^* = \min\{cx : Ax \geq b, x \in X\}$$

where X is a (discrete) set defined by linear constraints

$$X = \{x \in \mathcal{Z}_+^n : Dx \geq d\}.$$

There are two reasons for this:

- we are able to convexify X when it is discrete; in this case $Ax \geq b$ are complicating constraints;
- X can be decomposed into independent sub-problems; in this case $Ax \geq b$ are linking constraints;
- both things may occur simultaneously.

Reformulation

Convexification. We rewrite

$$z^* = \min\{cx : Ax \geq b, x \in X\}$$

as

$$z^* = \min\{cx : Ax \geq b, x \in conv(X)\}.$$

Minkowsky and Weyl's theorem. We rewrite

$$z^* = \min\{cx : Ax \geq b, x \in conv(X)\}$$

as

$$z^* = \min\{cx : Ax \geq b, x = \sum_{i=1}^{n} \lambda_i x_i, \sum_{i=1}^{n} \lambda_i = 1, \lambda \in \{0,1\}^n\}$$

where x_i indicates the generic extreme point of $conv(X)$.

Decomposition

If X is defined by a decomposable matrix, such that $X = \bigcup_{k \in K} X^{(k)}$, we have

$$z^* = \min\{cx : Ax \geq b,$$
$$x^{(k)} = \sum_{i=1}^{n_k} \lambda_i^{(k)} x_i^{(k)}, \sum_{i=1}^{n_k} \lambda_i^{(k)} = 1, \lambda^{(k)} \in \{0,1\}^{n_k} \ \forall k \in K\}$$

where

- K is the set of blocks into which the matrix can be decomposed,
- $X^{(k)}$ is the feasible region of the sub-problem corresponding to each block k,
- n_k is the number of extreme points of $conv(X^{(k)})$,
- $x_i^{(k)}$ is the generic extreme point of $conv(X^{(k)})$.

Reformulation (continued)

- **Objective function.** We rewrite cx as

$$\sum_{k\in K}\sum_{i=1}^{n_k}\lambda_i^{(k)}c_i^{(k)}$$

  where $c_i^{(k)} = cx_i^{(k)}$ is the cost of the extreme point $x_i^{(k)}$.
- **Linking constraints.** We rewrite $Ax \geq b$ as

$$\sum_{k\in K}\sum_{i=1}^{n_k}\lambda_i^{(k)}a_i^{(k)} \geq b,$$

  where $a_i^{(k)} = Ax_i^{(k)}$.

The reformulated and decomposed problem

We obtain

$$z^* = \min\{ \sum_{k \in K} \sum_{i=1}^{n_k} \lambda_i^{(k)} c_i^{(k)} :$$

$$\sum_{k \in K} \sum_{i=1}^{n_k} \lambda_i^{(k)} a_i^{(k)} \geq b,$$

$$\sum_{i=1}^{n_k} \lambda_i^{(k)} = 1, \lambda^{(k)} \in \{0, 1\}^{n_k} \ \forall k \in K \}$$

The variables in this problem are the $\lambda_i^{(k)}$. They are as many as the extreme points of the polyhedra $X^{(k)}$, i.e. they can be exponentially many.

Linear relaxation

The linear relaxation of the master problem is

$$z^*_{LMP} = \min\{ \sum_{k \in K} \sum_{i=1}^{n_k} \lambda_i^{(k)} c_i^{(k)} :$$

$$\sum_{k \in K} \sum_{i=1}^{n_k} \lambda_i^{(k)} a_i^{(k)} \geq b,$$

$$\sum_{i=1}^{n_k} \lambda_i^{(k)} = 1, \lambda^{(k)} \geq 0 \; \forall k \in K \}.$$

This is a linear programming problem with exponentially many variables: we solve it with column generation.

It is called linear restricted master problem (LRMP). New columns are

generated within the convex hull of the feasible regions of the sub-problems. The constraint $\sum_{i=1}^{n_k} \lambda_i^{(k)} = 1$ is called convexity constraint.

How good is the bound we get?

The optimal value of the linear relaxation of the master problem, z^*_{LMP} is the same as the optimal value of the Lagrangean dual problem we obtain when we apply Lagrangean relaxation to the constraints of the master problem.

The constraints taken into account in the pricing sub-problem are convexified, while the constraints remaining in the master problem are not.

Two main differences between LR and CG are:

- in CG optimal dual variable values are computed by the LP solver; in LR approximate dual variable values are computed by subgradient optimization;
- in LR we always obtain a valid dual bound at each iteration; in CG the optimal value of the RLMP is a valid dual bound only at the end of CG.

Branch-and-price

When

- the LRMP problem has been solved to optimality,
- the pricing algorithm states that there no more columns with negative reduced cost in any sub-problem,

we have the optimal solution of the linear master problem. Its value z^*_{LMP} is a valid lower bound to the optimal value of the original minimization problem.

The lower bound is achieved only at the end of column generation.

However we can still use the dual values associated with the constraints of the master problem as Lagrangean multipliers to compute a valid lower bound at any point during column generation. Linear programming can be seen as an alternative to the sub-gradient algorithm to provide dual values in a Lagrangean relaxation algorithm.

Branch-and-price

The current optimal value z^*_{RLMP} of the RLMP during column generation is always monotonically decreasing. It is an upper bound to z^*_{LMP}

The lower bound z^*_{LR} obtained from Lagrangean relaxation (with the dual values provided by the master problem) is not monotone in general. It is a lower bound to z^*_{LMP}.

The two bounds tend to coincide as far as CG progresses, but there may be a “tailing-off” effect: many iterations are needed for very small improvements of the two bounds.

It is common practice to stop column generation when the gap between the two bounds is small.

It is also possible to use stabilization techniques when the master problem is degenerate.

Branch-and-price

The optimal solution of the LMP can be fractional.

To achieve integrality, we must branch. This is a fundamental difference between column generation and cutting planes algorithms (row generation) that allow producing an integer optimal solution without branching. It is recommended not to branch on binary

variables λ, because setting one of them to 0 excludes one specific column (likely to be the optimal one) as a feasible solution of a pricing sub-problem. In general it is difficult to find the second best, the third best etc. solution to an ILP problem.

It is more recommendable to branch on the original variables x.

Ad hoc branching strategies can be devised for each particular problem. A branch-and-price algorithm is called “robust” when the branching policy does not change the structure of the pricing sub-problem.

Heuristics

The columns in the RLMP can also be used as building blocks for a heuristic algorithm.

One typical master-problem-based heuristic consists of iteratively fixing to 1 the column with the largest λ value in the current optimal solution of the RLMP, to update the constraints RHSs and to solve the LRMP again (without generating any additional column), until a complete integer and feasible solution is produced.

This is easy to implement with general-purpose ILP solvers (CPLEX, GuRoBi, etc.).