Algorithm 928: A General, Parallel Implementation of Dantzig–Wolfe Decomposition

JOSEPH RIOS, National Aeronautics and Space Administration

Dantzig–Wolfe Decomposition is recognized as a powerful, algorithmic tool for solving linear programs of block-angular form. While use of the approach has been reported in a wide variety of domains, there has not been a general implementation of Dantzig–Wolfe decomposition available. This article describes an open-source implementation of the algorithm. It is general in the sense that any properly decomposed linear program can be provided to the software for solving. While the original description of the algorithm was motivated by its reduced memory usage, modern computers can also take advantage of the algorithm's inherent parallelism. This implementation is parallel and built upon the POSIX threads (pthreads) library. Some computational results are provided to motivate use of such parallel solvers, as this implementation outperforms state-of-the-art commercial solvers in terms of wall-clock runtime by an order of magnitude or more on several problem instances.

Categories and Subject Descriptors: G.1.6 [**Numerical Analysis**]: Optimization—*Linear programming;* G.4 **[Mathematical Software]**—*Parallel and vector implementations*

General Terms: Algorithms

Additional Key Words and Phrases: Linear programming, optimization, parallel implementations

ACM Reference Format:
Rios, J. 2013. Algorithm 928: A general, parallel implementation of Dantzig–Wolfe decomposition. ACM Trans. Math. Softw. 39, 3, Article 21 (April 2013), 10 pages.
DOI: http://dx.doi.org/10.1145/2450153.2450159

1. INTRODUCTION

Researchers in several domains have long recognized the importance of decomposition techniques for large-scale linear programs. Dantzig–Wolfe (DW) decomposition is the original decomposition approach and the predecessor to all column-generation schemes. Originally proposed to mitigate concerns with limited main memory [Dantzig and Wolfe 1960], the method is inherently parallel and can be implemented to take advantage of clusters of machines or multiple cores on a single machine.

DW is provably optimal for linear programs [Dantzig and Wolfe 1960] and is useful in several integer programming schemes (see, for example Vanderbeck [2000, 2006] or Barnhart et al. [1998]). The approach has been used in domains as diverse as power system management [Fu et al. 2005], airline crew pairing [Desaulniers et al. 1997], economic moral-hazard problems [Prescott 2004], traffic flow management [Rios and Ross 2010], and job-shop scheduling [Gélinas and Soumis 2005]. Yet, despite the

Author's address: J. Rios, NASA Ames Research Center, Mail Stop 210-15, Moffett Field, CA 94035; email: joseph.l.rios@nasa.gov.

DOI: http://dx.doi.org/10.1145/2450153.2450159

positive results reported through the use of DW, a general implementation of the algorithm (commercial or otherwise) does not exist.

In the 1980s, James Ho was active in developing an implementation of DW called DECOMP. The original implementation [Ho and Loute 1981] was tied to a no-longer available commercial linear programming solver from IBM. A fully-documented implementation (written in FORTRAN) using Stanford's LPM1 linear programming code was published eight years later [Ho and Sundarraj 1989]. More recently, James Tebboth implemented DW and tested it on a series of input problems [Tebboth 2001]. He analyzes the results in great detail and offers insights as to the classes of problems best suited for application of DW. Though Tebboth's dissertation describes his implementation and even provides a user's manual as an appendix, the software is not publicly available. A search for Dantzig–Wolfe implementations may turn up examples described in the modeling languages AMPL [Fourer et al. 2002] or GAMS [GAMS Development Corporation 2010], but there is no general implementation in any language.

The motivation for this article is to describe a general, parallel implementation of Dantzig–Wolfe decomposition under an open source license in a language with demonstrated longevity (the entire implementation is in C). The software described here is written in C and uses the GNU Linear Programming Kit [Makhorin 2010] for the general optimization library. By releasing the code as open source it is hoped that future researchers in various domains will have access to a stable platform from which to begin experimentation without the need to implement the algorithm from scratch.

The remainder of this article is organized as follows. Section 2 describes the DW algorithm. Next, Section 3 details the implementation of the software in terms of parallelization and synchronization. The performance of this DW implementation is then measured and compared against a state-of-the-art LP solver in a parametric study and with a large, practical example. Results of these performance studies are provided in Section 4. Concluding remarks are provided in Section 5.

2. DANTZIG–WOLFE DECOMPOSITION

Dantzig–Wolfe decmoposition (DW) is an approach to solving linear programs (LP) of a special form. Dantzig and Wolfe motivated the development of their approach through an example of a company with several divisions wherein there are sets of constraints unique to each division and a set of constraints common to all divisions through shared resources. This motivates solving separate problems for each of the company's divisions while making certain to satisfy the common (or "connecting") constraints and optimize the company's overall objective.

The remainder of this section will provide details on the DW approach. First, we describe the form of the constraint matrix necessary for application of DW. Next, the details of the algorithm itself are discussed. Then the specific design choices used when implementing the algorithm are given.

To apply DW to a given problem, it must exhibit a *block-angular form*. To gain some perspective on this form, we borrow some notation from Bertsimas and Tsitsiklis [1997], noting that bold lowercase variables represent vectors and bold capital letters represent matrices. Begin with a linear program (LP) of the form:

$$\begin{aligned}
\text{Minimize:} \quad & \mathbf{c}_1'\mathbf{x}_1 + \mathbf{c}_2'\mathbf{x}_2 + \cdots + \mathbf{c}_l'\mathbf{x}_l && (1)\\
\text{Subject to:} \quad & \mathbf{D}_1\mathbf{x}_1 + \mathbf{D}_2\mathbf{x}_2 + \cdots + \mathbf{D}_l\mathbf{x}_l = \mathbf{b}_0 && (2)\\
& \mathbf{F}_1\mathbf{x}_1 = \mathbf{b}_1 && (3)\\
& \mathbf{F}_2\mathbf{x}_2 = \mathbf{b}_2 && (4)\\
& \vdots\\
& \mathbf{F}_l\mathbf{x}_l = \mathbf{b}_l, && (5)
\end{aligned}$$

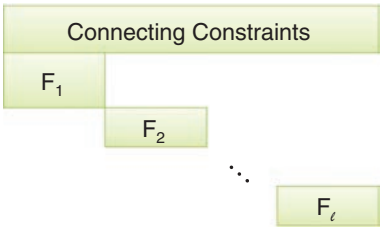


Fig. 1. Block angular form of the constraint matrix required for Dantzig–Wolfe decompostion.

where $\{\mathbf{x}_i | i \in [1 \dots l]\}$ are the decision variables. Visually, the constraint matrix defined in Equations (2) through (5) is described in Figure 1. Constraints (2) are called *connecting constraints* since they involve variables from various subproblems. Assuming each set $P_i = \{\mathbf{x}_i \geq 0 | \mathbf{F}_i \mathbf{x}_i = \mathbf{b}_i\}$ is bounded,[1] each P_i can be described as a convex combination of its extreme points. Let J_i be the set of extreme points for subproblem $i \in [1, \dots, l]$ and denote individual extreme points as $\mathbf{x}_i^j, j \in J_i$. It follows that

$$\mathbf{x}_i = \sum_{j \in J_i} \lambda_i^j \mathbf{x}_i^j, \tag{6}$$

where

$$\sum_{j \in J_i} \lambda_i^j = 1 \ \ \forall i, j$$
$$\lambda_i^j \geq 0 \ \ \forall i, j.$$

Given the description of $\mathbf{x}_i$ from Equation (6), a substitution can be made into the LP described in Equations (1) to (5) resulting in the following *full master program*.

$$\begin{aligned}
\text{Minimize:} \quad & \sum_{i=1}^{l} \sum_{j \in J_i} \lambda_i^j \mathbf{c}_i' \mathbf{x}_i^j \\
\text{Subject to:} \quad & \sum_{i=1}^{l} \sum_{j \in J_i} \lambda_i^j \mathbf{D}_i \mathbf{x}_i^j = \mathbf{b}_0 \\
& \sum_{j \in J_1} \lambda_1^j = 1 \\
& \sum_{j \in J_2} \lambda_2^j = 1 \\
& \quad \vdots \\
& \sum_{j \in J_l} \lambda_l^j = 1,
\end{aligned} \tag{7}$$

where $\{\lambda_i^j | \forall j \in J_i, \forall i \in [1 \dots l]\}$ are the decision variables. Notice that these decision variables are actually the weights on the extreme points of each P_i.

If the original formulation (Equations (1) to (5)) had m constraints (rows) and n variables (columns), then the master formulation has only $m_0 + l$ constraints (where

[1]This assumption is not necessary for DW decomposition in general, but it simplifies the mathematical discussion.

m_0 is the number of coupling constraints and m_0 is typically significantly less than m depending on the model and problem instance), but an exponentially larger number of columns since there is a column for each extreme point generated by the sets of constraints illustrated in Equations (3) to (5).

Since the vast majority of the λ variables are valued zero at any given iteration, most columns are irrelevant (that is, nonbasic) to the master. This leads to the heart of the decomposition algorithm: *column generation*. Only potentially useful columns are added to a so-called reduced master problem. For each P_i, an independent LP is created. The constraints of each of those subproblems are represented by P_i. The complete formulation of the subproblem is

$$\begin{aligned} &\text{Minimize:} && (\mathbf{c}_i' - \mathbf{q}'\mathbf{D}_i)\mathbf{x}_i && (8)\\ &\text{Subject to:} && \mathbf{F}_i\mathbf{x}_i = \mathbf{b}_i, \end{aligned}$$

where $\mathbf{q}$ is the dual variable vector associated with the connecting constraints. This vector is made available as usual through the simplex algorithm applied to the reduced master problem. Objective (8) forces the subproblem to provide a variable to the reduced master problem with the greatest reduced cost, since the reduced cost is defined as

$$(\mathbf{c}_i' - \mathbf{q}'\mathbf{D}_i)\mathbf{x}_i^j - r_i, \tag{9}$$

where r_i is the dual variable for the connectivity constraint associated with subproblem i. The value of r_i is known to the reduced master problem through its previous simplex iteration in the same way $\mathbf{q}$ was discovered. Recall that in the traditional simplex algorithm, each iteration has a step to choose a variable to enter the current basis that might improve the objective function (a variable with negative reduced cost in the case of a minimization problem). Using DW, the search for such a variable is assigned to separate LPs (the subproblems). If the reduced cost described by Expression (9) is negative, the variable enters the reduced master formulation. When the reduced cost is non-negative, the inclusion of that decision variable will not improve the reduced master's objective and the column is not added to the formulation. Since there are a finite number of corner points for each subproblem and the subproblems are solved with some form of the simplex method, the DW algorithm is guaranteed to terminate.

Assuming there are n subproblems in a DW implementation, at any iteration of the algorithm there are up to n potential columns to add to the reduced master formulation. One must establish a policy governing which (if any) of the columns are to be included in the reduced master. Some options include choosing the column(s) with the greatest reduced cost(s), choosing the first available column(s) (if all subproblems are being solved in parallel), or choosing all available columns that will improve the objective. The implementation for this study accepts all columns with strong potential to improve the master's objective (all columns with negative reduced cost).

Since there will be up to n columns entering the reduced master at each iteration of the DW algorithm, a decision needs to be made as to what should be done with the columns that exit the basis of the reduced master. This question is generally one that needs to be answered based on computing resources. If memory usage is (or will become) an issue, then at some point the nonbasic columns (those with value of zero) should be purged from the reduced master. If memory usage is not a concern, then there is little to no harm in allowing the nonbasic columns to remain in memory, and therefore remain part of the reduced master formulation [O'Neill 1977]. A benefit to keeping the nonbasic columns is that there may be an opportunity to use them later in an integerization step, if necessary. There is a significant body of research on the application of DW to integer programs (see, for example Barnhart et al. [1998]; Vanderbeck [2000]; and Vanderbeck and Savelsbergh [2006]) that will not be covered here.

The interested reader is directed to the original paper by Dantzig and Wolfe [1960] or a modern textbook on linear optimization [Bertsimas and Tsitsiklis 1997] for details on this decomposition method. For discussions of computational issues associated with DW, the work of Ho [Ho and Loute 1981; Ho 1987] is insightful and informative and the dissertation by Tebboth [2001] offers a more modern and complete perspective.

3. PARALLELIZATION AND SYNCHRONIZATION

A major implementation issue involves how the subproblems should be solved. This decision is based on several factors including coding complexity, computing resources and problem size. Some problem instances of DW may have only a single subproblem, and thus do not have to be concerned with the decision of whether to solve subproblems in parallel or serially. To minimize wall clock runtime, solving all subproblems in parallel is more efficient in the presence of multicore or cluster computers. For this study, all subproblems are launched simultaneously and each solves completely at each iteration of the DW algorithm as long as the subproblem has a new objective function for that iteration. All generated columns that may improve the master's objective function are added to the reduced master.

Multithreading is achieved through the use of the pthread library. An overview of the software with a focus on interthread communication is provided in Figure 2. The synchronization is based on a modified "Sleeping Barber" problem. This problem with its solution was first formally described by Dijkstra [1965]. The problem is described in more detail in several other references [Downey 2008; Tanenbaum 2007]. The Sleeping Barber problem involves a single server thread and several client threads. The server thread sleeps while there are no clients to serve. When a client wants service, it enters a first-come, first-served queue. While the queue is not empty, the server services clients within the queue in the correct (first-come, first-served) order. In this implementation, when all subproblem threads have been serviced exactly once, the master thread performs additional computations while the subproblem threads block, awaiting the next iteration.

Sychronization in this manner is important for two reasons. First, it avoids starvation of any given subproblem thread, as the master problem's thread ensures that it will service each subproblem at each DW interation. Note that often this service is simply to acknowledge that this is no new column to be generated by that particular subproblem, thus incurring no significant computation. Second, this implementation avoids deadlock, as the subproblems claim a mutex variable when adding themselves to the waiting queue and the master uses the same mutex to remove subproblems from the waiting queue after servicing them. As an added benefit, this scheme keeps the master busy whenever a subproblem is ready to be serviced rather than having it wait for the completion of all subproblem threads.

4. PERFORMANCE

To test the correctness and performance of the implementation, a set of DW instances is needed. This author is unaware of any DW instances publicly available. To generate a robust set of test instances, one would either need to build a tool to discover the correct block-angular structure of general LPs or build a tool to randomly generate instances. The former approach would either need to be solved as an optimization problem on its own (see for example, Weil and Kettler [1971] or Borndorfer et al. [1998]) or one would need access to the original model to understand its structure [Tebboth 2001]. This work will use the latter approach of generating random test instances. This will allow for a more controlled and complete set of DW instances on which to test. In addition to the randomly generated instances, results from a large-scale, traffic flow management problem demonstrate the utility of the software on real, nonrandom problems.

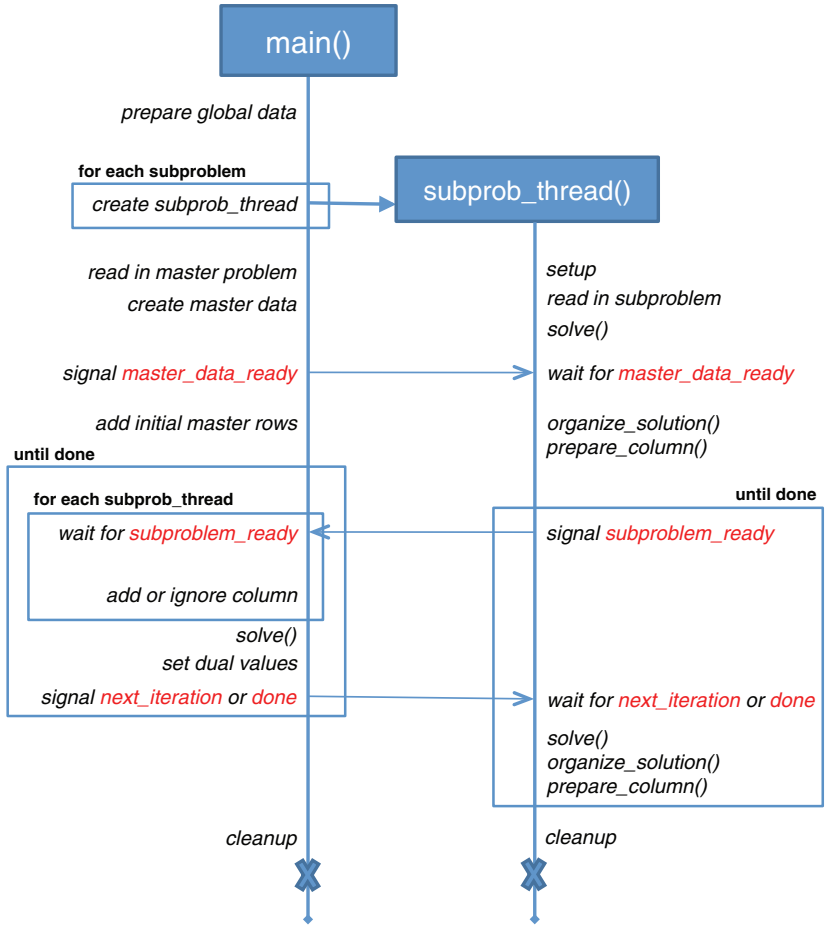


Fig. 2. The dwsolver communication architecture (ignoring algorithmic differences between Phases I and II).

4.1. Parametric Study of Randomly Generated DW Instances

A script was used to generate DW instances. This script had several parameters for tuning the generated problems. The results for varying four such parameters are detailed below. The author acknowledges that there are many additional parameters that could have been studied, but for the purposes of demonstrating the utility of the software, the provided results are sufficient.

The parameters tested were the number of subproblems, the size of the subproblems, the number of connecting constraints, and the density (or sparsity) of the subproblems. During the testing of a given parameter, all other parameters were held constant. One set of parameter values served as the baseline and was included in the results for each set of tests. That baseline had the following qualities/quantities:

—200 subproblems;
—A subproblem size of 100×30;
—200 connecting constraints;
—Subproblem density of 30%.

To measure the performance of dwsolver and verify its correctness, we compared the runtime and optimal solutions to a state-of-the-art optimization tool. Version 11.2 of CPLEX was used for this purpose. The test sytem had a quad-core, Intel Xeon processor running at 3.0 GHz with 16GB of memory. No other user processes were running during any of the tests. The results are provided in Figure 3 with speedups calculated as

$$\text{speedup} = T(1)/T(p) = T(\text{CPLEX})/T(\text{dwsolver}),$$

where T is either a measure of elapsed (wall-clock) time or total CPU time. Notice that in most cases, the two speedups are similar. This implies that much of the speedup comes from the effectiveness of the DW algorithm at handling these LPs rather than the parallelization of the algorithm. The largest discrepancy between the elapsed time and CPU time speedups occurs with large subproblems. This is due to the fact that the parallelization in the algorithm is exercised to solve the various subproblems in parallel versus being stuck in the serial bottleneck of solving and resolving the reduced master problem. With an understanding of DW, the results follow what might be expected. As one increases the number of subproblems (Figure 3(a)), dwsolver provides a greater speedup over the traditional, simplex-based approaches (as are performed by CPLEX). In addition, as the size (Figure 3(b)) and density (Figure 3(d)) of the subproblems increase, so does the performance of dwsolver. Conversely, as the number of connecting constraints increases, CPLEX begins to outperform DW (Figure 3(c)). In terms of correctness, CPLEX and dwsolver always provided the same optimal values. The values of the corresponding decision variables were not checked, as it is possible to have mulitple solutions with the same objective value due to degeneracy.

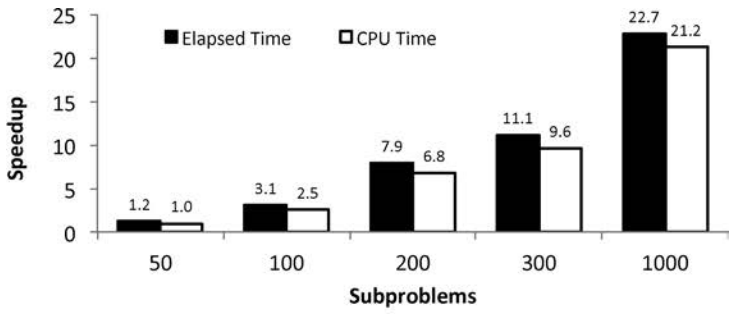
The reason for the negative result in Figure 3(c), has to do with the bottleneck in the DW algorithm. Namely, as columns are added to the reduced master problem, that problem needs to be resolved. The more connecting constraints there are, the more work needs to be done at that stage. This is a serial process in dwsolver (and in the DW algorithm) and all the subproblem threads block awaiting the result. Thus, as the number of connecting constraints increases, the relative effectiveness of the algorithm will suffer.

To summarize the results presented in Figure 3, dwsolver will perform best when there are many subproblems of some difficulty (large size and/or very dense). The speedup in these cases comes from both the efficiency of the algorithm and the more complete use of computing resources (the multiple cores in the system).

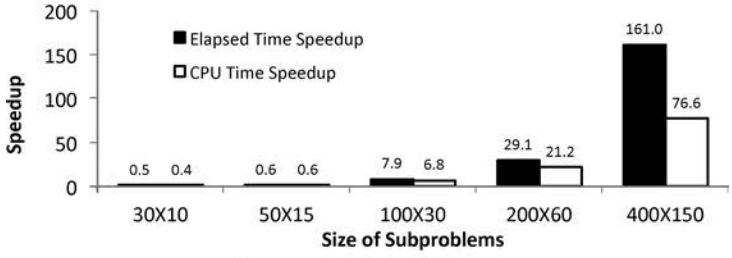
4.2. An Example from Air Traffic Management

An example of the computational performance of dwsolver on a nontrivial problem has been provided in great detail in a previous publication [Rios and Ross 2010]. That study implemented a binary integer model for air traffic flow management that produces on the order of millions of variables and constraints for practical-sized instances. Four of the instances used in that study are used here to produce new, comparative, results. The hardware and sofware used for the parametric study are the same for these results.

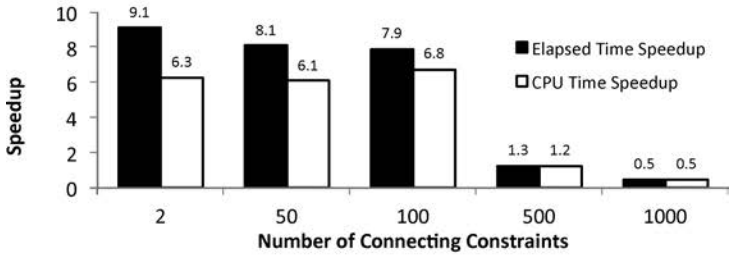
Note that in the table, the scenarios become progressively more difficult from A through D. For this particular model and data inputs, dwsolver performs better in terms of runtime in all cases. While the model used in this particular study was integer, the computational results given in Table I show just the times needed for the tools to solve the relaxation of the problem (the integer nature of the variables is ignored). The speedups follow a similar pattern to that of the parametric study. Specifically, problems of increasing size and complexity demonstrate improved speedups.



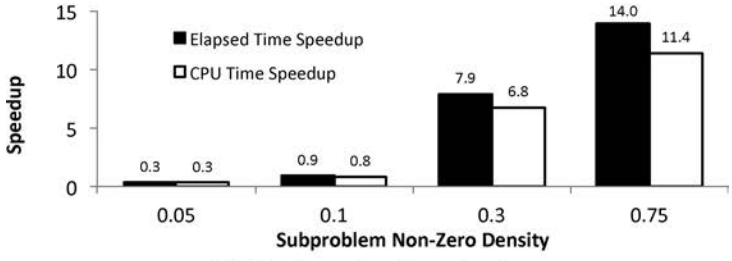
(a) Varying number of subproblems



(b) Varying subproblem size



(c) Varying number of connecting constraints



(d) Varying subproblem density

Fig. 3. Speedup of dwsolver versus CPLEX over a variety of parameters.

Table I. Results From a Large-Scale, Traffic Flow Management Model

				Elapsed Time (s)		
Scenario	Rows	Columns	Non-Zeros	CPLEX	dwsolver	Speedup
A	1,164,662	654,300	2,533,315	141.26	62.33	2.3
B	1,167,544	654,300	2,594,236	156.18	69.59	2.2
C	1,173,652	654,300	2,703,533	856.03	124.93	6.9
D	2,175,397	1,191,680	4,892,524	1,290.39	151.34	8.5

5. CONCLUDING REMARKS

The software described here is a general, parallel implementation of Dantzig–Wolfe decomposition built upon the GNU Linear Programming Kit. It takes LP-formatted input files and uses pthreads to solve subproblems in parallel to reduce the overall wall-clock runtime for solving large-scale linear programming instances with block-angular form. Though the algorithm has been known for many decades, there is no software available (commercially or otherwise) that implements it. This implementation can serve as a starting point for researchers in various domains looking to implement a model and solve it using Dantzig–Wolfe Decomposition.

There is room for improvement in this implementation. Specifically, more options for controlling the solve process would be valuable. For example, allowing termination based on some other stopping criteria would be useful in many applications. Additionally, support for unbounded subproblems will need to be added to increase the program's general applicability. Allowing for additional input formats would also increase the flexibility of the software. Despite these shortcomings the computational results indicate that use of the algorithm and software may decrease runtimes for some models by orders of magnitude.

ACKNOWLEDGMENTS

I wish to thank my thesis advisor at the University of California, Santa Cruz, Kevin Ross for his guidance and collaboration during my graduate study. I also thank Andrew Makhorin, the developer and maintainer of GLPK, for being supportive of the entire GLPK community by answering questions and providing regular updates to the software.

REFERENCES

BARNHART, C., JOHNSON, E. L., NEMHAUSER, G. L., SAVELSBERGH, M. W., AND VANCE, P. H. 1998. Branch-and-price: Column generation for solving huge integer programs. *Oper. Res. 46*, 3.

BERTSIMAS, D. AND TSITSIKLIS, J. N. 1997. *Introduction to Linear Optimization*. Athena Scientific, Belmont, MA.

BORNDORFER, R., FERREIRA, C. E., AND MARTIN, A. 1998. Decomposing matrices into blocks. *SIAM J. Optim. 9,* 1, 236–269.

DANTZIG, G. B. AND WOLFE, P. 1960. Decomposition principle for linear programs. *Oper. Res. 8*, 1, 101–111.

DESAULNIERS, G., DESROSIERS, J., DUMAS, Y., MARC, S., RIOUX, B., SOLOMON, M., AND SOUMIS, F. 1997. Crew pairing at Air France. *Euro. J. Oper. Res. 97*, 245–259.

DIJKSTRA, E. W. 1965. Cooperating sequential processes. Tech. rep., Technological University, Eindhoven, The Netherlands.

DOWNEY, A. B. 2008. *The Little Book of Semaphores* 2.1.5 Ed. Green Tea Press.

FOURER, R., GAY, D., AND KERNIGHAN, B. 2002. *AMPL: A Modeling Language for Mathematical Programming* 2nd Ed. Duxbury Press, Murray Hill, NJ.

FU, Y., SHAHIDEHPOUR, M., AND LI, Z. 2005. Long-term security-constrained unit commitment: Hybrid Dantzig–Wolfe decomposition and subgradient approach. *IEEE Trans. Power Syst. 20*, 4, 2093–2106.

GAMS DEVELOPMENT CORPORATION. 2010. General algebraic modeling system (GAMS). http: //www.gams.com/.

GÉLINAS, S. AND SOUMIS, F. 2005. Dantzig-Wolfe decomposition for job shop scheduling. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, Eds., *Column Generation*, Springer, 271–302.

Ho, J. K. 1987. Recent advances in the decompostion approach to linear programming. *Math. Program. Study 31*, 119–127.

Ho, J. K. and Loute, E. 1981. An advanced implementation of the Dantzig–Wolfe decomposition algorithm for linear programming. *Math. Program. 20*, 1, 303–326.

Ho, J. K. and Sundarraj, R. P. 1989. Decomp: An implementation of Dantzig–Wolfe decomposition for linear programming. In *Lecture Notes in Economics and Mathematical Systems*. Vol. 338, Springer-Verlag, Berlin.

Makhorin, A. 2010. GNU Linear Programming Kit, Version 4.44. http://www.gnu.org/software/glpk/glpk.html.

O'Neill, R. P. 1977. Column dropping in the Dantzig–Wolfe convex programming algorithm. *Oper. Res. 25*, 1, 148–155.

Prescott, E. S. 2004. Computing solutions to moral-hazard programs using the Dantzig–Wolfe decompostion algorithm. *J. Econ. Dynamics Control 28*, 777–800.

Rios, J. and Ross, K. 2010. Massively parallel Dantzig–Wolfe decomposition applied to traffic flow scheduling. *J. Aerospace Comput. Inform. Comm. 7*, 1, 32–45.

Tanenbaum, A. S. 2007. *Modern Operating Systems* 3rd Ed. Prentice Hall Press, Upper Saddle River, NJ.

Tebboth, J. R. 2001. A computational study of Dantzig–Wolfe decomposition. Ph.D. thesis, University of Buckingham.

Vanderbeck, F. 2000. On Dantzig–Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Oper. Res. 48*, 1, 111–128.

Vanderbeck, F. and Savelsbergh, M. W. 2006. A generic view of Dantzig–Wolfe decomposition in mixed integer programming. *Oper. Res. Lett. 34,* 296–306.

Weil, R. L. and Kettler, P. C. 1971. Rearranging matrices to block-angular form for decomposition (and other) algorithms. *Manage. Sci. 18*, 1, 98–108.

Received October 2010; revised March 2012, August 2012; accepted September 2012