ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

Linear Algebra Enhancements to the PATH Solver

Qian Li, Michael C. Ferris, and Todd Munson

Mathematics and Computer Science Division

Preprint ANL/MCS-P1565-1208

April 20, 2009

[1]This work was supported in part by Air Force Office of Scientific Research Grant FA9550-07-1-0389; by National Science Foundation Grants DMI-052 1953, DMS-0427689, and IIS-0511905; and by the Office of Advanced Scientific Computing Research, Office of Science, U.S Department of Energy, under Contract DE-AC02-06CH11357.

Linear Algebra Enhancements to the PATH Solver*

Qian Li† Michael C. Ferris‡ Todd Munson§

April 20, 2009

Abstract

This research aims enhancing the efficiency and reliability of PATH, the most widely used solver for mixed complementarity problems. A key component of the PATH algorithm is solving a series of linear complementary subproblems with a pivotal scheme. Improving the efficiency of the linear system routines (factor, solve, and update) required by the pivotal method is the critical computational issue. We incorporate two new options besides the default LUSOL package in PATH for such functionalities. One of the options employs the UMFPACK package for factor and solve operations, together with an implementation of a stable and efficient block-LU updating scheme, which leads to a significantly more effective version of PATH for solving many large-scale sparse systems. The other option exploits the COIN-OR utilities enhanced by adapting the linear refinements and scaling schemes used in the COIN-LP routines, which is effective in solving smaller-scale systems but less competitive on large-scale cases.

1 Introduction

Complementarity problems arise in diverse engineering and economics applications [18] and play an important role in constrained optimization problems, encompassing the optimality conditions for linear and convex nonlinear programs and for variational inequalities. PATH [10, 16] is a generalized Newton method for solving complementarity problems that has been effective at solving relatively difficult problems (see, for example, [2, 35]). For larger problems, however, the numerical linear algebra is inadequate to obtain good performance.

Most of the computational effort in PATH involves factoring and solving linear systems of equations and performing rank-one updates to find the Newton point via a pivotal method. LUSOL [24] is the method currently used by PATH for this functionality. We

*This work was supported in part by Air Force Office of Scientific Research Grant FA9550-07-1-0389; by National Science Foundation Grants DMI-052 1953, DMS-0427689, and IIS-0511905; and by the Office of Advanced Scientific Computing Research, Office of Science, U.S Department of Energy, under Contract DE-AC02-06CH11357.

†Mathematics Department, University of Wisconsin-Madison, 480 Lincoln Dr., Madison, WI 53706, USA. e-mail: qli@math.wisc.edu

‡Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706, USA. e-mail: ferris@cs.wisc.edu

§Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, USA. e-mail: tmunson@mcs.anl.gov

explore two other options: UMFPACK and COIN-OR. UMFPACK [5] is effective at factoring and solving sparse linear systems; we provide rank-one updates using the Sherman-Morrison-Woodbury formula. COIN-OR [27] supports COIN-LP and many other optimization projects in the COIN-OR repository; their linear algebra routines have factor, solve, and rank-one update capabilities.

This paper describes our experiences using these two options. Section 2 provides background information on complementarity problems and the PATH algorithm. Section 3 motivates exploring the UMFPACK and COIN-OR packages by first providing some statistics demonstrating that solving linear systems efficiently is the key computational issue for PATH. We then show that UMFPACK is more effective than LUSOL for factoring and solving certain linear large-scale systems. We also present some successes obtained by the COIN-LP solver to motivate our choice of the COIN-OR utilities as the other candidate for improving PATH. Section 4 describes the PATH basis object, the necessary functions for factoring and solving linear systems, and details for the existing and new UMFPACK and COIN-OR implementations. Computational results comparing the basis packages are given in Section 5. These results show that PATH/UMFPACK is more efficient than PATH/LUSOL at solving most large-scale complementarity problems, whereas PATH/COIN is effective at solving smaller-scale systems but solves large-scale problems less effectively than the other options. Section 6 summarizes our conclusions.

2 Complementarity Problems and the PATH Algorithm

The mixed complementarity problem (MCP) is defined by the set $\mathbf{B} := \{z \in \mathbb{R}^n | l \leq z \leq u\}$ with bounds $l_i \in \mathbb{R} \cup \{-\infty\}$ and $u_i \in \mathbb{R} \cup \{\infty\}$ such that $l_i \leq u_i$ for all $i = 1, \ldots, n$, and a function $F : \mathbf{B} \mapsto \mathbb{R}^n$. A vector $z \in \mathbb{R}^n$ is a solution if and only if one of the following holds for each $i = 1, \ldots, n$:

$$\begin{aligned} l_i \leq z_i \leq u_i \quad &\text{and} \quad F_i(z) = 0 \\ z_i = l_i \quad &\text{and} \quad F_i(z) > 0 \\ z_i = u_i \quad &\text{and} \quad F_i(z) < 0. \end{aligned} \tag{1}$$

Note that if $l_i = u_i$, then $z_i = l_i = u_i$ is a fixed value and any $F_i(z)$ satisfies (1).

Solving a mixed complementarity problem is equivalent to finding a zero of the normal equation [33]:

$$F_{\mathbf{B}}(x) := F(\pi_{\mathbf{B}}(x)) + x - \pi_{\mathbf{B}}(x) = 0,$$

where $\pi_{\mathbf{B}}(\cdot)$ is the Euclidean projection onto the set $\mathbf{B}$. If x^* solves the normal equation, then $z^* := \pi_{\mathbf{B}}(x^*)$ solves (1). Conversely, if z^* solves (1), then $x^* = z^* - F(z^*)$ solves the normal equation.

PATH is a generalized Newton method for solving the normal equation that is globalized via a nonmonotone search using a smooth merit function. Because the merit function is smooth, a steepest descent direction can be used when the search using the Newton point fails to yield a new iterate satisfying the nonmonotone search criteria. This algorithmic framework is well defined with global convergence guarantees and locally fast convergent rates [14, 30]. The implementation contains the following parts:

1. Preprocess the mixed complementarity problem to fix variables, improve bounds, and eliminate redundancy [17].

2. Identity an approximation to the active set at the solution using a crash technique.

3. Linearize the normal map at the current iterate, solve the linearization by constructing a piecewise linear path between the current iterate and the solution, and search using the generated path to determine a new iterate satisfying the nonmonotone search criteria.

Details for the crash method, forming and solving the linearizations, and the nonmonotone search criteria are given in the following sections. Proofs are omitted; additional details can be found in the corresponding references.

2.1 Merit Function

When solving a nonlinear system of equations, a search along the Newton direction is performed to find a new iterate that sufficiently decreases the chosen merit function value. In the implementation of the Newton method in PATH, this search uses nonmonotone descent criteria [15, 25, 26] with a watchdog technique [3]. The current default merit function is based on the Fisher-Burmeister function [19], $\phi : \mathbb{R}^2 \to \mathbb{R}$, defined as

$$\phi(a, b) := \sqrt{a^2 + b^2} - a - b.$$

Any zero of the Fisher-Burmeister function is known to satisfy the complementarity conditions:

$$\phi(a, b) = 0 \Leftrightarrow a \geq 0, b \geq 0, ab = 0.$$

With this property, a general mixed complementarity problem in the form of (1) can be reformulated as the system of equations $\Phi(z) = 0$, with $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ defined componentwise as

$$\Phi_i(z) := \begin{cases} \phi(z_i - l_i, F_i(z)) & \text{if } -\infty < l_i < u_i = +\infty \\ -\phi(u_i - z_i, -F_i(z)) & \text{if } -\infty = l_i < u_i < +\infty \\ \phi(z_i - l_i, \phi(u_i - z_i, -F_i(z))) & \text{if } -\infty < l_i < u_i < +\infty \\ -F_i(z) & \text{if } -\infty = l_i < u_i = +\infty \\ 0 & \text{otherwise} \end{cases}$$

for $i \in \{1, \ldots, n\}$. The merit function is

$$\Psi(z) := \frac{1}{2}\|\Phi(z)\|^2. \tag{2}$$

The advantage of using Ψ over other classical merit functions such as the norm of the normal map is that Ψ is continuously differentiable with gradient $\nabla\Psi(z) = \partial\Phi(z)^T\Phi(z)$. Note that the Φ is nondifferentiable in general; hence $\partial\Phi(z)$ is the generalized Jacobian of $\Phi(z)$. Detailed formulas for calculating the gradient of the merit function can be found in [14].

2.2 Crash Method

The crash method is a way to compute a good active set. It is based on a projected Newton method. First the active set with the correctly signed function values at the initial point

$z^0 = \pi_{\mathbf{B}}(x^0)$ is identified. The corresponding index set of variable z is denoted by $\mathcal{A}$, where

$$\mathcal{A} := \{i \in \{1, \ldots, n\} | \{z_i = l_i, F_i(z) \geq 0\} \text{ or } \{z_i = u_i, F_i(z) \leq 0\}\}.$$

The Newton direction for a reduced system is then computed:

$$(\nabla F_{\mathcal{I}\mathcal{I}}(z^k) + \epsilon I)\, d_{\mathcal{I}} = F_{\mathcal{I}}(z^k), \tag{3}$$

where $\mathcal{I} = \{1, \ldots, n\} \backslash \mathcal{A}$ and ϵ is a perturbation parameter (see below). We then set $d_{\mathcal{A}} = 0$ and compute an $\alpha \in (\bar{\alpha}, 1]$ such that the new iterate

$$z^k(\alpha) = (1 - \alpha) z^k + \alpha \pi_{\mathbf{B}}(z^k - d)$$

with the default line search or

$$z^k(\alpha) = \pi_{\mathbf{B}}((1 - \alpha) z^k + \alpha(z^k - d))$$

with an arc search decreases the merit function

$$\Psi(z(\alpha)) \leq \begin{cases} \Psi(z^k) - \sigma \nabla \Psi(z^k)^T (z^k - z^k(\alpha)), \\ \qquad\qquad \text{if } \nabla \Psi(z^k)^T (z^k - z^k(\alpha)) < 0 \\ (1 - \alpha\sigma)\Psi(z^k), \quad \text{otherwise} \end{cases}$$

where $\bar{\alpha}$ is a constant minimum step size and $\Psi(z)$ defined in (2) is chosen as the merit function. In solving the reduced system in (3), the perturbation parameter ϵ is zero, unless the reduced matrix $\nabla F_{\mathcal{I}\mathcal{I}}(z^k)$ is rank deficient. In this case we choose a large enough ϵ such that $\nabla F_{\mathcal{I}\mathcal{I}}(z^k) + \epsilon I$ is numerically nonsingular. Then ϵ is reduced on subsequent crash steps based on the residual of the merit function. Since the crash method does not guarantee convergence, we terminate the process if any of the following criteria are satisfied: the number of iterations exceeds a maximum value; the step length is too small; not enough changes of the active set have been made consecutively for several iterations; or the perturbation scheme is not successful. If the crash iterates manage to converge to a small enough merit function value, the original MCP is solved solely in the crash process. Some benefits of using the crash technique and its convergence properties can be found in [11].

2.3 Linearization

Newton's method for smooth functions linearizes the function at the current iterate and solves a linear system of equations to obtain a direction. The normal map $F_{\mathbf{B}}(x)$ is nonsmooth, however, because of the projection operator $\pi_{\mathbf{B}}(x)$, and a linearization of $F_{\mathbf{B}}(x)$ is not available. Rather, the following piecewise affine map approximates the function around x^k:

$$L_k(x) := (\nabla F(\pi_{\mathbf{B}}(x^k)) + \epsilon I)(\pi_{\mathbf{B}}(x) - \pi_{\mathbf{B}}(x^k)) + F(\pi_{\mathbf{B}}(x^k)) + x - \pi_{\mathbf{B}}(x), \tag{4}$$

with a perturbation parameter ϵ similarly defined and updated as in the crash procedure. This approximation is solved by constructing a parametric piecewise linear path $p^k(t)$ for $t \in [0, T^k]$, with $T^k \in (0, 1]$ satisfying

$$p^k(0) = x^k, \tag{5}$$

$$L_k(p^k(t)) = (1 - t) F_{\mathbf{B}}(x^k). \tag{6}$$

The Newton point x_N^k is defined as $p^k(T^k)$, which solves the linearization if $T^k = 1$. These conditions require that the path start at the current point x^k and that the norm of the approximation at points on the path decrease at least linearly in $1 - t$. Note that $p(t)$ may not be single valued because the path can make turns.

Instead of constructing the path directly by using (5)–(6), a pivotal technique is used to solve an equivalent linear complementarity problem constructed as follows. Let $z(t) = \pi_{\mathbf{B}}(x(t))$, $v(t) = (x(t) - z(t))_+$ and $w(t) = (z(t) - x(t))_+$. Then, $p^k(t)$, can be expressed as

$$p^k(t) = z(t) - w(t) + v(t) \quad \forall\ t \in [0, T^k].$$

Using the transformation above, together with the definition of the piecewise affine map in (4), we can express (6) as

$$Mz(t) + q - w(t) + v(t) = (1 - t)r,$$

where $M = \nabla F(\pi_{\mathbf{B}}(x^k)) + \epsilon I$, $q = F(\pi_{\mathbf{B}}(x^k)) - \nabla F(\pi_{\mathbf{B}}(x^k))\pi_{\mathbf{B}}(x^k)$, and $r = F_{\mathbf{B}}(x^k)$. In the actual implementation, we scale the covering vector r in the above equation by a scalar s. We also augment the system by incorporating a vector of artificial variables (a) to help construct an invertible basis under possible rank deficiency. In particular, the linear complementarity problem becomes

$$\begin{aligned} Mz(t) + q - w(t) + v(t) + a - \tfrac{(1-t)(sr)}{s} &= 0 \\ w^\top(z - l) &= 0 \\ v^\top(u - z) &= 0 \\ z \in \mathbf{B}, w \geq 0, v \geq 0, a \equiv 0, t \in [0, 1]. & \end{aligned} \tag{7}$$

A guess of the initial basis of the above system follows the active set approximation resulting from the crash process. We rely on the factorization routines to detect possible rank deficiency and identify linearly dependent rows and columns in the initial basis. The basis is then defined appropriately based on the singularity information, so that the system (7) has an invertible basis at the starting point. A pivotal technique similar to Lemke's method with specific entering and leaving pivotal rules can then be used to construct the path. Each pivot leads to a new piece on the path. If, in the end, the pivots terminate with t leaving the basis at 1, then the linear complementarity problem is solved successfully, and the Newton point $x_N^k = p^k(1)$ is generated from $(z(1), w(1), v(1))$. When ray termination or cycling occurs, however, the path generation will terminate at a point $p^k(T^k)$ with $T^k < 1$. More details on constructing an invertible basis and pivotal rules can be found in [10] and [16].

2.4 Nonmonotone Search

The nonmonotone descent scheme implemented in the PATH algorithm distinguishes between *m-steps*, *d-steps*, *watchdog steps* [3], and *projected gradient steps*.

The merit function value at the Newton point $z^k(T_k)$ is checked by using nonmonotone descent criteria during m-steps. In particular, given a *reference value* $\mathcal{R}$, the point is acceptable if

$$\Psi(z^k(T_k)) \leq \begin{cases} \mathcal{R} - \sigma\nabla\Psi(z^k(0))^T(z^k(0) - z^k(T_k)), \\ \qquad\qquad \text{if } \nabla\Psi(z^k(0))^T(z^k(t) - z^k(T_k)) < 0 \\ (1 - \sigma)\mathcal{R}, \quad \text{otherwise} \end{cases} \tag{8}$$

The reference value is decreased as the algorithm proceeds. If the Newton point satisfies this criterion, we save it as a *check point* for use with the watchdog strategy. Every time the check point is updated, the corresponding Newton point $x^k(T_k)$ comprising $(z^k(T_k), w^k(T_k), v^k(T_k))$ and the Newton point found in the next iteration $z^{k+1}(T_{k+1})$ and $x^{k+1}(T_{k+1})$ are saved so that regeneration of the path will not be necessary if we have to go back to this check point.

A d-step is acceptable if the Newton point $z^k(T_k)$ is close enough to the current point $z^k(0)$. In particular, the point is accepted if

$$||z^k(T_k) - z^k(0)|| \leq \Delta$$

and the merit function does not become too large, where Δ is initialized to a preset value and decreasing as the algorithm progresses. If, at the same time, the nonmonotone descent criterion is satisfied, then current point is saved as a check point. If the d-step conditions are not satisfied, then the nonmonotone criterion is checked. Moreover, after every $\bar{n}$ d-steps, the nonmonotone descent criterion is checked.

If the nonmonotone descent criterion is violated in an m-step or if the merit function value would increase too much over the current reference value when accepting a d-step, then a *watchdog* step is taken. The watchdog step retrieves the most recent check point saved consisting of the four points $z^{\tilde{k}}(0)$, $x^{\tilde{k}}(0)$, $z^{\tilde{k}}(T_{\tilde{k}})$, and $x^{\tilde{k}}(T_{\tilde{k}})$ for some $\tilde{k}$. With these points, a search is performed to find a new point satisfying the nonmonotone descent criterion. The user can select to search along the line segment connecting the two projected points

$$z^{\tilde{k}}(\alpha) = (1-\alpha)z^{\tilde{k}}(0) + \alpha z^{\tilde{k}}(T_{\tilde{k}}) \tag{9}$$

or the projected arc connecting the two Newton points

$$z^{\tilde{k}}(\alpha) = \pi_{\mathbf{B}}((1-\alpha)x^{\tilde{k}}(0) + \alpha x^{\tilde{k}}(T_{\tilde{k}})). \tag{10}$$

In either case, we find a step length $\alpha \in (\bar{\alpha}, 1)$ iteratively such that

$$\Psi(z^{\tilde{k}}(\alpha)) \leq \begin{cases} \mathcal{R} - \sigma\nabla\Psi(z^{\tilde{k}}(0))^T(z^{\tilde{k}}(0) - z^{\tilde{k}}(\alpha)), \\ \qquad\qquad \text{if } \nabla\Psi(z^{\tilde{k}}(0))^T(z^{\tilde{k}}(0) - z^{\tilde{k}}(\alpha)) < 0 \\ (1-\alpha\sigma)\mathcal{R}, \quad \text{otherwise,} \end{cases} \tag{11}$$

where $\bar{\alpha}$ is a constant minimum step size and $\mathcal{R}$ is the current reference value. The default search type in the PATH solver is a line search.

If the step size in the search becomes too small without finding a point satisfying the nonmonotone descent criterion, then a monotone projected gradient step for our smooth merit function Ψ is taken as follows. We first move back to the *best point*, which is the check point with the smallest merit function value, say $\bar{k}$. Then a step length $\alpha \in (\bar{\alpha}, 1)$ is determined such that

$$\Psi(z^{\bar{k}}(\alpha)) \leq \begin{cases} \Psi(z^{\bar{k}}(0)) - \sigma\nabla\Psi(z^{\bar{k}}(0))^T(z^{\bar{k}}(0) - z^{\bar{k}}(\alpha)) \\ \qquad\qquad \text{if } \nabla\Psi(z^{\bar{k}}(0))^T(z^{\bar{k}}(0) - z^{\bar{k}}(\alpha)) < 0 \\ (1-\alpha\sigma)\Psi(z^{\bar{k}}(0)), \quad \text{otherwise,} \end{cases} \tag{12}$$

with

$$z^{\bar{k}}(\alpha) = (1-\alpha)z^{\bar{k}}(0) + \alpha\pi_{\mathbf{B}}(z^{\bar{k}}(0) - \nabla\Psi(z^{\bar{k}}))$$

when using a line search and

$$z^{\bar{k}}(\alpha) = \pi_{\mathbf{B}}((1-\alpha)z^{\bar{k}} + \alpha(z^{\bar{k}} - \nabla\Psi(z^{\bar{k}})))$$

when using an arc search. The default gradient search type in PATH is an arc search.

2.5 Summary

A summary of the main PATH algorithm is as follows:
PATH CODE

1. Initialization: let $z^0, \bar{n} \geq 1, \triangle = \bar{\triangle} > 0, \beta \in (0,1)$ be given:
   set $k = 0$, *check_point* $= 0$, *best_point* $= 0$; $j = 0, b = 0, \triangle_0 = \triangle, \mathcal{R}_0 = \Psi(z^0)$.

2. If $\Psi(z^k) = 0$, stop.

3. Update the perturbation parameter ϵ. Using the linearization approximation L_k, apply the transformation and generate a path from z^k to the solution of the linear complementarity subproblem: $[0, T_k] \mapsto \mathbf{B}, T_k \in (0,1]$ satisfying (7).

4. If $(k <$ *check_point* $+ \bar{n})$ then
   d-step:
   if $(||z^k(T_k) - z^k(0)|| < \triangle)$, the step is small enough; accept it:
   set $z^{k+1}(0) = z^k(T_k)$;
   set $\triangle = \triangle * \beta$;
   if the nonmonotone descent criterion in (8) is satisfied,
   update *check_point*;
   increment k and go to Step 2.
   if $\Psi(z^k) >$ *LargeConstant* $* \mathcal{R}_j$,
   perform a watchdog step;
   else, the step is too big; perform an m-step.
   else
   m-step:
   if the monotone descent criterion in (8) is satisfied with the reference value $\mathcal{R}_j$,
   accept the step:
   set $z^{k+1}(0) := z^k(T_k)$;
   else perform a watchdog step:
   set $k =$ *check_point*, $\triangle = \triangle_j$;
   if $\alpha \in (\bar{\alpha}, 1)$ can be found satisfying the condition in (11) with

reference value $\mathcal{R}_j$, by conducting a line or arc search in (9)-(10),

set $z^{k+1}(0) := z^k(\alpha)$;

else perform a projected gradient step:

set $k = best\_point, \triangle = \triangle_b$;

find $\alpha \in (\bar{\alpha}, 1)$ satisfying (12);

update *check_point*:

increment j; update $\mathcal{R}_j$; set $\triangle_j = \triangle$;

set $check\_point = k + 1$;

update *best_point* if $\Psi_{check\_point} < \Psi_{best\_point}$, by cloning the *check_point* info to the *best_point*;

5. Increment k, and go to Step 2.

At the beginning of Step 3, with $z^k \in \mathbf{B}$ given, the initial values for w^k and v^k need to be supplied. In other words, we need to calculate a corresponding x^k, whose projection is z^k and which has the best normal map residual. We do so by solving the following optimization problem (omitting the superscripts):

$$\begin{array}{rc} & \min_{w,v} ||x - z + F(z)|| \\ \text{s.t.} & x = z - w + v \\ & \pi_{\mathbf{B}}(x) = z \\ & w \geq 0, v \geq 0, w^T v = 0. \end{array}$$

In practice, this problem is solved simply as

$$\begin{cases} \text{if } z_i \leq l_i \text{ and } f_i > 0 & x_i = l_i - f_i, w_i = f_i, v_i = 0 \\ \text{else if } z_i \geq u_i \text{ and } f_i < 0 & x_i = u_i - f_i, w_i = 0, v_i = -f_i \\ \text{else} & x_i = z_i, w_i = v_i = 0 \end{cases}$$

for $i \in \{1, \ldots, n\}$.

Some "safeguard" steps are omitted from the algorithm summary. For example, to determine if whether we should go into a watchdog step directly, we always check first to see that the normal function $F_{\mathbf{B}}$ is defined at the newly generated point z^k.

3 UMFPACK and COIN-OR Utilities

The key to obtaining efficiency in the PATH algorithm depends on solving a series of subproblems in the form of linear complementarity problems (7) using a relative of Lemke's method. We call each iteration of PATH a *major* iteration and each pivotal step in solving the linear MCP a *minor* iteration. At each major iteration, factorization of the current basis matrix, which corresponds to the active set at the current point, together with rank-one updates (corresponding to the pivotal steps) is required. In the crash procedure described above, a Newton system as in (3) needs to be solved at each iteration, which also requires

routines capable of factoring and solving linear systems. In the current implementation of PATH [16], the factor, solve, and update procedures were first coded to use the LUSOL routines.

A test run on the MCPLIB problems with PATH/LUSOL suggests that on average, 74% of the total solving time is spent factoring, solving, and updating linear systems. Moreover, as we can see below using a dynamic game (*dyngame*), the proportion of time spent in these routines increases significantly as the size of the system grows. The dynamic game (*dyngame*) also provides better means for comparing the performance of the LUSOL routines with one of its alternatives – the UMFPACK routines.

Dynamic games are mathematical models of the interaction between independent agents controlling a dynamical system. Such situations occur in military conflicts, economic competition, and parlor games such as chess or bridge. The actions of the agents (also called players) influence the evolution over time of the state of the system. The difficulty in deciding what should be the behavior of these agents stems from the fact that each action an agent takes at a given time will influence the reaction of the opponent(s) at later times. The specific model considered here is a game played on a grid based on the model of dynamic competition in an oligopolistic industry [13, 29]. This model has been used extensively in applications such as advertising, collusion, mergers, technology adoption, international trade, and finance and has become a central tool in analysis of strategic interactions among forward-looking players in dynamic environments.

Figure 1 is the nonzero structure of the initial basis matrix of the *dyngame* problem. The size of the matrix is 1600×1600, with $16,656$ nonzero elements. Hence the basis matrix is rather sparse, with density equal to 0.65%.

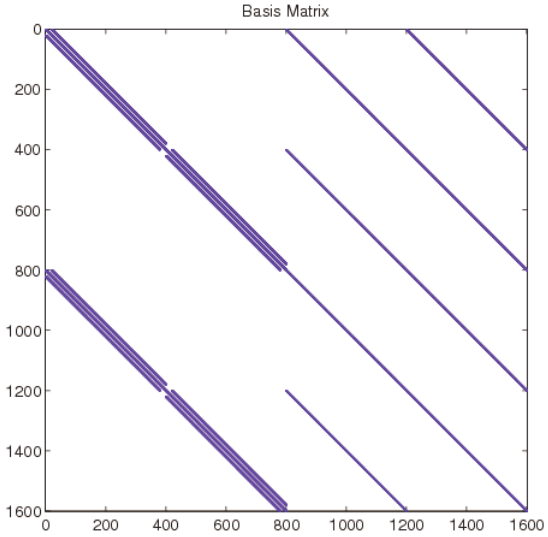

Figure 1: Initial basis matrix of *dyngame* for $\gamma = 1$ and N=20

Figures 2 and 3 are the fill-in of the summations of the lower and upper triangular matrices obtained from the the LU factorization computed by the LUSOL routine and UMFPACK routine, respectively. The LUSOL routine computes lower and upper triangular

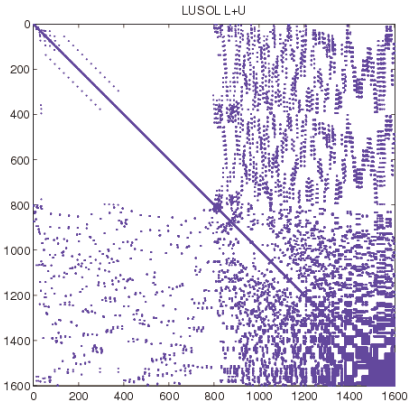

Figure 2: $L + U$ obtained from the LU factorization by the LUSOL routine

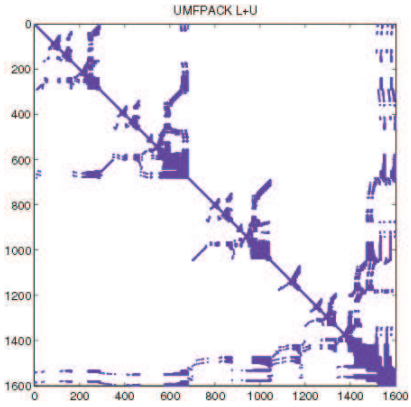

Figure 3: $L + U$ obtained from the LU factorization by the UMFPACK routine

matrices L and U which satisfy $LU = PBQ$, where B is the original matrix and P, Q are the permutation matrices. The L and U matrices obtained from the UMFPACK routine satisfy $LU = P\tilde{B}Q$, where P, Q are the permutation matrices and $\tilde{B}$ is the original matrix after row scaling. The number of nonzero elements is $68,203$ in Figure 2 and $66,684$ in Figure 3. Hence the density of the matrices is 2.66% in Figure 2 and 2.6% in Figure 3. The densities of the LU matrices resulting from the two factorization routines are also similar when the size of the problem increases, as seen in Table 1, where *Dim* is the dimension of the basis matrix and *Nnz* is the number of nonzero elements in $L + U$.

Table 1: *Density of the $L + U$ Matrices from Factorization*

Problem		LUSOL		UMFPACK	
N	Dim	Nnz	Density	Nnz	Density
20	1600×1600	68,171	2.66%	66,684	2.60%
50	$10,000 \times 10,000$	587,112	0.59%	658,755	0.66%
100	$40,000 \times 40,000$	2,773,928	0.17%	2,778,235	0.17%

Table 1 shows that the densities of the resulting $L+U$ matrices do not increase as the size of the problem grows. However, the time taken to perform the basis package functionalities when using LUSOL grows significantly with the increase of the problem size, as seen in Table 2 (*Factor* is the time spent in the basis package routines; *Total* is the total CPU time taken in the whole PATH code; *Pct* is the percentage of the total time spent in performing factor and solve routines), whereas a rather moderate growth in the time spent in the basis routines is observed when using the UMFPACK package. Table 2 again indicates that the major computational issue in PATH lies in performing the basis routines efficiently, since most of the time is spent in the basis routines. One might postulate that the time increase is due to more irregular data access, but we have not demonstrated this rigorously at this stage. Clearly, however, if we can carry out the basis routines efficiently, we are potentially able to substantially improve the efficiency of the PATH algorithm.

Table 2: *Time Spent in Basis Routines Vs Total Time Taken in PATH*

Problem	LUSOL			UMFPACK		
N	Factor	Total	Pct	Factor	Total	Pct
20	0.13	0.18	72.2%	0.09	0.14	64.3%
50	2.66	2.99	89.0%	0.86	1.18	72.9%
100	16.55	17.92	92.4%	4.41	5.78	76.3%

COIN-OR is a collection of open source routines for solving linear, mixed integer, nonlinear, and mixed integer nonlinear programming problems. These solvers have basis factorization utilities designed to support many of the projects in the COIN-OR repository, including the COIN-LP solver. The results of solving several moderately sized LPs using different LP solvers are presented in Table 3. The size of the initial models is given by the number of rows, columns, and number of nonzeros (*Row, Col* and *Nnz*). The CPU-seconds (*Time*) spent in solving these LPs using various LP solvers are presented under each solver name for comparison. The examples in Table 3 shows that the COIN-LP solver is comparable to the CPLEX solver, which is widely considered to be the best linear program (LP) solver, and is much more efficient than LUSOL as used in the MINOS/SNOPT solver. Since the COIN-LP solver depends on the COIN-OR utilities to process its linear systems, these results lead to the assumption that the COIN-OR utilities may provide a more efficient linear systems solver than LUSOL. This assumption may fail, however, because the COIN-LP solver may rely on its pivot choice instead of linear systems solver to achieve the efficiency exhibited here.

Table 3: *LP Examples*

Problem				Time			
Name	Row	Col	Nnz	CPLEX	COIN-LP	MINOS	SNOPT
storm	14388	34115	114974	0.61	1.25	15.19	16.45
pds-06	9882	28656	82270	0.24	0.34	18.68	19.29
pds-10	16559	48765	140064	0.56	0.71	155.94	126.14

4 Basis Packages

The PATH solver requires a basis package to provide certain functionality expressed via the following functions:

Basis_Factor() Factors the given basis matrix.

Basis_Solve() Uses the factors to solve a linear system of equations.

Basis_Replace() Replaces a column of the basis matrix.

Basis_NumSingular() Indicates the singular row(s) and column(s) of the basis matrix when singularity is detected.

4.1 LUSOL

Currently, the PATH solver uses the LUSOL [24] sparse factorization routines from the MINOS [28] nonlinear programming solver. The routines are based on a Markowitz factorization and a Bartels-Golub update [1, 31, 32]. The major functions provided by the LUSOL routines are as follows:

Factor For a given sparse matrix $A_{m\times n}$, computes a factorization $A = LU$ by Gaussian elimination with a Markowitz pivotal strategy to choose permutations P and Q, so that PLP' is lower triangular and $P'UQ$ is upper triangular (when $m = n$) or upper trapezoidal (when A is rectangular).

Solve For a given vector b_m, uses the LU factors to find a vector x_n that solves the linear system $Ax = b$.

Update Modifies L and U to obtain a new factorization $A = LU$ when A is updated. The updates include addition, deletion, or replacement of a column or row of matrix A and rank-one modification.

Hence the LUSOL routines are able to provide the functionalities required both by the crash procedure (as in (3)) and Lemke's procedure (as in (7)) in the PATH algorithm.

If the matrix A is singular or ill-conditioned, the return status from the factorization routine will indicate the detection of singularity. Since the dimensions and condition of A are almost always reflected in U, the number of "apparent" singularities is taken to be the number of the "small" diagonals of the permuted U. This number, together with the positions of such elements, is also returned by the factorization routine. The $Basis\_NumSingular()$ routine is designed to use this information to determine the corresponding singular row(s) and column(s) of A.

4.2 UMFPACK

UMFPACK is a set of routines for solving unsymmetric sparse linear systems [6, 5, 8, 7]. It is based on the unsymmetric multifrontal method and direct sparse LU factorization. The primary UMFPACK routines required to factorize A and/or solve $Ax = b$ are as follows:

Factor For a given sparse matrix A, performs a column preordering to reduce fill-in and a symbolic factorization. Then performs a numerical factorization, $PAQ = LU$, $PRAQ = LU$ or $PR^{-1}Q = LU$, where R is a diagonal matrix of scale factors, P and Q are permutation matrices, L is lower triangular, and U is upper triangular or upper trapezoidal when A is rectangular, using the earlier symbolic ordering and analysis.

Solve Solves a square sparse linear system $Ax = b$, using the numeric factorization computed by factorization routines.

The UMFPACK package alone is sufficient to support the crash procedure, since only factor and solve functionalities are required. However, in order to solve the linear MCPs in (7) efficiently using the pivotal technique, an algorithm that enables rank-one updates must be provided. In our implementation of the new basis package, we exploit a stable and efficient

block-LU updating method, proposed in [22] and [12]. A brief description of the rank-one updating method is follows.

A sequence of rank-one modification occurs when we solve the linear MCP in (7). For the sake of simplicity, let us express the system in (7) in a more general form:

$$\begin{aligned} Hy &= h \\ y &\in \tilde{\mathbf{B}} \end{aligned} \tag{13}$$

with $\tilde{\mathbf{B}} = \mathbf{B} \times [0,1] \times [0,\infty] \times [0,\infty]$. Suppose that $H_{.B}$ denotes the basis matrix corresponding to a basic feasible solution of the above system (13), whose LU factorization is computed. At each subsequent pivot a rank-one modification is made to the basis matrix. After a certain number of updates, let V_k contain the columns from H that have newly become basic since the factorization of $H_{.B}$; let U_k contain unit vectors representing the location of the column being updated, with k referring to the number of columns in U_k. (Note that k is not necessarily equal to the number of updates performed since the factorization of $H_{.B}$.) Hence the above matrices have the following dimensions: $H_{.B}$ is $n \times n$, V_k is $n \times k$, and U_k is $n \times k$. The new basis matrix $H_{.\tilde{B}}$ can be expressed as

$$H_{.\tilde{B}} = H_{.B} + (V_k - H_{.B}U_k)U_k^T,$$

and it is easy to see that the system $H_{.\tilde{B}}\, y = \tilde{h}$ is equivalent to

$$\begin{bmatrix} H_{.B} & V_k \\ U_k^T & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \tilde{h} \\ 0 \end{bmatrix} \tag{14}$$

with the solution $y = y_1 + U_k\, y_2$.

The matrix in (14) has the following block-triangular factorization:

$$\begin{bmatrix} H_{.B} & V_k \\ U_k^T & \end{bmatrix} = \begin{bmatrix} H_{.B} & \\ U_k^T & -C_k \end{bmatrix} \begin{bmatrix} I & Y_k \\ & I \end{bmatrix},$$

where

$$H_{.B}Y_k = V_k, \quad C_k = U_k^T Y_k.$$

We see that the solution to (13), and hence $H_{.\tilde{B}}\, y = \tilde{h}$ may be obtained by

$$\begin{aligned} H_{.B}w &= \tilde{h}, \\ C_k y_2 &= U_k^T w, \\ y_1 &= w - Y_k y_2. \end{aligned}$$

All updating information is carried along via the Schur-complement matrix $C_k (= U_k^T H_{.B}^{-1} V_k)$ and the matrix of transformed columns Y_k. Rather than modifying the factors of $H_{.B}$, we can now carry out the updates on the factors of a much smaller matrix C_k, which has dimension k independent of the size of the original matrix $H_{.B}$. The LU factors of $H_{.B}$ can be used without modification for many iterations.

In our implementation, the maximum size of C_k is set to be 100. As long as the dimension of C_k doesn't exceed this number, no refactorization is performed. (The maximum size is

set to be an option so that the user can modify this option to control the refactorization frequency.)

The updates of C_k are carried out on the LU factors of C_k, in a slightly different form: $L_k C_k = U_k$, where U_k is an upper triangular matrix and L_k is a square matrix. L_k and U_k are stored as dense matrices, since the size of C_k is relatively small. Depending on which columns enter and leave the basis at each pivot, different types of updates to matrices C_k and Y_k are performed. Correspondingly, the LU factors of C_k are modified by using sweeps of stabilized elementary transformations [23, 4]. In solving $C_k y_2 = U_k^T w$, only a matrix-vector multiplication with L_k and $U_k^T w$ and a backward triangular solve with U_k are required. Therefore the updates can be achieved at a much lower cost than performing a full factorization at every pivot. A detailed description of the update types can be found in [12], together with the description of how to store Y_k efficiently with dense and sparse parts. After a number of iterations, the scheme performs a factorization of the current basis $H_{.\tilde{B}}$ and redefines it to be $H_{.B}$. This part of our C code is based on a Fortran code LUMOD, originally developed by [34]. The updating routines implemented above can be combined with other factor and solve packages (besides UMFPACK) and be used more generally in other linear system operations where such functionalities are required.

When the matrix being factorized is singular or ill-conditioned, UMFPACK routines do not provide the same singularity information (such as the number of apparent singular elements and their locations in U) as the LUSOL routines. Therefore the upper triangular matrix U needs to be extracted from the object returned by UMFPACK together with the permutation matrix P and Q. Corresponding routines are supplied to determine the singular row(s) and column(s) of the original matrix A. The threshold number used by the LUSOL routines to determine "small" elements in U is adopted for the factors obtained by using the UMFPACK routines.

4.3 COIN

The COIN-OR utilities are a collection of open source utilities written in C++. Factor, solve, and update are contained in a set of "CoinFactorization" routines, based on a Markowitz factorization and a Forrest-Tomlin update [21].

Factor For a sparse matrix A given as triplets, computes a factorization by exploiting the "CoinFactorization::factorize" routine in COIN-OR utilities. The factor process starts from a sparse factor routine. Conditioning on the number of elements in the selected pivotal row and column, the sparse factor routine continues in one of four sparse routines. At every pivot iteration, a check is performed in order to determine whether to switch to a dense factor routine provided by either a Fortran code from LAPACK (preferred) or the COIN-OR utilities itself.

Solve For a given vector b, uses the existing LU factors of A to solve $Ax = b$ with the "CoinFactorization::updateColumnFT" routine of the COIN-OR utilities. Permutes the resulting solution vector by using the permutation matrix returned from the above factor routine.

Update Exploits the "CoinFactorization::replaceColumn" routine in the COIN-OR utilities to obtain a new factorization when matrix A is updated.

The COIN-OR utilities alone are able to provide the functionalities required by both the crash procedure (as in (3)) and Lemke's procedure (as in (7)) in PATH. To improve the accuracy and stability of COIN, we adapted the linear refinements and scaling used by COIN-LP routines. These extra procedures cost one more solve at each crash step and one more refactorization at each major iteration.

If the matrix A is singular or ill-conditioned, the factorization routine will indicate the detection of singularity on exit. The columns that are left unpivoted will be marked by -1 in the permutation matrix (in vector form) returned. The $Basis\_NumSingular()$ and $Basis\_GetSingular()$ routines are modified to exploit this information and determine the corresponding singular row(s) and column(s) of the matrix A.

4.4 DENSE

Besides the three basis packages designed for sparse systems, PATH can use a set of dense basis factorization utilities. The dense routines exploit the updating procedure [20] and provide the following major functionalities.

Factor For a given dense square matrix $A_{n\times n}$, computes a factorization $PAQ = LU$ by Gaussian elimination with a complete pivoting strategy to choose permutations P and Q, so that L is unit lower triangular and U is upper triangular.

Solve For a given vector b_n, solves the two triangular systems with the L, U matrices using forward and backward substitution to find a vector x_n that solves the linear system $Ax = b$.

Update Updates L and U in a stable manner to obtain a new factorization when a rank-one update is performed on A.

Despite the dense routines' simplicity, we do not recommend it in our applications because we generally deal with large sparse systems, especially within the Lemke procedure. In the dense case, only the singularity message is printed, but no actual $Basis\_NumSingular()$ routine is supplied.

5 Computational Results

The results obtained from using the LUSOL, UMFPACK, and COIN basis packages on the *dyngame* problem are presented in Table 4. The problem size containing the number of columns and rows (Col/rows) and the number of nonzeros ($n$nz) is listed in the first column of Table 4. We then report the data parameter γ and the time in CPU-seconds spent in the basis package (Basis time), the total time (Total), and the final residual (Residual represented by $\alpha(-\beta) = \alpha \times 10^{-\beta}$) when using the LUSOL, UMFPACK and COIN basis packages.

The CPU time spent in the basis package and the whole program in Table 4 shows clearly that the UMFPACK basis package is significantly more efficient than LUSOL. The final residuals suggest that the UMFPACK basis package leads to increased accuracy. Arguably, the UMFPACK basis package also improves the reliability of the MCP solutions, not only because of the reduced residuals, but also because for $\gamma = 3$ with $N = 200$ and $N = 300$

Table 4: *Results of the* dyngame *Problem*

Problem Size	γ	LUSOL		UMFPACK		COIN	
		Basis/Total Time	Residual	Basis/Total Time	Residual	Basis/Total Time	Residual
N = 20	1	0.13/0.18	2.82(−9)	0.09/0.14	1.96(−9)	0.17/0.22	1.96(−9)
Col/rows = 2,400	2	0.14/0.19	3.56(−9)	0.09/0.14	1.60(−9)	0.17/0.22	1.61(−9)
Nnz = 19,856	3	0.07/0.12	3.30(−9)	0.08/0.13	9.69(−10)	0.08/0.13	9.77(−10)
N = 50	1	2.66/2.99	1.99(−8)	0.86/1.18	1.99(−9)	4.80/5.12	2.00(−9)
Col/rows = 15,000	2	0.93/1.23	1.27(−8)	0.55/0.86	1.65(−9)	1.32/1.62	2.51(−9)
Nnz = 127,616	3	0.86/1.17	4.84(−9)	0.49/0.79	9.99(−10)	1.34/1.64	1.01(−9)
N = 100	1	16.55/17.92	6.88(−7)	4.41/5.78	2.09(−9)	∗	∗
Col/rows = 60,000	2	6.07/7.39	5.36(−7)	2.03/3.33	1.70(−9)	∗	∗
Nnz = 5,152,216	3	9.28/10.57	5.97(−9)	1.87/3.16	1.10(−9)	18.61/19.90	1.10(−9)
N = 200	1	93.52/99.14	3.73(−7)	23.48/29.10	2.43(−9)	-	-
Col/rows = 240,000	2	29.07/34.34	7.82(−8)	8.72/14.02	1.89(−9)	-	-
Nnz = 2,070,416	3	92.88/98.16	1.50(−7)	8.45/13.72	1.37(−9)	135.82/141.12	1.38(−9)
N = 300	1	258.78/271.56	2.81(−7)	63.01/75.73	2.90(−9)	-	-
Col/rows = 540,000	2	69.41/81.35	2.95(−7)	22.13/34.13	2.17(−9)	-	-
Nnz = 4,665,616	3	329.76/342.28	5.45(−7)	21.72/34.18	5.32(−7)	352.71/365.20	5.32(−7)

the time spent on LUSOL is significantly longer than for $\gamma = 2$, while experiences from the rest of the smaller problems indicate that $\gamma = 3$ should take a similar amount of time to solve as $\gamma = 2$, which is precisely the situation with UMFPACK. The COIN basis package, on the other hand, is successful only in solving a subset of the *d*yngame problems. For the instances marked with $*$, COIN encounters accuracy issues when solving the linear systems. For example, in solving the problem with $\gamma = 1$ and $N = 100$, the inf-norm of the residual from computing the first crash iteration is checked. The COIN basis package has a residual as big as 5.91×10^2, whereas LUSOL solves the system with inf-norm of the residual equal 1.88×10^{-09}, and UMFPACK solves the system with residual equal 2.27×10^{-13}. We report this error but do not want to change COIN-OR source, so we can use the updated versions of COIN-OR utilities as they are available. The problems marked with $-$ are too big for COIN to factor. On the successfully solved instances, COIN takes much more time than both LUSOL and UMFPACK, especially as the problem size grows. The number of major, minor, and crash iterations taken to solve each problem (successfully) is identical for all three basis packages.

Except for the option setting for choosing between different basis packages, the default settings have been used for all the problems in Table 4 but one. For the last problem with $N = 300$ and case 3, crash_searchtype is set to be *arc*, since the default line search with both basis packages takes enormous amount of time to solve.

As expected, it is important to exploit sparse factoring and solving routines on the *dyngame* problem rather than the dense basis routines. This is clearly seen from the results in Table 5, obtained by running *dyngame* with factor_method set to be *dense*. The time taken by the dense option is hundreds of times slower than the sparse options (UMFPACK and LUSOL). It takes more than 2400 CPU-seconds to solve the first crash step of *dyngame* with $N = 50$; hence the time for solving larger size problems is not shown.

Table 5: *CPU Time Spent in the Basis Package for Solving* dyngame *Using Dense Option*

Problem Size	γ	Basis Time	Residual
N = 20	1	61.37	1.96(−9)
Col/rows = 2,400	2	60.51	1.61(−9)
Nnz = 19,856	3	51.63	9.70(−10)

The new basis packages are also tested on the MCPLIB [9] problems (1005 MCPs). The models on which all basis packages succeed or fail are not reported here. Table 6 lists only the models whose solution status is different for different basis packages. The total number of solves with different starting points for each problem is listed in the second column. Under each basis package name, the number of failures is shown. Comparatively, LUSOL has the most failures in solving these relatively difficult problems. UMFPACK does slightly better than LUSOL. COIN has the fewest failures among the three options. The summation of CPU times (basis and total), together with the total of number of major and minor iterations for solving this MCP test set is presented in Table 7. In order to make the computational times comparable, only the problems that are solved successfully by all the options are considered (972 MCPs). We set the LUSOL statistics to be the base value (i.e., letting them be 100%) and compute the relative ratios of the other options' statistics to the base value. As we can see in Table 7, LUSOL outperforms both UMFPACK and COIN slightly in time. This result suggests that UMFPACK does not have much advantage in reducing the solving time for these relatively smaller-scale problems. As we improve the numerical stability of COIN by incorporating refinements and scaling process, COIN does achieve more successes but take more time to solve these MCPs.

Table 6: *Comparative Results on MCPLIB Models*

Model	No. of Solves	LUSOL	UMFPACK	COIN
cgereg	22	1	1	0
duopoly	1	0	1	1
fixedpt	2	0	2	0
fried8	5	2	0	0
jiangqi	3	1	0	0
kyh	2	0	0	1
kyh-scale	2	1	1	0
pgvon105	6	1	0	0
pgvon106	6	1	1	1
tiebout1	2	0	1	0
tinsmall	64	0	0	1
venables	2	2	0	0
fails count		9	7	4

Table 7: *Sum of Iterations and Time spent on MCPLIB problems*

Basis Package	Major (ratio%)	Minor (ratio%)	Basis Time (ratio%)	Total Time (ratio%)
LUSOL	10110 (100)	188720 (100)	93.64 (100)	119.59 (100)
UMFPACK	10448 (103)	224354 (119)	102.46 (109)	135.97 (114)
COIN	11434 (113)	180530 (96)	107.69 (115)	133.03 (111)

Table 8 contains the set of the problems in MCPLIB whose sizes can be increased. The CPU-seconds spent on the basis package using UMFPACK, LUSOL, and COIN in solving each instance are listed in Table 8. For most of these problems (*bai_haung, bratu, dirkse2, dongbai, obstacle*, and *o*pt_cont), as the problem size increases, the advantage of UMFPACK

in reducing the solving time over LUSOL becomes more significant. For the other problem (*bert_oc*), a time reduction using UMFPACK basis package is not observed. Nevertheless, the UMFPACK basis package can be used as an alternative to LUSOL without much loss of efficiency on this problem. The COIN option in general takes more time than LUSOL except for the *dirkse2* and *dongbai* problems. The COIN basis time marked by $*$ suggests that COIN may have accuracy issues because it takes different PATH steps from LUSOL and UMFPACK. COIN cannot process the problems marked by $-$ because the problems sizes are too big. For the largest instance of the *dongbai* problem, UMFPACK essentially takes 19 Newton steps to solve (time marked by $\diamond$), whereas LUSOL and COIN both take 20, which suggests that UMFPACK solves this system with better accuracy than LUSOL and COIN.

Table 8: *Comparative Results on Enlarged MCPLIB Problems*

Test MCP			Basis Time		
Problem Name	Size	Density %	LUSOL	UMFPACK	COIN
bai_haung	4,900	0.10	0.10	0.04	0.18
	19,600	0.03	1.25	0.21	15.70
	78,400	0.01	9.16	1.06	552.07
	313,600	0.00	103.47	6.12	-
bert_oc	5,000	0.05	0.02	0.05	0.02
	50,000	0.01	0.29	0.56	0.44
	500,000	0.00	5.56	8.47	26.91
bratu	5,625	0.09	0.95	0.54	2.33
	22,500	0.02	16.60	5.29	97.54
	90,000	0.01	278.58	49.71	5918.56*
dirkse2	64,001	0.00	29.45	17.90	26.52
	216,001	0.00	395.49	261.64	363.42
	512,001	0.00	2287.89	1461.80	2181.53
dongbai	7500	0.08	4.64	0.60	2.21
	14,700	0.04	51.97	2.59	43.89*
	30,000	0.02	658.19	10.42$^\diamond$	255.01
opt_cont	288	5.59	0.002	0.007	0.005
	8,192	0.21	0.12	0.17	0.15
	32,032	0.05	0.57	0.76	0.82
	160,032	0.01	5.17	4.95	9.27
	320,032	0.01	17.32	12.52	30.66
	480,032	0.00	31.47	22.51	-
obstacle(1)	10,000	0.05	0.33	0.31	0.64
(2)			0.85	0.39	1.40
(3)			0.67	0.63	1.45
(4)			0.75	0.64	36.46*
(5)			0.87	0.35	1.69
(6)			1.25	0.65	154.19*
(7)			1.06	0.59	29.08*
(8)			1.17	0.43	3.05
obstacle(1)	40,000	0.01	5.61	3.16	22.98
(2)			8.74	3.54	48.76
(3)			11.28	5.65	3402.44*
(4)			12.15	5.65	1746.77*
(5)			8.39	2.62	52.92
(6)			31.46	9.00	262.90
(7)			18.34	5.74	2272.43*
(8)			10.69	2.44	120.46

A closer examination at the distribution of the basis time taken by the UMFPACK and LUSOL options is given in Table 9, in which we randomly generate three sets of LCPs and total their basis time for each set. (The COIN option is not compared here because, in general, it performs more solves and factors than do the other options.) Both options took a similar number of iterations to solve each LCP in the test sets. Problem Set I comprise 50 LCPs of relatively smaller-scale (average number of cols/rows = 1168) and higher density (average density = 4.5%). While LUSOL spends slightly more time in the factor and update routines, its speed in the solve routines outweighs the other statistics and makes it more efficient than UMFPACK. When the problem size increases in Set II

(20 LCPs with average number of cols/rows = 3795 and density = 4.4%), the advantage of UMFPACK over LUSOL in factor and update routines becomes more significant, and it outperforms LUSOL. Problem Set III is generated with the same size as Set II but with reduced density (of 0.3% on average) and slightly different matrix structure. For these systems, LUSOL works better than UMFPACK because the factor time taken by the two options is comparable, and even though LUSOL takes more time performing updates, it is much faster in its solve routines. Similar observations are obtained with the basis time for the MCPLIB problems. In particular, LUSOL spends 82%, 13%, and 5% of basis time in factor, update, and solve routines, where UMFPACK spends 51%, 4%, and 45% of basis time in these routines, respectively.

Table 9: *Distribution of Basis Time for Randomly Generated LCPs*

	LUSOL		UMFPACK	
Test Set	Factor/Update/Solve Time	Basis Time	Factor/Update/Solve Time	Basis Time
I	22.89/7.00/2.26	32.15	17.24/1.67/20.66	39.57
II	821.16/642.69/47.75	1511.61	172.59/88.18/324.08	584.85
III	1.11/11.14/2.86	15.08	1.86/4.39/33.56	39.77

6 Discussion

We have shown that incorporating the UMFPACK package as an alternative to the LUSOL basis package in PATH improves the solution of large-scale problems in that the time spent in the basis package (hence in the overall program) is reduced and the reliability or accuracy of the solution is increased. This advantage is most significant when the solution process is dominated by the crash procedure. However, on general sparse problems requiring large numbers of pivots in the complementarity subproblems or on small problems, the LUSOL basis package tends to be more reliable and more efficient than UMFPACK, in part because of the efficiency of the solve routines in LUSOL. Therefore, the LUSOL basis package remains the default basis option in the PATH code, and it is advantageous to have interchangeable basis packages in PATH, since their performances vary with different problem characteristics.

The alternative of using the COIN basis package is motivated by COIN-LP's better performance than the LUSOL-based MINOS/SNOPT in solving LPs. In solving the MCPLIB problems, the COIN basis package is the most effective of the three options in the number of successes with the help of the linear refinements and scaling procedures. The trade-off, however, is an increase in the solution time. It is possible that utilizing these methods more generally within PATH would improve robustness with other basis routines. In solving large-scale systems, COIN is less efficient than LUSOL and UMFPACK; and for several large instances, we observe numerical instability with COIN. As an open source code, COIN-OR utilities have the advantage of being under constant modification and improvement. Hence we believe that the COIN has the potential to perform better in the future.

References

[1] R. H. Bartels and G. H. Golub, *The simplex method of linear programming using the LU decomposition*, Communications of the ACM, 12 (1969), pp. 266–268.

[2] S. C. Billups, S. P. Dirkse, and M. C. Ferris, *A comparison of large scale mixed complementarity problem solvers*, Computational Optimization and Applications, 7 (1997), pp. 3–25.

[3] R. M. Chamberlain, M. J. D. Powell, C. Lemarechal, and H. C. Pedersen, *The watchdog technique for forcing convergence in algorithms for constrained optimization*, Mathematical Programming, 16 (1982), pp. 1–17.

[4] A. K. Cline, *Two subroutine packages for the efficient updating of matrix factorizations*, Tech. Rep. TR-68, Department of Computer Sciences, The University of Texas, Austin, TX, 1977.

[5] T. A. Davis, *Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, 30 (2004), pp. 196–199.

[6] ——, *A column pre-ordering strategy for the unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, 30 (2004), pp. 165–195.

[7] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM Journal on Matrix Analysis and Applications, 18 (1997), pp. 140–158.

[8] ——, *A combined unifrontal/multifrontal method for unsymmetric sparse matrices*, ACM Transactions on Mathematical Software, 25 (1999), pp. 1–19.

[9] S. P. Dirkse and M. C. Ferris, *MCPLIB: A collection of nonlinear mixed complementarity problems*, Optimization Methods and Software, 5 (1995), pp. 319–345.

[10] ——, *The PATH solver: A non-monotone stabilization scheme for mixed complementarity problems*, Optimization Methods and Software, 5 (1995), pp. 123–156.

[11] ——, *Crash techniques for large-scale complementarity problems*, in Complementarity and Variational Problems: State of the Art, M. C. Ferris and J. S. Pang, eds., SIAM, Philadelphia, Pennsylvania, 1997, pp. 40–61.

[12] S. K. Eldersveld and M. A. Saunders, *A block-LU update for large-scale linear programming*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 191–201.

[13] R. E. Ericson and A. Pakes, *Markov-perfect industry dynamics: A framework for empirical work*, Review of Economic Studies, 62 (1995), pp. 53–82.

[14] M. C. Ferris, C. Kanzow, and T. S. Munson, *Feasible descent algorithms for mixed complementarity problems*, Mathematical Programming, 86 (1999), pp. 475–497.

[15] M. C. Ferris and S. Lucidi, *Nonmonotone stabilization methods for nonlinear equations*, Journal of Optimization Theory and Applications, 81 (1994), pp. 53–71.

[16] M. C. Ferris and T. S. Munson, *Interfaces to PATH 3.0: Design, implementation and usage*, Computational Optimization and Applications, 12 (1999), pp. 207–227.

[17] ——, *Preprocessing complementarity problems*, in Complementarity: Applications, Algorithms and Extensions, Volume 50 of Applied Optimization, M. C. Ferris, O. L. Mangasarian, and J. S. Pang, eds., Kluwer Academic Publishers, 2001, pp. 143–164.

[18] M. C. Ferris and J. S. Pang, *Engineering and economic applications of complementarity problems*, SIAM Review, 39 (1997), pp. 669–713.

[19] A. Fischer, *A special newton-type optimization method*, Optimization, 24 (1992), pp. 269–284.

[20] R. Fletcher and S. P. J. Matthews, *Stable modifications of explicit LU factors for simplex updates*, Mathematical Programming, 30 (1984), pp. 267–284.

[21] J. J. H. Forrest and J. A. Tomlin, *Updated triangular factors of the basis to maintain sparsity in the product form simplex method*, Mathematical Programming, 2 (1972), pp. 263–278.

[22] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, *Sparse matrix methods in optimization*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 562–589.

[23] P. E. Gill, W. Murray, and M. J. Wright, *Numerical Linear Algebra and Optimization*, vol. 1, A. M. Wylde, Redwood City, CA, 1991.

[24] P. R. Gill, W. Murray, M. A. Saunders, and M. H. Wright, *Maintaining LU factors of a general sparse matrix*, Linear Algebra and Its Applications, 88/89 (1987), pp. 239–270.

[25] L. Grippo, F. Lampariello, and S. Lucidi, *A nonmonotone line search technique for Newton's method*, SIAM Journal on Numerical Analysis, 23 (1986), pp. 707–716.

[26] ——, *A class of nonmonotone stabilization methods in unconstrained optimization*, Numerische Mathematik, 59 (1991), pp. 779–805.

[27] R. Lougee-Heimer, *The common optimization interface for operations research*, IBM Journal of Research and Development, 47 (2003), pp. 57–66.

[28] B. A. Murtagh and M. A. Saunders, *MINOS 5.5 User's Guide*, report sol 83-20r, Stanford University, 1998.

[29] A. Pakes and P. McGuire, *Computing Markov-perfect Nash equilibria: Numerical implications of a dynamic differentiated product model*, Rand Journal of Economics, 25 (1994), pp. 555–589.

[30] D. Ralph, *Global convergence of damped Newton's method for nonsmooth equations via the path search*, Mathematics of Operations Research, 19 (1994), pp. 352–389.

[31] J. K. Reid, *Fortran subroutines for handling sparse linear programming bases*, Tech. Rep. AERE R8269, Atomic Energy Research Establishment, Harwell, England, 1976.

[32] ——, *A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases*, Mathematical Programming, 24 (1982), pp. 55–69.

[33] S. M. Robinson, *Normal maps induced by linear transformations*, Mathematics of Operations Research, 17 (1992), pp. 691–714.

[34] M. A. Saunders, *LUMOD.* Fortran software for updating dense LU factors, http://www.stanford.edu/group/SOL/software/lumod.html, 2000.

[35] F. Tin-Loi and M. C. Ferris, *Holonomic analysis of quasibrittle fracture with nonlinear softening*, in Advances in Fracture Research, B. L. Karihaloo, Y. W. Mai, M. I. Ripley, and R. O. Ritchie, eds., vol. 2, Pergamon Press, Oxford, 1997, pp. 2183–2190.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.