Column Generation: Cutting Stock
– A very applied method

Thomas Stidsen

thst@man.dtu.dk

DTU-Management
Technical University of Denmark

Outline

- History
- The Simplex algorithm (re-visited)
- Column Generation as an extension of the Simplex algorithm
- A simple example !

Introduction to Column Generation

Column Generation (CG) is an old method, originally invented by Ford & Fulkerson in 1962 !

- Benders decomposition algorithm dealt with adding constraints to a master problem
- CG deals with adding variables to a master problem
- CG is one of the most used methods in real life with lots of applications.

(Relax it is significantly easier than Benders algorithm)

Given an LP problem

Min

$$c^T x$$

s.t.

$$\begin{aligned} Ax &\geq b \\ x &\geq 0 \end{aligned}$$

This is a simple “sensible” (in the SOB notation) LP problem

The simplex slack variables

We introduce a number slack variables, one for each constraint, (simply included as extra x variables). They are also positive.

Min

$$z = c^T x$$

s.t.

$$\begin{aligned} Ax &= b \\ x &\geq 0 \end{aligned}$$

The simplex (intermediate) problem

In each iteration of the simplex algorithm, a number of $basic$ variables x_B may be non-zero (B matrix) and remaining variables are called $non-basic\ x_N$ (A matrix). In the end of each iteration of the simplex algorithm, the values of the variables x_B and x_N contain a feasible solution.

Min

$$z = c_B^T x_B + c_N^T x_N$$

s.t.

$$\begin{aligned} Bx_B + Ax_N &= b \\ x_B, x_N &\geq 0 \end{aligned}$$

A reformulated version

Min

$$z = c_B^T x_B + c_N^T x_N$$

s.t.

$$\begin{aligned} x_B &= B^{-1}b - B^{-1}A_N x_N \\ x_B, x_N &\geq 0 \end{aligned}$$

A reformulated version

Min

$$z = c_B B^{-1} b + (c_N - c_B B^{-1} A_N) x_N$$

s.t.

$$\begin{aligned} x_B &= B^{-1} b - B^{-1} A_N x_N \\ x_B, x_N &\geq 0 \end{aligned}$$

Comments to the reformulated version

At the end of each iteration (and the beginning of the next) it holds that:

- The current value of the non-basic variables are: $x_N = 0$.
- Hence the current optimization value is: $z = c_B B^{-1} b$.
- And hence the values of the basic variables are: $x_B = c_B B^{-1}$.

Comments to the reformulated version

- The $coefficients$ of the basic variables (x_B) in the objective function are zero ! This simply means that the x_B variables are not "participating" in the objective function.
- The coefficients of the non-basic variables are the so-called **reduced costs**: $c_N - c_B B^{-1} A_N$
- Because $z = c_B^T x_B + c_N^T x_N$, and we want to minimize, non-basic variables with negative reduced costs can improve the current solution, i.e. $c_N - c_B B^{-1} A_N < 0$

The Simplex Algorithm

$x_S = b$
$x_{\overline{S}} = 0$
while $MIN_j(c_j - c_B B^{-1} A_j) < 0$
    Select new basic variable j: $c_j - c_B B^{-1} A_j < 0$
    Select new non-basic variable j' by
        increasing x_j as much as possible
    Swap columns between matrix B and matrix A
end while

Comments to the algorithm

There are three parts of the algorithm:

- Select the next basic variable.
- Select the next non-basic variable.
- Update data structures.

We will now comment on each of the three steps.

Selecting the next basic variable

- To find the most negative reduced cost, all the reduced costs must be checked: $c_{red} = MIN_j(c_j - c_B B^{-1} A_j) < 0$.
- In the revised simplex algorithm, first the system $yB = c_B$ is solved. (In the standard simplex algorithm, the reduced costs are given directly).

Conclusion: This part (mainly) is dependent on *the number of variables*.

Selecting the next non-basic variable

- There are the same number of basic variables (to compare) as there are constraints
- We are only considering one non-basic variable
- Basically we are looking at the system: $x_B = x_B^* - B^{-1}A_j x_j$.
- The revised simplex algorithm solves the system $Bd = a$, where a is the *column* of the entering variable.

Conclusion: This part is dependent on *the number of constraints*

Bookkeeping

In the end what needs to be done (in the revised simplex algorithm) ?

- Update of the matrix's B and A: Simply swap the columns.

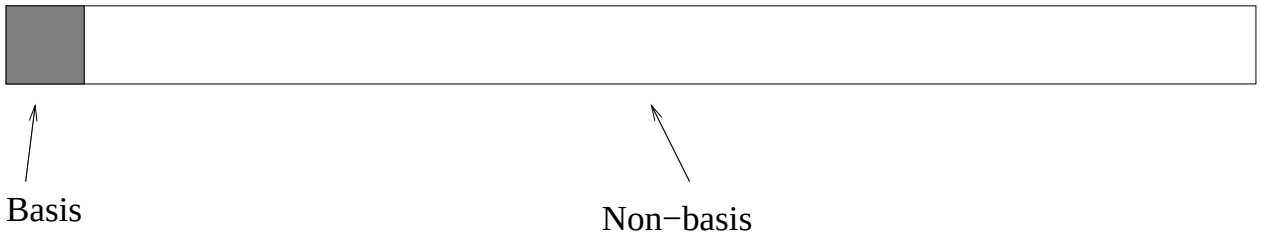
Conclusion: This part is dependent on *the number of constraints*

Comments to the algorithm

- Selecting a non-negative variable principally requires checking all the many variables (time consuming).
- Selecting the new non-basic variable:
  - There are the same number of basic variables (to compare) as there are constraints
  - We are only considering one non-basic variable
  - Basically we are looking at the system: $x_B = x_B^* - B^{-1}A_j x_j$
- Update of the matrix's B and A: Simply swap the columns.

LP programs with many variables !

The crucial insight: The number of non-zero variables (the basis variables) is (at most) equal to the number of constraints, hence eventhough the number of possible variables (columns) may be large, we only need a small subset of these in the optimal solution.



The simple column generation idea

The simple idea in column generation is *not* to represent all variables explicitly, represent them implicitly: When looking for the next basic variable, solve an optimization problem, finding the variable (column) with the most negative reduced cost.

The column generation algorithm

$x_S^* = b$
$x_{\overline{S}}^* = 0$
repeat
    repeat
        Select new basic variable j: $c_j - c_B B^{-1} A_j < 0$
        Select new non-basic variable j' by
            increasing x_j as much as possible
        Swap columns between matrix B and matrix A
    until $MIN_j(c_j - c_B B^{-1} A_j) >= 0$
    $x = MIN(c_j - c_B B^{-1} A_j, LEGAL)$
until no more improving variables

The reduced costs: $c_j - c_B B^{-1} A_j < 0$

Lets look at the individual components of the reduced costs $c_j - c_B B^{-1} A_j < 0$:

- c_j: The *original* costs for the variable
- A_j: The column in the A matrix, which defines the variable.
- $c_B B^{-1}$: For each constraint in the LP problem, this factor is multiplied to the column. c_B is the cost vector for the current basic variables and B^{-1} is the inverse basic matrix.

Alternative $c_B B^{-1}$

For an optimization problem: $\{min\ cx | Ax \geq b, x \geq 0\}$. Outside the inner loop, the reduced costs are: $c_N - c_B B^{-1} A_N \geq 0$ for the existing variables. Notice that this **also** holds for the basic variables for which the costs are all zero, i.e. $c - cB^{-1}A \geq 0$. If we rename the term $c_B B^{-1}$ to the name π we can rewrite the expression: $\pi A \leq c$. But this is exactly dual feasibility of our original problem ! Hence we can use the *dual variables* from our inner loop when looking for new variables.

The column generation algorithm

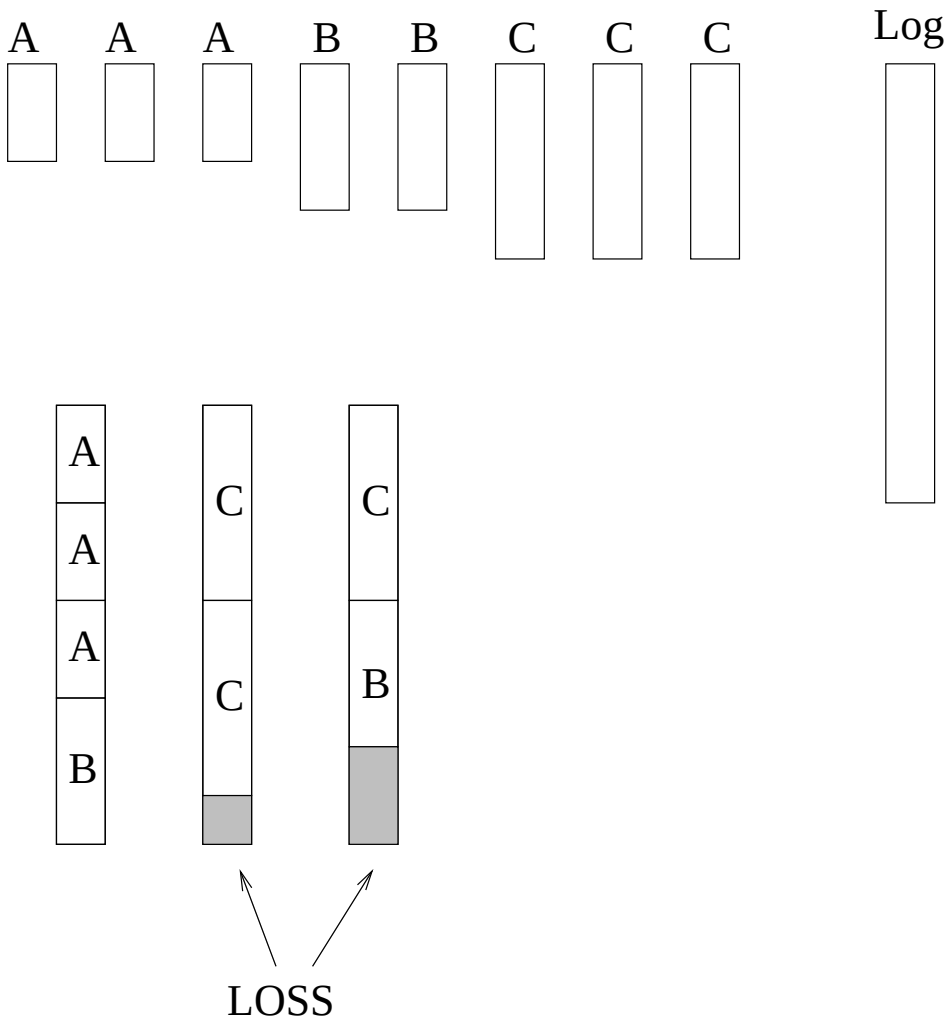
$x_S^* = b$
$x_{\overline{S}}^* = 0$
repeat
$\pi = MIN(min\ cx | Ax \geq b, x \geq 0)$
$x_j = MIN(c_j - c_B B^{-1} A_j, LEGAL)$
until no more improving variables

Cutting Stock Example

The example which is always referred to regarding column generation is the cutting stock example. Assume you own a workshop where steel rods are cut into different pieces:

- Customers arrive and demand steel rods of certain lengths: 22 cm., 45 cm., etc.
- You serve the customers demands by cutting the steel rods into the rigth sizes.
- You recieve the rods in lengths of e.g. 200 cm.

Cutting Stock Example



Cutting Stock Objective

How can you minimize your material waste (what's left of each rod after you have cut the customer pieces) ?

Cutting Stock Formulation 1

Min:

$$z = \sum_k y_k$$

s.t.:

$$\sum_k x_k^i = b_i \qquad \forall i$$

$$\sum_i w_i \cdot x_k^i \leq W \cdot y_k \qquad \forall k$$

$$x_k^i \in N_0 \qquad y_k \in \{0,1\}$$

Cutting Stock Formulation 1

What is wrong with the above formulation ?

- It contains many symmetric solutions, $k!$. This makes the problem extremely hard for branch and bound algorithms.
- There are many integer or binary variables

For these reasons, it is never used in practice.

New Formulation

How can we change the formulation ?

- Instead of focussing on which steel rod a particular part is to be cut from, look at possible **patterns** used to cut from a steel rod.
- The question is then changed to focussing on how many times a particular pattern is used.

Cutting Stock Formulation 2

Min:

$$z = \sum_j x_j$$

s.t.:

$$\sum_j a_j^i \cdot x_j \geq b^i \qquad \forall i$$

$$x_j \in N_0$$

Cutting Stock Formulation 2

What are the x_j and a_j^i ?

- a_j^i is a cutting pattern, describing how to cut one steel rod into pieces.
- Notice that any legal solution to formulation 1 will always be consisting of a number of cutting patterns
- x_j is then simply the number of times that cutting pattern is used.

Cutting Stock Formulation 2

There are two different problems with the formulation:

- We will only solve the **relaxed** problem, i.e. the LP version of the MIP problem, how do we get integer solutions ?
- We need to “consider” all the cutting patterns ???

Today we will only deal with the last problem ...

Cutting Patterns

We have a number of questions to these cutting patterns:

- But where do we get the cutting patterns from ?
- How many are there ? $\binom{|i|}{\overline{a}} = \frac{|i|!}{\overline{a}!(|i|-\overline{a})!}$, i.e. a lot !
  - Where: $|i|$ is the number of different types (lengths) required
  - Where: $\overline{a}$ is the average number of cutsizes in each of the patterns.

Cutting Patterns

This is a very real problem: Even if we had a way of generating all the legal cutting patterns, our standard simplex algorithm will need to calculate the “efficient” variables, but we will not have memory to contain the variables in the algoritm, now what ?

The improving variable

How can we find the improving variable. We know that *any* variable where $c_j - c_B B^{-1} A_j = c_j - \pi A_j$. But we want not just to choose any variable, but $min\ c_j - \pi A_j$, the so called Dantzig rule, which is the standard non-basic variable selection rule. The Dantzig rule hence corresponds to the original costs (which might be zero), subtracted by the dual variable vector multiplied by the column of the new variable x_j

Pattern generation: The subproblem

For each iteration in the Simplex algorithm, we need to find the most negative column. We can do that by defining a new optimisation problem:

$$Min\ z = 1 - \sum_i \pi_i a_j^i$$

Pattern generation: Ignore the constants

Max:

$$z = \sum_i \pi_i a_j^i$$

s.t.:

$$\sum_i l_i \cdot a_j^i \leq L$$

$$a_j^i \in Z^+$$

What about the j index ?

What is the subproblem ?

The subproblem is the classical **knapsack** problem.

- The problem is theoretically NP-hard ....
- But it is an “easy” NP-hard problem, meaning that we can solve the problem for relative large sizes of N, i.e. number of items (in this case different lengths)

Knapsack solution methods

The problem may e.g. be solved using dynamic programming. We will ignore the problem of efficiency and just use standard MIP solvers. Notice that we are going to solve exactly the same problem, but with updated profit coefficients.

The Column Generation Algorithm

Create initial columns

repeat

Solve master problem, find π

Solve the subproblem $min\ z_{sub} = 1 - \sum_i \pi_i \cdot a_i$

Add new column problem to master problem

until $z_{sub} \geq 0$

Lets look at an example !

We have some customers who wants:

- Pieces in three lengths: 44 pieces of length 81 cm., 3 pieces of length 70 cm. and 48 pieces of length 68 cm.
- We have steel rods of length 218 cm. of unit price.

How do we get the initial columns ???

The initial columns

We need some initial columns and we basically have two choices:

- Start with fake columns, which are so expensive that we know they will not be in the final problem.
- If possible, create some more “competetive” columns.

We simply use columns where each column contains 1 in different rows

Master problem

```
\ Initial master problem

minimize
         x_1 + x_2 + x_3
subject to
l1: x_1                          >= 44
l2:      x_2                     >= 3
l3:           x_3                >= 48
end
```

Sub problem: Given $\pi = [1, 1, 1]$

```
\ initial sub problem

maximize
          a_1 + a_2 + a_3
subject to
l:        81a_1 + 70a_2 + 68a_3 <= 218
bound
          a_1 <= 2
          a_2 <= 3
          a_3 <= 3
integer
          a_1 a_2 a_3
end
```

First sub-problem solution

The best solution $(a_1, a_2, a_3) = (0, 0, 3)$

New Master Problem

```
\ second master problem

minimize
         x_1 + x_2 + x_3 + x_4
subject to
l1: x_1                          >= 44
l2:      x_2                     >= 3
l3:           x_3       + 3x_4   >= 48
end
```

First master solution

The best solution **duals** $(\pi_1, \pi_2, \pi_3) = (1.0, 1.0, 0.33)$

Second Sub problem: Given $\pi = [1.0, 1.0, 0.33]$

```
\ initial sub problem

maximize
          a_1 + a_2 + 0.33a_3
subject to
l:        81a_1 + 70a_2 + 68a_3 <= 218
bound
          a_1 <= 2
          a_2 <= 3
          a_3 <= 3
integer
          a_1 a_2 a_3
end
```


First sub-problem solution

The best solution $(a_1, a_2, a_3) = (0, 3, 0)$

New Master Problem

```
\ second master problem

minimize
         x_1 + x_2 + x_3 + x_4   + x_5
subject to
l1: x_1                                  >= 44
l2:      x_2                    + 3x_5 >= 3
l3:           x_3          + 3x_4          >= 48
end
```

Why is this so interesting ???

I have told you a number of times that this is the most applied method in OR, why ?

- We can solve problems with a "small" number of constraints and an exponential number variables ... this is a special type of LP problem but ...
- Through the so called Dantzig-Wolfe decomposition we can change *any* LP problem to a problem with a reduced number of constraints, but an increased number of variables ...
- By performing clever decompositions we can improve the LP bounding ...

Subproblem solution

Usually (though not always) we need an efficient algorithm to solve the subproblem. Subproblems are often one of the following types:

- Knapsack (like in the cutting stock problem).
- Shortest path problems in graphs.

Hence: To use column generation efficiently you often need to know something about complexity theory ...

Getting Integer Solutions

There are several methods:

- Rounding up, not always possible ....
- Getting integer solutions using (meta)heuristics. This is the reason for the great interest in the set partioning/set covering problem.
- Branch and price, hard and quite time consuming ...