Les Cahiers du GERAD ISSN: 0711–2440

A Fast and Accurate Algorithm for Stochastic Integer Programming, Applied to Stochastic Shift Scheduling

R. Pacqueau, F. Soumis,
L.-N. Hoang

G–2012–29

May 2012

A Fast and Accurate Algorithm for Stochastic Integer Programming, Applied to Stochastic Shift Scheduling

Rémi Pacqueau
François Soumis
Le-Nguyen Hoang

GERAD & Département de mathématiques et de génie industriel
École Polytechnique de Montréal
C.P. 6079, Succ. Centre-ville
Montréal (Québec) Canada, H3C 3A7

remi.pacqueau@polymtl.ca
francois.soumis@polymtl.ca
le-nguyen.hoang@polymtl.ca

May 2012

Les Cahiers du GERAD
G–2012–29

Abstract

Stochastic programming can yield significant savings over deterministic approaches. For example, the stochastic approach for the shift scheduling problem solved in [6] yields more than 15% savings on some instances. However, stochastic approaches always lead to very large problems (around 10 million IP variables in [6]), since a recourse must be computed for every scenario. There is no fast and exact method for solving such problems. In this article, the algorithm presented in [6] is improved in two ways: a Benders cuts dynamic management algorithm for the master problem and a multithreaded implementation to solve the subproblems. Those two improvements yield a heuristic able to solve a 10 million variables IP problem in less than 5 minutes, with a very good accuracy, and enables the resolution of larger instances. This algorithm uses general ideas that can easily be adapted to every problem that, in order to be solved, is split into a master problem and several subproblems: L-shaped method, column generation...

1 Introduction

Decomposition procedures, such as Benders decomposition [2], the L-shaped method [7], and Dantzig-Wolfe decomposition, have all been subject to recent developments and promising applications (shift scheduling, aircrafts and crew scheduling...). Although quite fast in LP, those decomposition methods are not able to solve to optimality very large MIP problems.

In order to solve a 10 million IP variables stochastic shift scheduling problem in [6], we developed an algorithm based on the L-Shaped method [7], a classical method for linear stochastic programming. The L-Shaped method could be considered an adaptation of Benders decomposition [2] with one recourse function for each random event (in the case of a finite probability space). This method is both accurate (0.01% from the optimal value in the studied toy-model and less than 0.2% from the optimal on larger instances) and fast (at most 3,200 seconds and down to 15 minutes, depending on the instances). CPLEX [4] used alone could only solve a 500,000 IP variable instance (25 scenarios) with a 0.2% optimality gap in 7,000 seconds (for the instance needing 3,200 seconds with our 500-scenario algorithm).

In this article, improvements on both the master problem and the subproblems are added to the heuristic process, so that it gets even faster. The implementation we used for our numerical experiments are certainly not the most efficient that could be used, but this work should be considered more a *proof of concept*, showing that it is possible to improve very quickly the most common decomposition methods.

This paper is organized as follows. In Section 2 the problem addressed in [6] and the heuristic procedure used to solve it are described. In Section 3, the improvements added to the heuristic are presented and the speed-up is measured over 10 different instances. The results are discussed in Section 4.

2 Problem presentation and heuristic description

The problem consists in scheduling shifts when the demand for employees is stochastic. A *first-stage* decision, the scheduling of full-time shifts, has to be taken before the demand is known (only the stochastic distribution of this demand is known). Once the demand is known, the company is allowed a *second-stage* decision (or *recourse*) to match the cover in employees to the actual demand. The possible recourses decisions are: hiring part-timers, changing the breaks schedules, asking full-timers for over-time and paying a penalty fee for not matching the demand.

A simple way to deal with a stochastic demand is to schedule full-time shifts according to the forecast demand and then to adapt the cover in employees thanks to the recourse decision. This is called the deterministic approach (see [3] for terminology). The problem studies another way of scheduling these shifts. Instead of minimizing the cost of the full-time shifts using only the forecast demand, the objective to minimize is the cost of the full-time shifts *and* the expected cost of the recourse associated to this full-time shifts assignment. This is called the stochastic approach. If the probability space is finite, the expected value can be written as a finite sum and the problem becomes a (very large) IP program.

2.1 Problem description

2.1.1 Notations

The sets used in this problem are described below:

P:	set of periods
J:	set of full-time shifts
JP:	set of part-time shifts
H:	set of overtime shifts
K:	set of breaks
Ω:	finite set of random events (scenarios)

The problem parameters are the following:

c_j: cost of full-time shift $j \in P$
cp_j: cost of part-time shift $j \in JP$
cs_h: cost of overtime shift $h \in H$
csc_p: cost of not covering demand in period $p \in P$
p_ω: probability of random event $\omega \in \Omega$
d_p^ω: demand for employees in period $p \in P$ if random event $\omega \in \Omega$ occurs
A_{jp}: equals 1 if full-time shift $j \in J$ matches period $p \in P$; 0 otherwise
AP_{jp}: equals 1 if part-time shift $j \in JP$ matches period $p \in P$; 0 otherwise
AK_{kp}: equals 1 if break $k \in K$ matches period $p \in P$; 0 otherwise
AH_{hp}: equals 1 if overtime shift $h \in H$ matches period $p \in P$; 0 otherwise
Q_{jk}: equals 1 if break $k \in K$ is within break window of full-time shift $j \in J$; 0 otherwise
R_{jh}: equals 1 if overtime shift $h \in H$ can be added to full-time shift $j \in J$; 0 otherwise

The decision variables are:

S_j: number of full-timers assigned to full-time shift $j \in J$
SP_j^ω: number of part-time employees assigned to part-time shift $j \in JP$ if event $\omega \in \Omega$ occurs
SH_h^ω: number of employees assigned to overtime shift $h \in H$ if event $\omega \in \Omega$ occurs
B_k^ω: number of full-timers assigned to break $k \in K$ if event $\omega \in \Omega$ occurs
X_{jk}^ω: number of full-timers assigned to full-time shift $j \in J$ and assigned to break $k \in K$ if event $\omega \in \Omega$ occurs
SC_p^ω: demand not covered in period $p \in P$ if event $\omega \in \Omega$ occurs

2.1.2 Formulation

The problem can then be formulated as follows (this is a stochastic adaptation of Aykin's formulation [1]):

$$\min \quad \sum_{j\in J} c_j S_j + \sum_{\omega\in\Omega} p_\omega \left[\sum_{j\in JP} cp_j SP_j^\omega + \sum_{h\in H} cs_h SH_h^\omega + \sum_{p\in P} csc_p \; SC_p^\omega \right] \tag{1}$$

$$\text{s.t.} \quad \sum_{j\in J} A_{jp} S_j + \sum_{j\in JP} AP_{jp} SP_j^\omega - \sum_{k\in K} AK_{kp} B_k^\omega + \sum_{h\in H} AH_{hp} SH_h^\omega + SC_p^\omega \geq d_p^\omega, \;\; \forall(\omega, p) \tag{2}$$

$$\sum_{k\in K} Q_{jk} X_{jk}^\omega - S_j = 0, \;\; \forall(\omega, j) \tag{3}$$

$$\sum_{j\in J} Q_{jk} X_{jk}^\omega - B_k^\omega = 0, \;\; \forall(\omega, k) \tag{4}$$

$$\sum_{h\in H} R_{jh} X_{jh}^\omega \leq S_j, \;\; \forall(\omega, j) \tag{5}$$

$$\sum_{j\in J} R_{jh} XH_{jh}^\omega - SH_h^\omega = 0, \;\; \forall(\omega, h) \tag{6}$$

$$S_j, \; SP_j^\omega, \; SH_h^\omega, \; B_k^\omega, \; X_{jk}^\omega, \; XH_{jh}^\omega \in \mathbb{Z}^+, \; SC_p^\omega \in \mathbb{R}^+, \; \forall(j, \omega, h, k) \tag{7}$$

The sum over Ω in the objective function is precisely the expected cost of the recourse. Constraint (2) ensures that the demand for employees is met in every period of each scenario. Constraints (3)–(4) ensure that every full-time shift is allocated a break and that every allocated break is within the time window of its associated shift. Constraints (5)–(6) ensure this for the overtime shifts.

2.1.3 Problem size

The studied period is one day, divided into 96 periods of 15 minutes. There are 65 possible shifts, 94 possible breaks, 166 part-time shifts. The state space is made of a finite number of *scenarios*, each corresponding to a possible demand on one day (therefore 96 demands: one for each 15-minute period). Every decision variable is integer, excepted the sub-cover. Table 1 shows the problem size for different number of scenarios considered. The more the scenarios, the more precise is the approximation of the stochastic distribution, but the harder to solve is the problem.

Table 1: Problem's size.

# Scenarios	# Variables	# Constraints
25	462 012	12 800
100	1 847 865	51 200
200	3 695 995	102 400
500	9 239 065	256 000
1 000	18 478 065	512 000

For reasons detailed in [6], the number of scenarios have been fixed to 500, thus making this shift-scheduling problem a 10-million IP variable program. Solving the LP relaxation of this problem with CPLEX used as a standalone solver takes already 4h45: a simple branch and bound algorithm is impossible.

2.2 The L-Shaped method

The heuristic is based on the L-shaped method, which is a classical method for solving stochastic linear programs. For a detailed presentation, see for example Birge and Louveaux [3].

The L-shaped method, like Benders decomposition, consists in splitting the problem into a master problem, containing the first-stage variables x (full-time shifts in this case) and a subproblem, containing the second-stage variables y (or recourse). Since the optimal values of the recourse variables depend only on x, the problem can be formulated as follows:

$$\min_{x \in X} \quad c \cdot x + \sum_{\omega \in \Omega} p_\omega Q^\omega(x) \tag{8}$$

Where X is the feasible space for the primary decision x and $Q^\omega(x)$ is the cost of the optimal recourse decision if first-stage decision x is taken and event ω occurs (with probability p_ω).

When there is no integrity constraint on the second-stage variables, every function Q^ω is a convex, piecewise linear function regarding to the first-stage decision x. These functions can therefore be expressed in problem (8) as a set of inequality constraints, to define problem (9)–(11):

$$\min_{x \in X} \quad c \cdot x + \sum_{\omega \in \Omega} p_\omega \theta^\omega \tag{9}$$

$$\theta^\omega \geq f_k^\omega(x), \ \forall k = 1, \ldots, K^\omega, \ \forall \omega \in \Omega \tag{10}$$

$$\theta \in \mathbb{R} \tag{11}$$

The principle of the L-shaped method is to start from an unconstrained master problem and to iteratively add these inequality constraints (also called Benders cuts) by solving the dual problems of the subproblems associated to the current first-stage solution (one subproblem for each scenario). Even if the number of Benders cut needed to exactly define the Q^ω functions is extremely large, only a few number of these cuts is actually needed to compute an optimum, since not all the regions of the first-stage feasible space need to be precisely described.

2.3 Heuristic description

The heuristic algorithm used to solve this problem in IP is described in Figure 1.

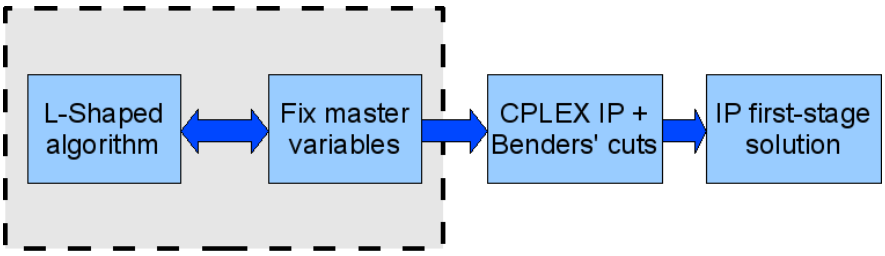


Figure 1: Algorithm description. New Benders cuts are iteratively generated as some first-stage variables are fixed. The processed is ended by a classical branch and bound scheme.

First, the LP relaxation is solved thanks to the L-shaped method. Then, every variable with a fractional part greater than 0.8 is fixed at the upper integer, and the problem is solved again thanks to the L-shaped method (the subproblems staying LP at all time). This operation is repeated as long as there are variables with fractional parts larger than 0.8. Even if the computation of the first LP relaxation can be quite long, fixing some variables and solving again is fast, since the Benders cuts are kept during all this process. Once the recourse function are well approximated, it is very fast to find a new optimum. The aim of fixing those variables to the upper integer is to accumulate as most Benders cuts as possible.

When there is no more first-stage variable to fix, the L-shaped method stops. At this points, the master problem contains less than 65 first-stage variables, 500 recourse variables (one for each scenario) and a large number of Benders cuts, that approximate the recourse function. This problem can be solved by CPLEX [4] using a simple branch and bound algorithm. This leads to an integer first-stage solution. The second-stage solution is not really important: it just has to be computed when the random event occurs. This algorithm is accurate when the cost of the recourse is small compared to the cost of the first-stage decision (the integrality gap in the subproblems can then be neglected), which is often the case in real-life applications.

This algorithm is accurate. On a 800,000-variable toy-model, 0.01% from the optimum. On the 500-scenario instances, the recourse cost (that is around 10% of the total cost) is underestimated by less than 0.2%. It is also fast, since it takes between 15 minutes and 1 hour depending on the stochastic distribution of the demand.

3 Improvements to the heuristic procedure

Figure 2 shows how the computational time is distributed into master problems and subproblems for different instances differing from their stochastic distribution only (same number of scenarios). See [6] for a more precise description of the instances.

Globally, computational time is a little more spent on solving the subproblems, however the master problem solving takes a non negligible amount of time. To speed-up our algorithm, we shall therefore introduce two improvements: the first one will speed-up the subproblems resolutions by dispatching them over several execution threads on multiprocessor machines, and the second one will speed-up the master problem resolutions by dynamically managing Benders cuts.

3.1 Reducing computational time for the subproblems using multithreading

Since every computer now comes with 2 or 4 cores, multithreading the subproblems resolution is a first way to quickly improve the computational time. This is possible since every subproblem for each scenario is totally independent of the others: on a 4-core computer, 125 subproblems can be assigned to each of the 4 cores. The number of Benders iterations *might* differ from the number of iterations needed when using one thread,

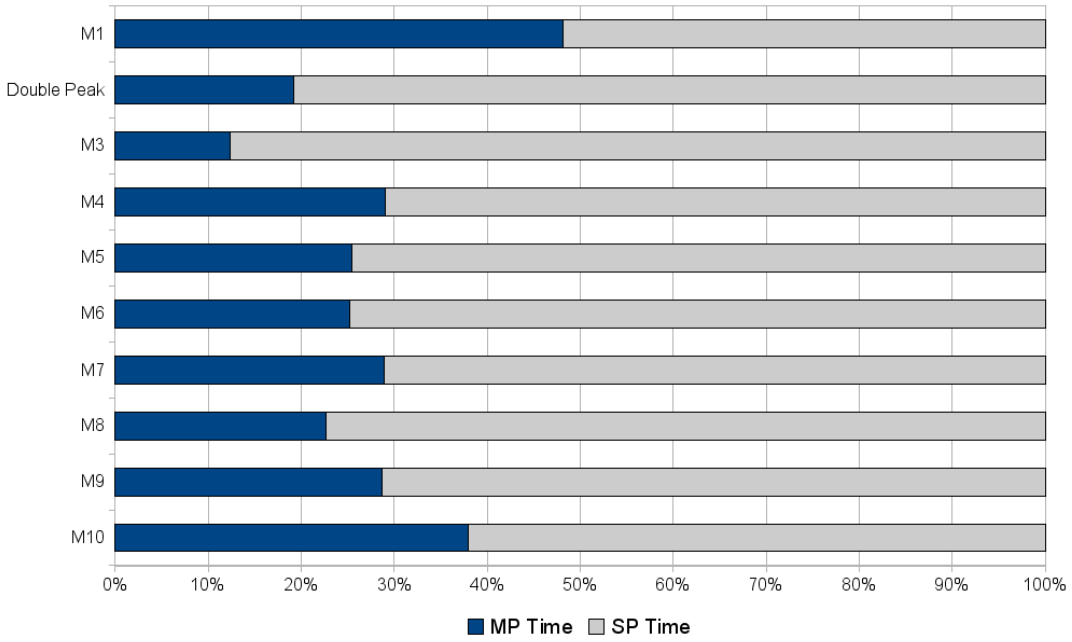


Figure 2: Repartition of the computational time between master problem and suproblems (single-threaded computations).

because of different re-optimizations (for example, to solve subproblem 126, CPLEX starts from problem 125 basis with one thread, and from a null basis with 4 threads). However, the procedure is still *deterministic*.

3.1.1 Procedure

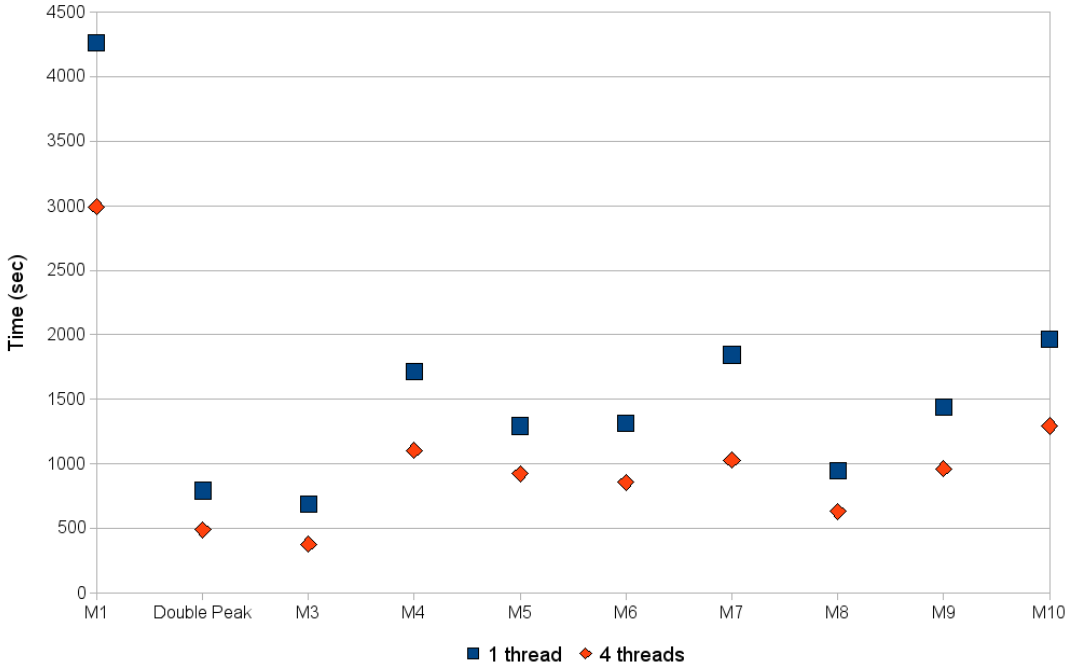
The implementation is done with CPLEX 12.2 (see [4]) and Java 1.6, using the native Java *thread* class and the *runnable* interface to implement multithreading (see [5] for documentation). All the computations are made on AMD Opteron 275 (2.2 GHz) servers, 8 GB RAM, running Linux. It should be noticed that, in order to be consistent, the single-thread computations has been done using the same class and data structure as the multithreaded ones. However, another single-threaded implementation we developed yields better computational time, since some redundancies are eliminated (10-15% improvements compared to the single-threaded implementation we used in this paper).

3.1.2 Results

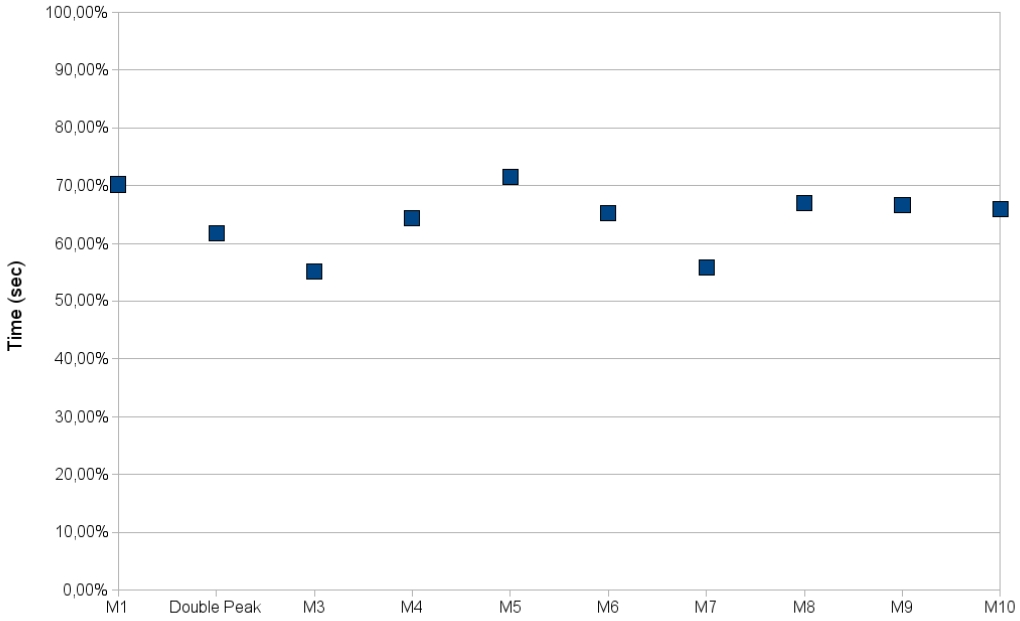
The results are presented in Figure 3. Numerical results can be found in Appendix A. It can be noticed that the absolute and relative new computational time greatly depends on the instance (55 to 71%). This is explained by the fact that the computational time is not used in the same way for every instance (Figure 2).

Since only the subproblems can be multithreaded, the instances with computational time mainly spent on the subproblems, like M3, benefit the most of this improvement. In instances whith computational time half split between the master problem and the subproblems, like M1 or M10, the improvements is sensible but not as interesting as in the M3 case.

However, the improvement compared to the *only* subproblem computational time with 4 cores is between 45% for M7 (581 seconds instead of 1,308) and 63% for M5 (607 instead of 983). The other instances are showing an around 50% relative subproblems computation time. This is more than the 25% that could theoretically be obtained since 4 cores are used instead of one, but a lot of other different factors affect the computational time (accessing memory, threads managing...).



(a) Computational time using 1 or 4 threads



(b) Relative computational time using 4 threads

Figure 3: Improvements in terms of computational time provided by the use of multithreading. Figure (a) shows the computational time in seconds, and Figure (b) shows the multithread computational time as a percentage of the single-thread computational time.

3.2 Reducing computational time for the master problems using a dynamic cut management algorithm

Now that the computational time of the subproblem has been divided by around two, the focus has to be made on improving the master problem computational time. To better understand why the master problem takes sometimes a long time to be solved, Table 2 shows the number of L-shaped iterations needed to solve the problem.

Table 2: L-shaped method iterations needed to solve the problem (LP relaxation *and* branching process).

Instance	M1	DP	M3	M4	M5	M6	M7	M8	M9	M10
Iterations	118	33	30	59	50	50	64	39	57	64

It can be noticed that in the longest to solve instances, like M1 and M10, the number of needed iterations is far larger than the number needed to solve fastest instances like M3.

The main issue is that 500 Benders cut are added to the master problem at each iteration, meaning that in M1, the master problem has accumulated 44,000 Benders cut, making it large and long to solve. Thus, although the master problem is fast to solve at the beginning, it gets quite large after 50 iterations, making every extra iteration a little longer than the previous one. The size of the master problem when the number of iterations increases dramatically slows down the resolution process.

Although all the cuts are theoretically needed, a lot of them are useless, since they might describe regions of the functions that are not needed anymore. Deleting those useless cuts in order to make the master problem smaller will have sensible effects on the resolution time.

Remark 1 In our algorithm, the problems we solve are LP at all times, even if some variables are fixed to the upper integer during the branching process. Each problem resulting from the branching process is solved with our improved algorithm. Therefore, we shall detail here only how to improve the linear L-Shaped method, keeping this method is used along our whole resolution procedure.

3.2.1 Procedure

The first idea that could come to mind to make the problem smaller would be to delete every inactive cut at each iteration. This approach is very likely to make the algorithm cycle. During the L-Shaped method, the current first-stage solution generally oscillates between several regions of the functions. If we simply delete the inactive cuts at each iteration, the algorithm will constantly go to a region, build the cuts, go to another region, build the cuts and delete the one of the first region, go back to the first region... A similar idea would be to delete every cut that has been inactive for more than K iterations. This does not solve the cycling problem: the algorithm could still enter into cycles longer than K. We chose a different approach.

The main idea of our approach is to try to impose a maximum number M of Benders cuts for the master problem, and not to delete any cut before this limit is reached. When M is reached, all the cuts that have been inactive for more than K iterations are deleted. If M is too low, it is multiplied by 1.5 (M-factor). We consider M too low when the algorithm has to delete cuts during 3 consecutive iterations (meaning that the number of cuts stays very close to M even after deletion).

Each Benders cut is stored onto an *ArrayList* (see [5] for documentation) and is associated to an *inactivity counter*. Every time a Benders cut is inactive (it would be considered inactive if its associated slack variable is larger than a given ϵ, typically 10^{-10}), this counter increases. It is set to 0 at an iteration if the Benders cut is active in the current solution. Algorithm 1 describes the procedure.

Algorithm 1 also includes a safety that checks if, after having deleted some cuts, the value of the new problem is actually greater than the old one (this should be the case in a classical L-shaped method). If this is not the case, the algorithm goes on without deleting any cut until the current value gets bigger than the previous solution value (which is the largest computed value since the L-Shaped method for a minimization problem yields a series of non-decreasing values, up to the problem optimal value).

Algorithm 1 Dynamic cut management algorithm

```
Solve the master problem, get its value z^k
if z^k = z^{k-1} then
  Stop: optimal solution found
else if z^k > z^{k-1} then
  Update the inactivity counter of each cut
  if # Benders cuts > M then
    if # Consecutive removals ≤ 3 then
      Remove all cuts that have been inactive for K or more iterations
      Reload master problem
    else
      M ← 1.5 × M
    end if
  end if
else
  Do not try to delete any cut
  Run standard L-Shaped method until z^k > z^{k-1}
end if
Solve the slave subproblems
Add the generated Benders cut to the master problem
Go to line 1
```

Theorem 1 *The dynamic cut management algorithm applied to the LP relaxation of the problem converges in a finite time to an optimum.*

Proof. The number of optima of the master problem can have is finite, since the power set of the set of Benders cuts is also finite (because there is a finite set of Benders cuts, see [3]). Now, we always move from an optimum to a strictly better one (except if global optimum is reached) in a finite time: if the optimum does not increase because of a cut removal, the L-shaped method, which is proven to converge in a finite time, is used until so. Hence, strictly improving an optimum is done in a finite time. As the number of these optima is finite, we have proven the result. □

The value returned by the algorithm is still optimal (for the LP relaxation), since the stopping criterion is the same. If some useful cuts are deleted, they will just be regenerated by the L-shaped method during a further iteration. What could happen at worth with this algorithm is that extra iterations will be needed if some useful cuts are deleted.

Corollary 1 *The improved algorithm (branching procedure, multithreading, dynamic cut management) converges in a finite time.*

3.2.2 Results

All the numerical results can be found in Appendix A. Figure 4 shows that the algorithm, in instance M1, manages to keep a small number of cuts during the whole resolution process: up to 15,000 when $K = 10$, but no more than 8,000 during the first LP resolution (iterations 1 to 88). The standard L-Shaped method would yield a 44,000-cut problem after 88 iterations, and 109,000 at the end of the algorithm.

Other values of K (the minimum inactivity counter before deleting a cut) and ϵ (the minimum slack to consider a constraint as inactive[1]) were tested. The results are presented in Tables 3 and 4. Original time was 4,251 seconds with 1 thread, 2,991 seconds with 4 threads.

It can be seen in Tables 3 and 4 that computational time is not very sensitive to the values of K and ϵ. The M-factor has been tested on instance M1 and does not have a sensible effect also. The parameters will

[1]For instance M1, the difference in the objective between the last and penultimate iterations of the LP relaxation is 2.10^{-12}.

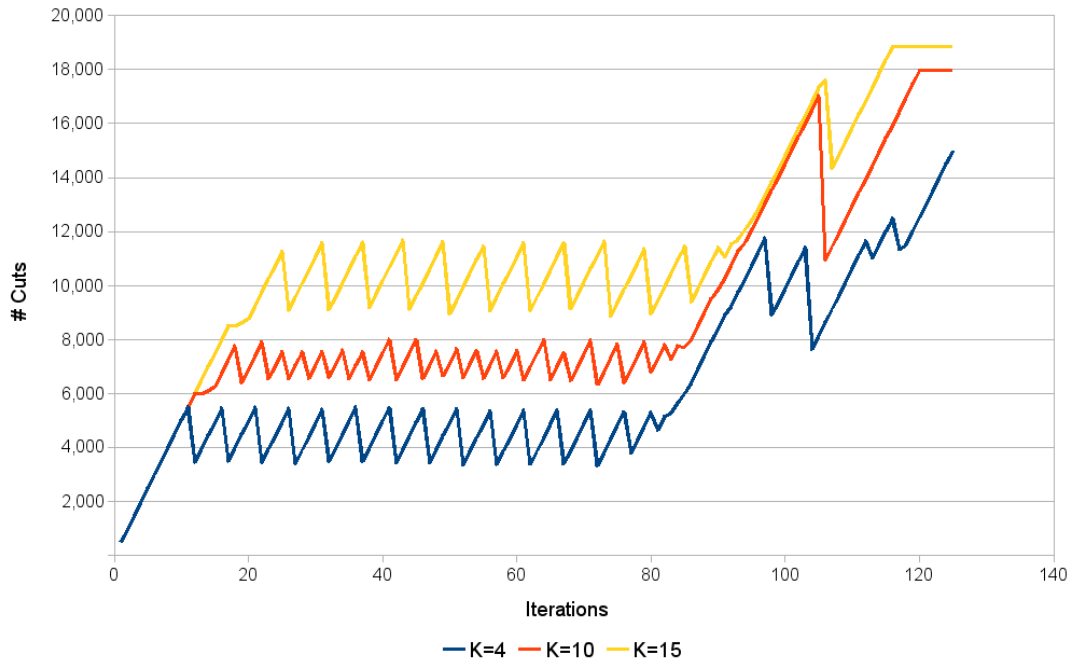


Figure 4: Number of cuts during the resolution process for several values of K. Instance: M1.

Table 3: Incidence of K on the resolution time and the total number of iterations (L-shaped and heuristic). $M = 5,000,\ \epsilon = 10^{-10}$.

K	15	10	8	6	4
Iterations	116	120	123	122	125
Time (sec)	1,873	1,639	1,832	1,782	1,783

Table 4: Incidence of ϵ on the resolution time and the total number of iterations (L-shaped and heuristic). $M = 5,000,\ K = 10$.

ϵ	10^{-6}	10^{-10}	10^{-13}
Iterations	124	120	123
Time (sec)	1,760	1,639	1,732

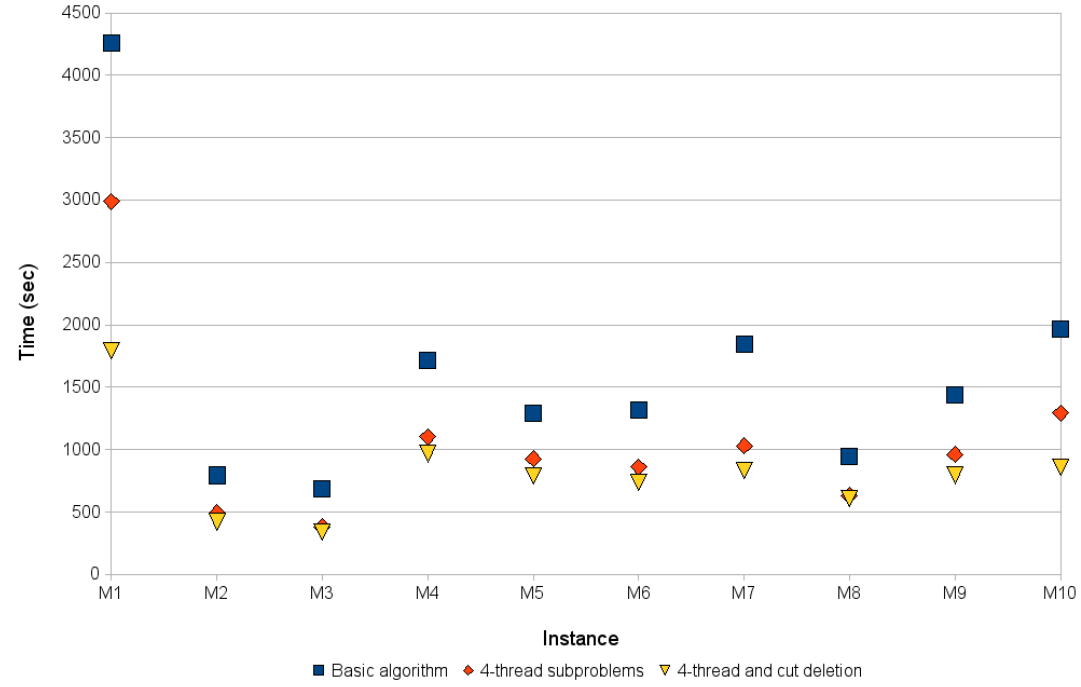
be fixed arbitrarily: K to 10, M to 5,000, M-factor to 1.5 and and ϵ to 10^{-10}. M has a minor role: since the minimal inactivity counter is 10, 10 iterations must have been achieved to remove some cuts. Since there are 500 scenarios, there are already 5,000 cuts after 10 iterations.

We chose those parameters since they tend to be "reasonable", thus might work well on other instances or other problems.

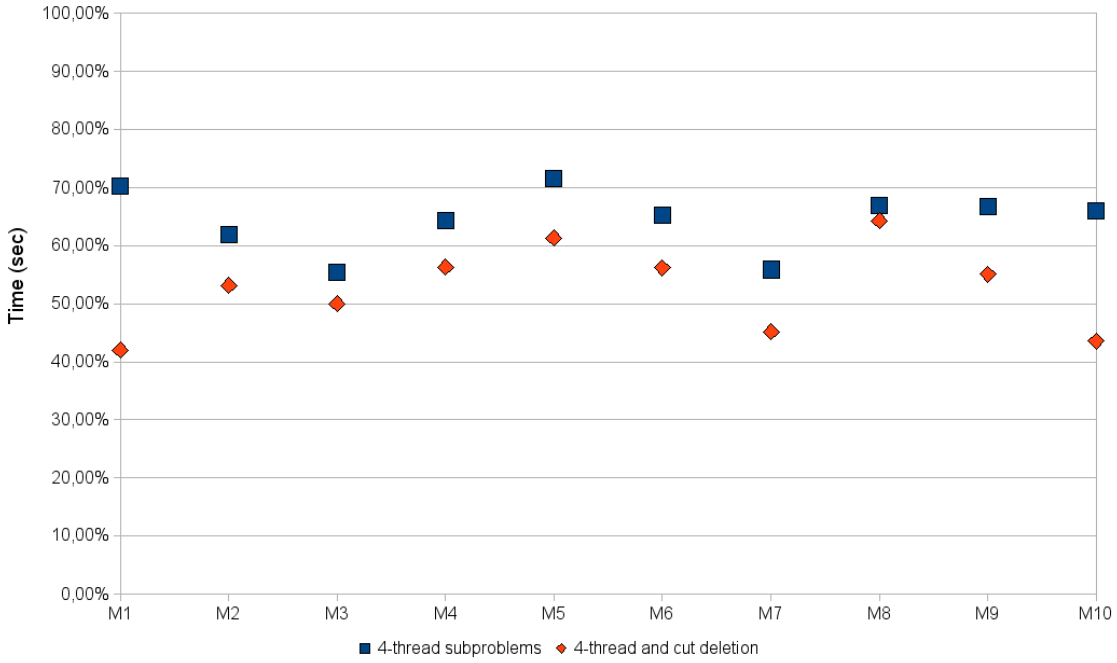
Every instance from [6] was solved with the new improved algorithm (multithreading and dynamic cut management) and the previously fixed parameters. Results are presented Figure 5. For every instance, the algorithm safety is triggered no more than 2 or 3 times, which has a quite negligible inpact on the computational time since it takes generally more than 100 iterations to compute a solution to the problem.

To evaluate the influence of dynamic cut management in the master resolution process, Figure 6 shows the relative master problem computation time using dynamic cut management regarding to the classical L-Shaped procedure computational time.

On the originally longest instances, M1 and M10, the master problem computational time is divided by 4 and 3.



(a) Computational time for the 3 algorithms



(b) Relative computational time for the two improved algorithms

Figure 5: Computational times (absolute and relative) for all the instances using the original algorithm, the multithreaded version and the multithreaded + dynamic cut management version. For this last one, $K = 10$, $\epsilon = 10^{-10}$, $M = 5,000$.

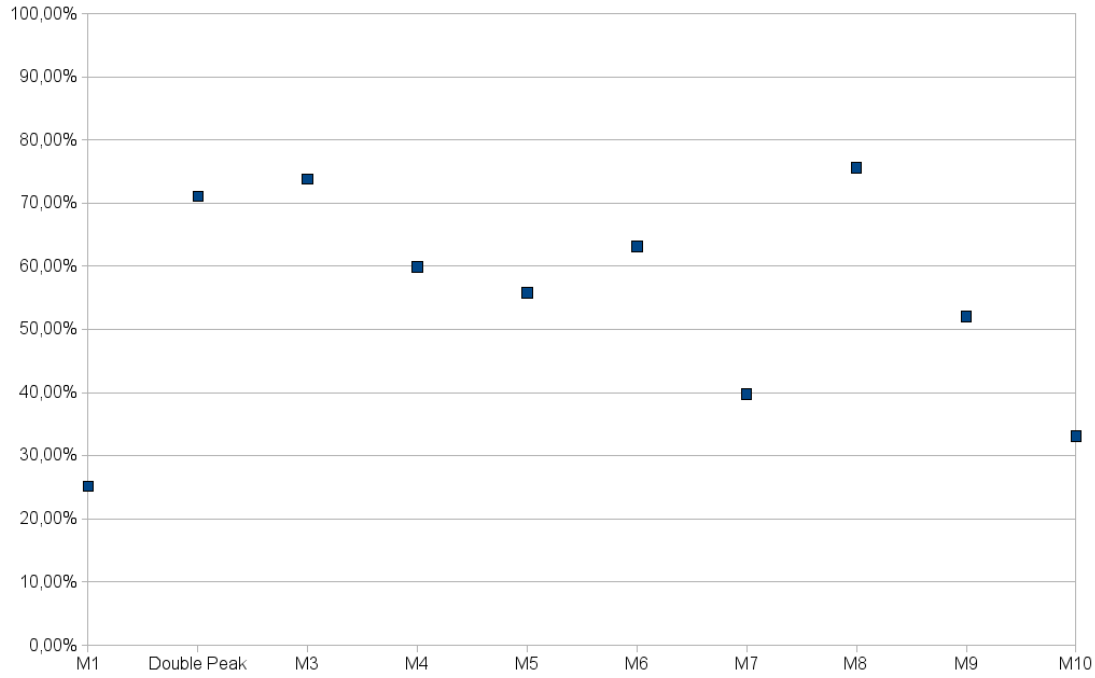


Figure 6: Master problem computational time using dynamic cut management relatively to master problem computational using the standard L-shaped procedure.

4 Discussion and larger instances

The results are conclusive. Using 4 threads to solve the subproblems yields significant reductions of the computational times (up to 45% overall, 56% for the subproblem computational time). However, speeding up the subproblems solutions does have a less sensible effect on the overall computational time for the longest instances, since a large part of the computational time is spent on solving the master problem, that gets larger at each further iteration.

The dynamic cut management algorithm tackles this problem at its source: the size of the master problem. By deleting "old" cuts, it manages to keep a reasonable size for the master problem, especially during the first LP resolution. For example, in instance 1 (Figure 4), the number of cuts at the end of the LP resolution (88 iterations, the further iterations corresponding to the branching procedure) is around 7,500, whereas 88 iterations yield a master problem containing 44,000 cuts with the standard L-Shaped method. Figure 6 shows that this algorithm needs up to 4 times less computational time on the master problem than the standard L-Shaped procedure. Tables 3 and 4 show that the parameters M, K and ϵ do not have a significant influence on the speed of our algorithm.

Globally, our approach divides by around 2 the overall computational time (savings from 40 to 60%) for the 500-scenario instances. The gap is slightly bigger on the originally slow to solve instances, like M1, M7 or M10. This is explained by the fact that the cut management algorithm yields better improvements to the master problem (gap up to 75% for M1) than the use of multithreading on the subproblems (up to 57%). Logically, instances with longer to solve master problems benefit the most from our approach.

We also solved a 1,000-scenario instance (near 20 million IP variables, same stochastic distribution as M1). Results are presented Table 5. Computational time of master problems (MP), subproblems (SP) and total time are in second. "Ratio" corresponds to the improved algorithm computational time expressed as a percentage of the standard L-Shaped computational time (with our branching procedure). CPLEX solved only the LP relaxation, whereas the two L-Shaped algorithm solved the problem with IP first-stage variables.

Table 5: Computational time for a 1000-scenario instance (20 million IP variable, same stochastic distribution as M1).

	Iterations	Benders Cuts	MP Time	SP Time	Total
CPLEX LP	/	/	/	/	107,693
L-Shaped	104	104,000	9,068	4289	13,357
Improved	109	30,825	1,218	2,006	3,227
Ratio (%)	105	30	13	47	24

The improvements divide the total computational time by more than 4 compared to the standard L-Shaped method combined with our branching strategy (54 minutes instead of 3h42). The master problem time is divided by around 10. CPLEX alone needs 30 hours to solve the LP relaxation only. This shows that the method we propose enables solving larger problems keeping a reasonable computational time, and that the relative gap between our method and the standard L-Shaped algorithm increases with the number of scenarios.

The reduction of both master and subproblems computational times also makes this approach stable: computational time is always divided by around 2 in the 500-scenario instances. However the gap repartition between master and subproblems depends on the instance. For fast to solve instances, most of the gap lies in the subproblem resolution, whereas for longer to solve instances, most of the gap lies in the master problem resolution.

5 Conclusion

The main contribution of this paper is to present a fast and accurate algorithm for stochastic integer programming. This algorithm features 3 enhancements to the linear L-Shaped method:

1. A branching strategy that uses Benders cuts as an approximation of the recourse function
2. The use of multithreading for solving the subproblems
3. A dynamic cut management algorithm to speed up the master problem resolutions

This algorithm solved a 10 million IP variables in 1,639 seconds (27 minutes, instance: M1), whereas the L-Shaped method with only the branching strategy needs 4,251 seconds (1h10) and CPLEX alone needs 17,085 seconds (4h45) to solve the LP relaxation only (which is also computed at the beginning of our algorithm). The overall time gap for all the 500-scenario instances tested is around 50% (from 35.7% to 61.5 %), compared to the algorithm using only the branching strategy. The precision of the algorithm is discussed in [6].

Our method also works well with larger instances: the 1000-scenario instance we tried, based on the same stochastic distribution as M1, was solved in 4 times less time than using the standard L-Shaped method (with our branching strategy).

Moreover, both the use of multithreading and the dynamic management algorithm can be used to improved the standard linear L-Shaped method (without any branching process). The solution is still optimal. The dynamic cut management is built so that the algorithm will also end in a finite time. This is a simple way to speed up the resolution process. Dynamic cut management can also be applied to simple Benders decomposition [2].

Appendix

A Computational time: Numerical results

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
T1	4,251	794	684	1,717	1,294	1,317	1,845	944	1,441	1,963
T2	2,991	491	379	1,105	925	860	1,031	632	962	1,294
T3	1,639	422	342	967	793	740	834	607	794	856
R2	70.2	61.8	55.4	64.4	71.5	65.3	55.9	67.0	66.8	65.9
R3	38.5	53.2	50.0	56.3	61.3	56.2	45.3	64.3	55.1	43.6

Figure 7: Computational time (in seconds) for the IP solution using the algorithm from [6] without any improvement (T1), using 4 threads for the subproblems (T2) and using both multithreading and the dynamic cut management algorithm (T3). R2 is the ratio between T2 and T1 (in %) and R3 is the ratio between T3 and T1 (in %). Relative improvements in % are $100 - R2$ and $100 - R3$.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
T4	2,202	639	598	1,215	962	983	1,308	729	1,024	1,216
T5	1,122	309	282	618	607	565	581	447	579	598
R	51.0	48.4	47.2	50.8	63.1	57.5	44.4	61.3	56.5	49.2

Figure 8: Computational time (in seconds) of the subproblems (LP relaxation *and* branching) using one thread (T4) and 4 threads (T5), and ratio in % (R). Relative improvement in % is $100 - R$.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
T6	2,058	155	86	502	332	334	537	215	417	747
T7	517	113	60	349	186	175	253	160	215	258
R	25.1	72.9	69.8	69.5	56.0	52.4	47.1	74.4	51.6	34.5

Figure 9: Computational time (in seconds) of the master problems (LP relaxation *and* branching) using the standard L-Shaped algorithm (T6) and the dynamic cut management algorithm (T7), and ratio in % (R).Relative improvement in % is $100 - R$.

References

[1] Aykin, T. Optimal shift scheduling with multiple break windows. Management Science 1996;42:591–602.

[2] Benders, J.F. Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik 1962;4:238–252.

[3] Birge, J.R., Louveaux, F. Introduction to Stochastic Programming. Springer, 1997.

[4] IBM Ilog. CPLEX Optimizer.

[5] Oracle. Java SE 6 documentation. 2011.

[6] Pacqueau, R., Soumis, F. Shift scheduling under stochastic demand. European Journal of Operations Research 2011.

[7] Van Slyke, R., Wets, R. L-shaped linear programs with applications to optimal control and stochastic programming. SIAM J Appl Math 1969;17(4):638–663.