# The Cutting Stock Problem

**Abstract**

In the mid 90's Petra Mutzel and I taught a course on optimization. We gave the cutting stock problem as a programming challenge. Naveen Garg and I competed in the competition. The first section of this paper contains what Naveen and I wrote back then.

## Contents



## 1 A Solution from 1995

The *cutting stock* problem is easy to state: we are given a supply of rods of length $L$ and are supposed to cut these rods into $n_i$ pieces of length $l_i$, $0 \leq i < k$. The goal is to minimize the number of rods used.

The cutting stock problem is easily formulated as an integer programming problem. Call a vector $a^T = (a_0, \ldots, a_{k-1})$ a pattern if $\sum_j a_j l_j \leq L$, let $a_0$, ..., $a_{N-1}$ be the set of all possible patterns, and let $n^T = (n_0, \ldots, n_{k-1})$ be our vector of demands. Let $x_i$, $0 \leq i < N$, be the number of times pattern $a_i$ is used. The goal is to minimize $\sum_i x_i$ subject to $\sum_i x_i a_i = n$, $x_i \geq 0$, and $x_i$ integral.

⟨*main_program.c*⟩≡

```
#include <LEDA/array.h>
#include <LEDA/list.h>
#include <LEDA/d_array.h>
#include <LEDA/stream.h>
#include <LEDA/sortseq.h>
#include "math.h"
#include "/apps/cplex/cpxdefs.inc"
typedef array<int> pattern;
static double eps = 1/1000000;
⟨knapsack⟩
⟨lower bounds⟩
⟨input routine⟩
⟨heuristics⟩;
⟨setting up the LP⟩;
⟨checking optimality and generating new columns⟩;
⟨adding columns to the LP⟩;
⟨postprocessing the LP solution⟩;
main()
{
  int all = Yes("if you want all examples types y       "); newline;
  int number_of_example = 1;
  double total_ratio = 0;
```

```
  if (!all) number_of_example = read_int("number of example =  "); newline;
  while (number_of_example <= 30)
  {
    string extension;
    sprintf(extension,"%d%",number_of_example);
    string file_name = "/KM/usr/mutzel/tmp/lehre/rg" + extension;
    newline; newline;
    cout << "problem number   " << number_of_example; newline;
    array<int> length;
    array<int> demand;
    int L = 600; // out raws are 600 centimeters in length
    read_from_file(file_name,length,demand);
    int i;
    int k = length.high() + 1;
    //if (Yes("if you want to see the problem type y    "))
    { cout <<"length   demand"; newline;
      for (i = 0; i < k; i++) cout << length[i] << "  " << demand[i] <<"\n";
    }
    int lb = lower_bound_1(L,length,demand);
    cout << "first lower bound  " << lb; newline;
    list<pattern> plan;
    int ub = first_fit_decreasing(L,length,demand,plan);
    cout << "first fit decreasing  " << ub; newline;
    FILE* logfile;
    logfile = fopen("logfile","w");
    setlogfile(logfile);
    ⟨LP approach⟩;
    /* let us do some statistics */
    cout << "ratio  " << (double) ub/lb; newline;
    total_ratio + = (double)ub/lb;
    newline;
    if (all) number_of_example++;
    else number_of_example = 31;
  }
  cout << "total ratio  " << total_ratio/30; newline; newline;
}
```

**Fractional Solutions:** If the integrality constraint is dropped for the $x_i$'s then the problem becomes amenable to linear programming, more precisely to the delayed column generation approach to linear programming. Note that the linear programm at hand has a small number of constraints ($k$ is less than 100 for the problems of interest to us) but a large number of columns; $N$ may be on the order of a few million. A crucial observation is now that only $k$ of the columns are actually used in the optimal solution of the linear program. Delayed column generation generates the columns of the constraint matrix in a lazy fashion. It starts with a small number of columns and generates further columns as needed.

**A Knapsack Problem:** Let $c^T = (c_0, \dots, c_{k-1})$ be a vector of real costs. We want to determine whether there is a cutting pattern $a$ such that $c * a >$ *threshhold* for some specified *threshhold*. We will use the procedure with different treshholds at different stages of our algorithm. This is tantamount to a knapsack problem which can be solved by dynamic programming. We assume that $L$ and the $l_i$'s are integers. In the problem at hand all lengths are measured in centimeters. We compute for each $l$, $0 \le l \le L$ the maximal value $c * a$ obtained by a pattern $a$ with $\sum_j a_j l_j = l$.

To do so we use an array $value(0, L)$ of doubles which we initialize to the all zero array. We then step through the rod lengths. Suppose we are considering length $l_j$. If $value[l - lj] + cj > value[l]$ for some $l$ then we have a better value for $value[l]$. We also need to store for each $l$ the pattern which brought about $value[l]$. We do so in an implicit way by means of an array *rod_used*. If $value[l]$ was defined as $value[l - lj] + cj$ we store $j$ in $rod\_used[l]$.

⟨*fill array of values*⟩≡

```
array<double> value(0,L);
array<int> rod_used(0,L);
for ( l= 0; l <= L; l++)
{ value[l] = 0;
  rod_used[l] = - 1;
}
rod_used[0] = 0;
for ( j = 0; j < k; j++)
{  int lj = length[j];
  for ( l = lj ; l <= L; l++)
  { if ( rod_used[l - lj] >= 0)
    { double candidate = value[l - lj] + cost[j];
      if (candidate > value[l])
      { value[l] = candidate;
        rod_used[l] = j;
      }
    }
  }
}
```

We use the code segment above in a procedure *gen_col* which given $L$, and the arrays of lengths and costs generates columns. The procedure also has a parameter *number_of_patterns*; it tries to generate that many columns.

⟨*knapsack*⟩≡

```
list<pattern> knapsack(int L, const array<int>& length,
const array<double>& cost, double threshhold, int number_of_patterns)
{ int l,i,j;
  int k = cost.high() + 1;
  ⟨fill array of values⟩;
  list<pattern> result;
  ⟨compute result list⟩;
  return result;
}
```

We generate the list of results. As long as we have not generated a sufficient number of patterns we select the largest value from *value* and then generate a pattern for it.

⟨*compute result list*⟩≡

```
double max_value;
int max_index;
while ( number_of_patterns --)
{ max_value = 0; max_index = - 1;
  for (l = 0; l <= L; l++)
  if (value[l] > max_value)
  { max_value = value[l]; max_index = l; }
  if (max_value <= threshhold + eps ) break;
  array<int> pattern(0,k - 1);
  for (j = 0; j < k; j++)  pattern[j] = 0;
  value[max_index] = 0; // pattern is used
  while (max_index)
  { pattern[rod_used[max_index]]++;
    max_index - = length[rod_used[max_index]];
  }
  result.append(pattern);
}
```

**Lower Bounds:** In order to evaluate the quality of our solutions we need to compute lower bounds. We give two simple procedures. A trivial lower bound is given by $\lceil \sum_i n_i l_i / L \rceil$.

⟨*lower bounds*⟩≡

```
int lower_bound_1(int L, const array<int>& length, const array<int>& demands)
{ int sum = 0;
  int k = length.high() + 1;
  for (int i = 0; i < k; i++) sum + = length[i] * demands[i];
  return (sum % L == 0? sum/L : sum/L + 1);
}
```

We need to read in the test data. An example file consists of $k$ lines of three integers, $a$, $b$, and $c$ each. Such a line represents the demand to cut $2c$ pieces of length $a$ and length $b$ each. The following routine reads test data from a file named "file_name" and computes arrays *length* and *demand*. The length array is sorted in decreasing order.

We use a *d_array*<*int*, *int*> $D$ to do the work. For every triple $(a, b, c)$ we increase the counts of $a$ and $b$ by $2c$.

⟨*input routine*⟩≡

```
void read_from_file(string file_name,array<int>& length, array<int>& demand)
{
  file_istream in(file_name);
  int a,b,c;
  d_array<int, int> D(0); // all entries are initialized to zero
  while (in >> a)
  { in >> b; in >> c;
    D[a] + = 2*c; D[b] + = 2*c;
  }
  int k = 0;
  forall_defined(a,D) k++;
  /* $k$ is now the number of different length */
```

```
    length = array<int>(0,k - 1);
    demand = array<int>(0,k - 1);
    int i = k - 1;
    forall_defined(a,D)
    { length[i] = a;
      demand[i] = D[a];
      i--;
    }
  }
```

We need some simple heuristics for the cutting stock problem. Let us start with first fit decreasing. We go through the demands one by one and satisfy each one of them with the first raw that works.

We keep a *sortseq*<*int*, *list*<*pattern*> > *raws_in_the_game* in which we store pairs $(x, \mathit{pat\_list})$ where each pattern $a$ in *pat_list* cuts $l - x$ from our raw of length $L$.

⟨*heuristics*⟩≡

```
  typedef list<pattern> list_of_patterns;
  int first_fit_decreasing(int L, const array<int> & length,
  const array<int>& demand,list<pattern>& plan)
  {
    int i,j;
    int k = length.high() + 1;
    array<list_of_patterns> P(0,L);
    int raws_used = 0;
    for (i = 0 ; i < k ; i++)
    { int d = demand[i];
      int l = length[i];
      while (d--)
      {
        for (j = l; j <= L && P[j].empty(); j++) {} // no body
        if (j == L + 1)
        { pattern a(0,k - 1);
          for (int r =0; r <k ; r++) a[r] = 0;
          P[L].append(a);
          raws_used ++;
          j = L;
        }
        pattern a = P[j].pop();
        a[i] ++;
        P[j - l ].append(a);
      }
    }
    plan.clear();
    for (j = 0; j <=L; j++) plan.conc(P[j]);
    return raws_used;
  }
```

**The LP approach:** I maintain the LP twice, once at the level of my program and once inside CPLEX. CPLEX modifies its matrix but the matrix at the program level will stay constant (except for the addition of additional columns). The matrix $M$ has $k$ rows and *maxcols* columns. The first *cols* of these columns are currently used.

- setting up the LP given $L$, arrays *length* and *demand*.
- checking the LP solution for optimality and generating new columns if necessary.
- adding the newly generated columns to the LP.
- postprocessing the optimal LP-solution.

⟨*LP approach*⟩≡

```
array<pattern> M;
int maxcols;
int cols;
CPXLPptr lp = set_up_lp(L,length,demand,M,maxcols,cols);
double objval;
optimize(lp);
getobjval(lp,&objval);
int number_of_LPs_solved = 1;
cout << "in iteration "<<number_of_LPs_solved <<" we have an objective value "<< objval<<"\n";
list<pattern> pattern_list = check_optimality(lp,L,length);
int iter = 10;
while (!pattern_list.empty() && iter > 0)
{
  if (!add_columns(lp,pattern_list,M,maxcols,cols)) break;
  optimize(lp); number_of_LPs_solved ++;
  getobjval(lp,&objval);
  cout << "in iteration "<<number_of_LPs_solved <<" we have an objective value "<<(objval)<<"\n";
  if (objval <= lb ) iter--;
  pattern_list = check_optimality(lp,L,length);
}
post_process(lp,L,length,demand,M,maxcols,cols,lb,ub);
```

### Setting up the LP:

⟨*setting up the LP*⟩≡

```
CPXLPptr set_up_lp(int L, const array<int>& length, const array<int>& demand,
array<pattern>& M, int & maxcols, int& cols)
{
  int i,j;
  int k = length.high() + 1;
  maxcols = 500;
  M = array<pattern>(0,maxcols - 1);
  char* probname = "cutting_stock";
  cols = k;
  int cspace = maxcols; // maximal number of columns
  int rows = k;
  int rspace = rows; // number of rows never changes
  int nzspace = rspace * cspace;
```

```
int objsen = 1;
double* obj = new double[cspace];
for (j = 0; j < cols; j++) obj[j] =1;
double* rhs = new double[rows];
for (i = 0; i < rows; i++) rhs[i] = demand[i];
char* sense = new char[rows];
for (i = 0; i < rows; i++) sense[i] = 'E';
int* matbeg = new int[cspace];
int* matcnt = new int[cspace];
for (j = 0; j < cols; j++) matcnt[j] =rows;
int* matind = new int[nzspace];
double* matval = new double[nzspace];
int index = 0;
for (j = 0; j < cols; j++)
{ matbeg[j] = index;
  matcnt[j] = rows;
  int max_of_j = L/length[j];
  M[j] = array<int>(0,k - 1);
  for (i = 0; i < rows; i++)
  { matval[ index] =  M[j][i] = ( j == i ?  max_of_j : 0);
    matind[ index ] = i;
    index ++;
  }
}
double* lb = new double[cspace];
for (j = 0; j < cols; j++) lb[j] = 0;
double* ub = new double[cspace];
for (j = 0; j < cols; j++) ub[j] = INFBOUND;
double* rngval = NULL;
return loadprob(probname,cols,rows,0,objsen,obj,rhs,sense,
matbeg,matcnt,matind,matval,lb,ub,rngval,NULL,NULL,
NULL,NULL,NULL,NULL,
NULL,NULL,NULL,NULL,NULL,
NULL,NULL,NULL,NULL,NULL,NULL,
cspace, rspace,nzspace,
0,0,0,0,0);
}
```

**Checking optimality and generating new columns:** We have solved the current LP to optimality. We check whether the solution to the current LP is a global optimum and generate new columns if it is not.

Let $y^T = (y_0, \ldots, y_{k-1})$ be the dual optimal solution. The current solution is optimal if $y^T a \le 1$ for every pattern $a$. Converesly, all pattern $a$ with $y^T a > 1$ are worth to be added to the LP.

We extract the dual optimal solution from CPLEX with its routine *getpi* and move the dual variables to an array *cost*. Then we call the knapsack routine with a threshhold of 1. We ask for up to 10 patterns

⟨*checking optimality and generating new columns*⟩≡

```
list<pattern> check_optimality(CPXLPptr lp, int L, const array<int>& length)
{
  int k = length.high() + 1;
  double* pi = new double[k];
  getpi(lp,pi,0,k - 1); // we want dual variables for constraints 0 to $k - 1$
  array<double> cost(0,k - 1);
  for (int i = 0; i < k; i++) cost[i] = pi[i];
  return knapsack(L,length,cost,1,1);
}
```

**Adding columns to the LP** *pattern_list* is an array of new columns. We add them to our LP.

⟨*adding columns to the LP*⟩≡

```
bool add_columns(CPXLPptr lp, list<pattern> pattern_list,array<pattern>& M,
int maxcols, int& cols)
{
  int rows = (pattern_list.head()).high() + 1;
  int ccnt = pattern_list.length();
  if (cols + ccnt > maxcols)
  { cout << "two many columns in LP; stop before reaching optimality";
    return false;
  }
  int nzcnt = ccnt * rows;
  double* objx = new double[ccnt];
  int* cmatbeg = new int[ccnt];
  int* cmatind = new int[rows * ccnt];
  double * cmatval = new double[rows * ccnt];
  double* bdl = new double[ccnt];
  double* bdu = new double[ccnt];
  char** cname = NULL;
  int index = 0;
  for (int new_col_index = 0; new_col_index < ccnt; new_col_index++)
  { array<int> a = pattern_list.pop();
    M[cols++] = a;
    cmatbeg[new_col_index] = index;
    for (int i = 0; i < rows; i++)
    { cmatval[ index ] = a[i];
      cmatind[ index ] = i;
      index++;
    }
    bdl[new_col_index] = 0;
    bdu[new_col_index] = INFBOUND;
    objx[new_col_index] = 1;
  }
  addcols(lp,ccnt,nzcnt,objx,cmatbeg,cmatind,cmatval,bdl,bdu,cname);
  return true;
}
```

**Postprocessing the LP:** At this point we have determined an optimal solution to the LP. We first extract the solution and its objective value from CPLEX by means of its function *solution*.

Rounding the objective value upwards yields the LP lower bound for out integer program.

We also construct two integer solutions from the Lp solution. The first is obtained by rounding the LP-solution upwads. For the second solution we round downwards and then use our heuristic to place the remaining pieces.

⟨*postprocessing the LP solution*⟩≡

```
void post_process(CPXLPptr lp,int L, const array<int>& length, const array<int>& demand,
const array<pattern>& M, int maxcols, int cols,int& lb, int &ub)
{
int i,j;
int k = length.high()+1;
double objval;
double* x = new double[maxcols];
solution(lp,NULL,&objval,x,NULL,NULL,NULL);
//if (Yes("want to see patterns in fractional solution? type y   "))
{  for (j = 0; j<cols; j++)
   if (x[j] != 0)
        { cout << "the following pattern is used  " << x[j] << "  times\n";
          int sum = 0;
          for (i = 0; i <k; i++)
           {if (M[j][i] > 0) cout << "  " << M[j][i] <<" times length " << length[i] <<"\n";
           sum += M[j][i] * length[i];
           }
          cout << "the total length of the pattern is " << sum << "\n\n";

         }
}
int LPlb = (int)objval;
if (LPlb != objval) LPlb++;
cout <<"the LP lower bound is  " << LPlb <<"\n";
if (LPlb > lb) lb = LPlb;
/* rounding upwards, downwards, and optimistic */
array<int> x_down(0,cols-1);
array<int> x_up(0,cols-1);
array<int> x_optimistic(0,cols-1);
int raws_in_upward_rounding = 0;
int raws_in_downward_rounding = 0;
int raws_in_optimistic_rounding = 0;
int optimistic_count = 0;
cout << "the LP-solution and its rounded versions"  << "\n";
cout << " LP    down    up     optimistic   diff to down   diff to up" <<"\n";
for (j = 0; j<cols; j++)
{
  x_down[j] = (int)(x[j]);
  if (x_down[j] < 0) {cout << "warning: negative x[j]"; x_down[j] = x_up[j] = 0;}
  raws_in_downward_rounding += x_down[j];
  if (x_down[j] < x[j]) x_up[j] = x_down[j] + 1;
  else x_up[j] = x_down[j];
  raws_in_upward_rounding += x_up[j];
/* my shortest rods are 50 centimeters, i.e., no more than 12 fit into
```

```
a raw.  Also I use at most $k$ distinct patterns
and the fractional solution satisfies all demands with equality. Therefore,
rounding upwards any fractional variable which is less than $1/(k*12)$
from the next larger integer increases the number of pieces of any kind
by less than 1 and hence cannot oversatisfy any demand. */
  double eps_optimistic = 1.0/((double)(k*12));
  if ( ((double)x_up[j] - x[j]) <= eps_optimistic)
     {
      x_optimistic[j] = x_up[j];
      }
  else
     {
      x_optimistic[j] = x_down[j];
      }
  raws_in_optimistic_rounding += x_optimistic[j];
if (x[j] != 0)
cout << x[j] << "    " << x_down[j]  <<"  " <<x_up[j] << "  " << x_optimistic[j] << "   " << x[j] - x_down[j] << "  " << x_up[j] - x[j] <<"\n";
}


newline;
cout << "number of raws in upward rounding   " << raws_in_upward_rounding <<"\n";
if (raws_in_upward_rounding < ub) ub = raws_in_upward_rounding ;
/* satisfying the residual demands by a heuristic */
array<int> res_demands(0,k-1);
array<int> res_up_demands(0,k-1);
array<int> res_optimistic_demands(0,k-1);
for ( i = 0; i < k; i++)
{ res_demands[i] = res_up_demands[i] = res_optimistic_demands[i] = demand[i];
  for (j = 0; j < cols; j++)
    {res_demands[i] -= M[j][i] * x_down[j];
     res_up_demands[i] -= M[j][i] * x_up[j];
     res_optimistic_demands[i] -= M[j][i] * x_optimistic[j];
    }
   if (res_up_demands[i] > 0) error_handler(1," upward rounding incorrect");
   if (res_optimistic_demands[i] < 0)
     { cout <<"warning: optimistic rounding  does too much";
       res_optimistic_demands[i] = 0;
     }
}
list<pattern> cutting_plan;
for (j = 0; j < cols; j++)
{ for (int l = 0; l < x_down[j]; l++) cutting_plan.append(M[j]);}
list<pattern> res_plan;
list<pattern> res_optimistic_plan;
raws_in_optimistic_rounding +=
first_fit_decreasing(L,length,res_optimistic_demands,res_optimistic_plan);
if (raws_in_optimistic_rounding < ub) ub = raws_in_optimistic_rounding ;
cout << "number of raws in optimistic rounding   " << raws_in_optimistic_rounding <<"\n";
raws_in_downward_rounding += first_fit_decreasing(L,length,res_demands,res_plan);
cutting_plan.conc(res_plan);
/* check correctness  */
```

```
for ( i = 0; i < k; i++) res_demands[i] = demand[i];
list_item it;
forall_items(it, cutting_plan)
{ int sum = 0;
   for ( i = 0; i < k; i++)
      { sum += cutting_plan[it][i] * length[i];
        res_demands[i] -= cutting_plan[it][i];
      }
  if (sum > L) error_handler(1,"illegal Pattern");
}
if (cutting_plan.length() != raws_in_downward_rounding )
    error_handler(1,"length incorrect");
for ( i = 0; i < k; i++)
 if (res_up_demands[i] > 0) error_handler(1," downward rounding incorrect");

if (raws_in_downward_rounding < ub) ub = raws_in_downward_rounding ;
cout << "number of raws in downward rounding   " << raws_in_downward_rounding <<"\n";
cout << " I checked that downward and upward cutting plans indeed satisfy all demands\n";
}
```

**The class cutting stock:**

⟨*class cutting stock*⟩≡

```
class cutting_stock
{
  int L; // length of raw;
  int k; // number of different length
  array<int> length; // the different lengths
  array<int> demand; // and their demands
  public:
}
```

## 2 A Nicer Interface