An Implementation of the Minimum-Degree Algorithm Using Simple Data Structures

Robert J. Vanderbei

April 20, 1994

Abstract

The most popular ordering algorithm for reducing the number of arithmetic operations in the Cholesky factorization of a symmetric sparse matrix is the minimum-degree algorithm. Until now, efficient implementations of this algorithm have been based on the so-called implicit model of Gaussian elimination since the straight-forward explicit model has unpredictable and potentially large temporary storage requirements. In this paper we describe a new implementation of the minimum-degree algorithm based on the explicit model. The problem of unpredictable memory requirements is solved by using C (and its utilities for dynamic memory allocation) as the programming language. The concern that the temporary storage requirement may be large turns out not to be a problem since we show that it never exceeds the amount of memory which is subsequently needed to store the nonzeros in the Cholesky factor. One attraction of using the explicit model is that the code is quite short and easy to understand. Also, efficient implementations of the implicit model often deviate somewhat from the definition of the minimum-degree algorithm whereas the explicit model is exactly a minimum-degree ordering algorithm. We include comparisons between implementations of the explicit model and the implicit model. The explicit model turns out to run as fast as the implicit model and it generates orderings which are generally just as good as those obtained with the implicit model.

1 Introduction.

Consider the problem of solving a sparse, symmetric, positive semi-definite, linear system of m equations:

$$Ax = b. \tag{1.1}$$

One method for solving such a system is based on factoring the matrix A into the product of a lower triangular matrix L (with ones on the diagonal), a diagonal matrix D and the transpose of L:

$$A = LDL^T. \tag{1.2}$$

Given this factorization, (1.1) can be solved by sequentially solving the following three problems:

$$Lz = b, \tag{1.3}$$

$$Dy = z, \tag{1.4}$$

and

$$L^T x = y. \tag{1.5}$$

These three systems of equations are easy to solve (compared to (1.1)) since (1.3) can be solved by *forward substitution*, (1.4) can be solved by componentwise division, and (1.5) can be solved by *backward substitution*. The triangular matrix L is called the *Cholesky factor*, and the process of forming L and D is called *Cholesky factorization*. The reader should note that Cholesky factorization is actually equivalent to Gaussian elimination. Indeed, the forward substitution corresponds to the first part of Gaussian elimination where the elements below the diagonal are zeroed, solving system (1.4) corresponds to normalizing each of the remaining equations so that the diagonal coefficients are one and finally solving (1.5) by backward substitution is the usual backward substitution that takes place as the last stage in Gaussian elimination.

A matrix which has a large number of zero elements is called a *sparse* matrix. Ignoring the possibility of exact numerical cancellation, it is easy to see that the Cholesky factor L of A will have nonzeros in all the same locations as in A plus certain additional nonzeros. These additional nonzeros are called *fillin*. The amount of fillin can be very large. However, by judiciously permuting the rows and columns of A, it is possible to obtain a system of equations equivalent to (1.1) but for which the Cholesky factor suffers only the minimal necessary amount of fillin. While finding the optimal permutation would be much more time consuming than simply solving (1.1) as given, there are several fast heuristics which come close to the optimal ordering.

The most popular heuristic is the *minimum-degree algorithm* (see e.g. [Duf82], [EGSS82], [GL81], [GL80], or [Ros73]). Using the correspondence between Cholesky factorization and Gaussian elimination, this heuristic is easy to motivate. Indeed, suppose that we have zeroed the elements below

the diagonal in the first $k-1$ columns of A. The amount of fillin created by eliminating the next column depends on the number of nonzeros in that column — lots of nonzeros is likely to translate into lots of fillin. Hence, before processing column k, it makes sense to look at all the remaining columns and pick one which has the minimum number of remaining nonzeros, and permute rows and columns so that this column becomes column k. The number of remaining nonzeros in a column is called its *degree* and hence this ordering scheme is one which at each stage picks a minimum-degree column.

In this paper we describe an implementation of the minimum-degree algorithm which uses the so-called explicit model of elimination. In the next section, we describe the explicit model. In Section 3, we describe the data structures used to implement the minimum-degree ordering algorithm using the explicit model. Finally, in Section 4, we compare the performance of the explicit model with the usual implicit model.

Acknowledgement. I would like to thank Jun-Min Liu for allowing me to use his implementation of the minimum-degree algorithm based on the implicit model.

2 The Explicit Model of Gaussian Elimination.

Let A be a sparse symmetric matrix. The *ordered graph* $G^A = (X^A, E^A)$ of A with vertex set X^A and edge set E^A is defined by

$$X^A = \{1,\ 2,\ \ldots,\ m\}$$

and

$$E^A = \{\{i, j\} | a_{ij} \neq 0\}.$$

The graph G^A captures the nonzero structure of the matrix A. The effect of Gaussian elimination on the nonzero structure can be described in terms of transformations of G^A. Indeed, zeroing all the elements below the diagonal in the first column of A induces the following changes to G^A:

1. node 1 is deleted from the graph;
2. edges are added so that all the nodes that were adjacent to 1 are now pairwise adjacent.

Let $G_0 = G^A$ and let G_1 denote the graph obtained after zeroing the elements in the first column. Proceeding with the processing of subsequent

columns results in a sequence of graphs

$$G_0 \rightarrow G_1 \rightarrow \cdots \rightarrow G_m = \emptyset.$$

The graphs G_j, $j = 0, ..., m$ are called the *elimination graphs* for A. The set of neighbors of j in G_{j-1} determines the set of nonzeros in the j^{th} column of the Cholesky factor L.

In terms of elimination graphs, the minimum-degree algorithm is easy to describe. Instead of eliminating nodes in sequential order, they are chosen sequentially so that at each stage the chosen node is of minimum degree. An implementation of the minimum-degree algorithm which explicitly calculates the elimination graphs is called an *explicit implementation.*

3 Explicit Implementation of the Minimum-Degree Algorithm

The matrix A is stored sparsely in three arrays, `A`, `IA`, and `KA`. Let `nz` denote the number of nonzeros in A. The array `A` contains `nz` double precision numbers which represent the nonzeros of A listed one after the other starting with column 1, followed by column 2 and so on. The integer array `IA` contains the row indices for each of the nonzeros stored in `A`. Finally, the array `KA` contains `M+1` integers which indicate where each new column starts in the arrays `A` and `IA`.

The elimination graph is stored as an array of arrays: for `i=1,2, ..., m, NBRS[i]` is an array containing the the index for each of the neighbors of node `i`. The length of `NBRS[i]` is stored in `DEGREE[i]`, where `DEGREE` is an array of `m` integers.

Using these data structures, the explicit implementation of the minimum-degree algorithm proceeds as follows:

1. Allocate enough space to store G_0 in `NBRS`.
2. Initialize `NBRS` to contain G_0.
3. Initialize linked list of uneliminated nodes.
4. While uneliminated nodes remain:
   (a) Compute current minimum degree and select a node achieving this minimum — call it `node`.

(b) Transfer NBRS[node] into IA allocating extra space if necessary. Update KA to reflect the extra column just added in IA. (The original contents of IA and KA are overwritten.)

(c) Remove node from each neighbor's neighbor list and decrement the corresponding element of DEGREE.

(d) Add links between each pair of neighbors of node which don't already have a link, allocating extra space as necessary and updating the array DEGREE appropriately.

(e) Delete node from linked list of uneliminated nodes.

(f) Free the space used to store NBRS[node].

As presented, this algorithm is not as efficient as the current state-of-the-art implementations (such as [EGSS82] or [GLN80]). The reason is that state-of-the-art implementations perform mass elimination of all nodes that are indistinguishable from the minimum-degree node. The concept of indistinguishable node is discussed at length in [GL81]. Here we settle for statements of the definition and of the important properties of indistinguishable nodes. Two nodes ν and μ are called *indistinguishable* if the set consisting of ν and its neighbors coincides with the set consisting of μ and its neighbors. After eliminating a minimum-degree node, each of the nodes which are indistinguishable from this minimum-degree node can be eliminated in any order (they are minimum-degree nodes in each of the subsequent elimination graphs) and they don't generate any further fillin. Hence, for these nodes, Step (4d) in the above algorithm can be skipped. Since this is the most time consuming step, skipping it can entail a significant speedup in the algorithm.

The explicit implementation of the minimum-degree algorithm which incorporates mass elimination of indistinguishable nodes proceeds as follows:

1. Allocate enough space to store G_0 in NBRS.

2. Initialize NBRS to contain G_0.

3. Initialize linked list of uneliminated nodes.

4. While uneliminated nodes remain:

   (a) Compute current minimum degree and select a collection of indistinguishable nodes achieving this minimum. Store a list of these indistinguishable nodes in an array INDST[k], k=1,...,numindst

and store a list of the neighbors of these indistinguishable nodes in the array `DST[k], k=1,...,numdst`.

(b) For each indistinguishable node (`INDST[k], k=1,...,numindst`), transfer the subsequent indistinuguishable nodes (`INDST[kk], kk=k+1,...,numindst`) and all of the distinguishable nodes into `IA` allocating extra space if necessary. Update `KA` to reflect the extra columns just added in `IA`.

(c) For each distinguishable node (`DST[k], k=1,...,numdst`), remove all the indistinguishable nodes from its neighbor list and decrement `DEGREE[DST[k]]` appropriately.

(d) Delete each indistinguishable node from the linked list of uneliminated nodes.

(e) For each indistinguishable node (`INDST[k], k=1,...,numindst`), free the space used to store `NBRS[INDST[k]]`.

(f) Add links between each pair of distinguishable nodes which don't already have a link, allocating extra space as necessary and updating the array `DEGREE` appropriately.

In each of the explicit algorithms presented above, once a node is eliminated, the memory required to store that node in the elimination graph `NBRS` is freed. If we were to assume that this space is not dynamically freed, then the memory requirement for `NBRS` would only grow and at the end `NBRS` would represent the ordered graph of L. Hence, it would have $nonz(L)$ edges. In `NBRS`, each edge requires two integers of storage. So, without dynamic freeing, the total memory requirement for `NBRS` would be 2 $nonz(L)$ integers. After the minimum-degree algorithm is complete, storage must be allocated to hold the nonzeros in L. This requires $nonz(L)$ reals. If storage for `NBRS` is freed before storage for the nonzeros in L is allocated and if a real takes twice as much memory as an integer then storing the nonzeros in L will take at least as much memory as `NBRS`. Therefore, even though the memory requirement for the algorithms presented here is "unpredictable", it is guaranteed not to be greater than what will subsequently be necessary to store the nonzeros of L.

Steps (4c) and (4f) are the most time consuming steps in the algorithm. Step (4c) could be speeded up substantially by using a more sophisticated data structure for the elimination graph. The data structure we have chosen being simply an array of lists of neighbors does not facilitate removing nodes — to remove an element from a node list requires time proportional to the

length of the list. Using something more sophisticated than a list can reduce this time to the logarithm of the length of the list. However, more sophisticated data structures would entail using more memory and so we would no longer be able to guarantee that the storage requirement be dominated by the number of nonzeros in L. For this reason together with the fact that the code would be somewhat more complicated, we have chosen not to pursue this enhancement.

The most time consuming part of Step (4f) is the allocation of more memory when needed. By allocating somewhat more than what is immediately necessary, we can greatly reduce the number of times that the reallocation needs to be done. For naive implementations, this enhancement also entails losing the guarantee that the storage requirement be dominated by the number of nonzeros in L. However, since we dynamically free memory as we eliminate nodes, it is possible to keep track of how much memory has been freed and parcel out no more than this amount. We have chosen to use this enhancement.

4 Implementation and Testing.

We programmed an explicit implementation of the minimum-degree algorithm incorporating the mass elimination of indistinguishable nodes enhancement and the infrequent reallocation of memory enhancement. The code is entirely written in C. Because it is so simple, it required only 225 lines of code.

We compared our explicit implementation with an implicit implementation developed by Jun-Min Liu (and modified slightly by the author). Both codes were compiled and tested on a Silicon Graphics IRIS workstation, model 4D80/GT, with 8 Mbytes of main memory, one processor and a 16.7 MHz clock. The compilers `f77` and `cc` were used with optimization level `-O2 -Olimit 800`. Times were measured on the UNIX command line with the `time` command.

For test problems, we used the linear programming problems available from NETLIB [Gay85]. The matrix we reordered is the positive semidefinite matrix AA^T where A is the constraint matrix for the given linear programming problem. The times we report include the time to read in the linear programming problem from an industry standard MPS format file and to form AA^T. This adds a fixed generally small overhead to the times for both codes.

The results are shown in Tables 1 and 2. Generally the explicit and implicit implementations generated orderings which yielded similar results in terms of nonzeros in the Cholesky factor and resultant number of arithmetic operations in the computation of L. Some notable exceptions where the explicit implementation was significantly better are CYCLE (with 5,092,396 vs. 8,448,794 arithmetic opts), D2Q06C (with 23,198,556 vs. 33,959,398), PILOT.WE (with 839,370 vs. 969,344) and PILOT (with 46,660,532 vs. 53,512,426). Exceptions in the other direction are PEROLD (with 2,451,566 vs. 1,908,582), PILOT.JA (with 6,736,252 vs. 5,671,524), PILOT4 (with 859,058 vs. 732,184) and PILOTNOV (with 6,201,088 vs. 5,433,922).

In terms of execution time, most often the times were comparable but on occasion a significant disparity appeared. For those problems where a disparity existed, it usually favored the implicit implementation. For example there were 5 instances (BNL2, D2Q06C, ETAMACRO, PEROLD, SEBA) where the implicit implementation was at least twice as fast as the explicit implementation, while there were only 2 instances (CZPROB and DEGEN3) of the reverse. The largest disparity occured with CZPROB where the explicit implementation was more than 3 times faster. It should be pointed out that the implicit implementation was developed over a long period of time and has been well optimized for speed of execution. In contrast, the explicit implementation described here has hardly been optimized at all.

Problem Name	$nonz(L)$		Arith. Ops.		Time (M:S)	
	Explicit	Implicit	Explicit	Implicit	Explicit	Implicit
25FV47	34215	33232	2596730	2382894	5.6	3.6
80BAU3B	42121	39827	2504764	2213852	25.7	16.9
ADLITTLE	355	355	2394	2394	0.1	0.1
AFIRO	80	80	188	188	0.0	0.0
AGG	4756	4695	193602	188738	0.8	0.7
AGG2	21297	20918	1060444	1008548	2.4	1.7
AGG3	21297	20918	1060444	1008548	2.4	1.7
BANDM	4358	4358	77078	77078	0.8	0.8
BEACONFD	2727	2728	61338	61398	0.8	1.4
BLEND	940	924	13750	13046	0.1	0.1
BNL1	12236	12507	493370	534588	2.5	2.2
BNL2	83072	84865	12747212	13807840	21.3	7.6
BOEING1	7155	7053	174636	167822	1.5	1.9
BOEING2	2571	2574	54034	54126	0.4	0.4
BORE3D	2861	2863	60680	60748	0.5	0.5
BRANDY	3252	3231	91892	89904	0.6	0.6
CAPRI	5569	5638	185132	188564	0.8	0.6
CYCLE	72765	87000	5092396	8448794	14.3	8.8
CZPROB	7462	7462	82798	82798	5.6	18.0
D2Q06C	138553	162027	23198556	33959398	27.9	13.4
DEGEN2	16210	15894	940806	896232	3.0	3.3
DEGEN3	119458	119202	15068516	15048958	48.5	1:55.8
E226	3407	3448	71116	73140	0.7	0.7
ETAMACRO	15079	15399	954990	998564	2.4	1.1
FFFFF800	17988	18291	862436	896052	3.4	2.7
FINNIS	6332	6302	119734	118114	1.3	1.4
FORPLAN	3587	3544	122528	118414	1.0	1.0
GANGES	28032	30231	1411094	1600776	6.1	3.1
GFRD-PNC	1533	1541	3050	3154	1.2	1.1
GREENBEA	84448	85734	5988850	6327178	23.4	24.1
GREENBEB	84448	85734	5988850	6327178	23.4	24.4
GROW15	5790	5790	108680	108680	1.5	1.4
GROW22	8590	8590	161880	161880	2.3	2.0
GROW7	2590	2590	47880	47880	0.7	0.6
ISRAEL	11259	11259	978014	978014	1.1	0.9
KB2	460	460	5370	5370	0.0	0.0
LOTFI	1722	1736	26212	26940	0.3	0.3
NESM	24071	24598	1466536	1549550	6.8	4.6
PEROLD	28552	26231	2451566	1908582	5.3	2.6
PILOT.JA	56808	53531	6736252	5671524	13.0	8.1
PILOT.WE	17097	18036	839370	969344	4.1	3.3
PILOT	199334	210608	46660532	53512426	55.9	28.6
PILOT4	14068	13106	859058	732184	2.8	2.7
PILOTNOV	55560	53033	6201088	5433922	12.8	7.2

Table 1: Computational Results (A-P).

References

[Duf82] I.S. Duff. A sparse future. In I.S. Duff, editor, *Sparse Matrices and Their Uses*. Academic Press, 1982.

[EGSS82] S.C. Eisenstat, M.C. Gursky, M.H. Schultz, and A.H. Sherman. The Yale sparse matrix package, I. the symmetric code. *Intl. J. for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[Gay85] D.M. Gay. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newslettter*, 13:10–12, 1985.

[GL80] A. George and J.W.H. Liu. A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Trans. Math. Softw.*, 6:337–358, 1980.

[GL81] A. George and J. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[GLN80] J.A. George, J.W.H. Liu, and E. Ng. User guide for SPARSPAK, Waterloo sparse linear equations package. Technical Report 78-30 (revised 1980), Dept. of Computer Science, Univ. of Waterloo, 1980.

[Ros73] D.J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R.C. Read, editor, *Graph Theory and Computing*. Academic Press, 1973.

Problem	$nonz(L)$		Arith. Ops.		Time (M:S)	
Name	Explicit	Implicit	Explicit	Implicit	Explicit	Implicit
RECIPE	667	667	10256	10256	0.1	0.1
SC105	437	442	1580	1632	0.1	0.1
SC205	882	918	3248	3644	0.2	0.2
SC50A	182	193	574	680	0.0	0.0
SC50B	185	193	644	760	0.0	0.0
SCAGR25	2508	2509	14286	14300	0.7	0.6
SCAGR7	636	637	3378	3392	0.1	0.1
SCFXM1	4400	4400	81642	81642	0.8	0.7
SCFXM2	8991	9002	169810	170176	1.9	1.6
SCFXM3	13619	13592	259992	258026	3.2	2.5
SCORPION	2099	2186	14294	16092	0.6	0.5
SCRS8	5706	5865	131666	143106	1.4	1.2
SCSD1	1315	1315	24756	24756	0.6	0.6
SCSD6	2398	2398	40114	40114	1.2	1.3
SCSD8	5482	5482	73678	73678	2.7	2.7
SCTAP1	2323	2348	24642	25294	0.6	0.5
SCTAP2	13836	13780	532984	530364	3.5	2.8
SCTAP3	17296	17296	545232	545938	5.0	3.9
SEBA	53691	53697	10420882	10421426	9.0	3.9
SHARE1B	1337	1266	18368	16098	0.3	0.3
SHARE2B	939	930	9174	9166	0.1	0.1
SHELL	3900	3855	57630	54182	1.8	1.7
SHIP04L	4428	4428	54448	54448	2.2	3.9
SHIP04S	3252	3252	39496	39496	1.5	1.8
SHIP08L	8948	8936	111672	111376	4.9	8.2
SHIP08S	5500	5460	66296	65032	2.8	3.0
SHIP12L	11193	11177	136680	136376	6.7	9.6
SHIP12S	6493	6457	75232	74296	3.6	3.4
SIERRA	11755	11730	262748	262388	3.9	3.2
STAIR	15102	14609	957432	847574	2.2	2.4
STANDATA	3028	2996	45762	43662	1.0	1.0
STANDMPS	4959	4949	97142	95946	1.3	1.6
STOCFOR1	843	843	6990	6964	0.1	0.1
STOCFOR2	25248	26846	403144	452204	8.1	4.4
TUFF	8283	8287	337476	337586	1.6	2.0
VTP.BASE	2733	2765	45148	46934	0.3	0.3
WOOD1P	18082	18082	1535528	1535528	16.4	30.7
WOODW	47493	47827	3064248	3109608	17.3	17.6

Table 2: Computational Results (R-Z).