Minimal Fill in O($n^{2.69}$) Time

Dieter Kratsch

LITA,Université de Metz,57045 Metz Cedex 01, France,
kratsch@sciences.univ-metz.fr

Jeremy Spinrad

Department of EECS, Vanderbilt University, Nashville TN 37235,
spin@vuse.vanderbilt.edu

20 December 2004

Abstract

We show that it is possible to find a minimal fill ordering of a graph in O($n^{2.69}$) time. Previous algorithms for the problem required $\Omega(n^3)$ time on dense graphs. The algorithm uses fast matrix multiplication to produce the same ordering of vertices as Tarjan's well known LEX-M algorithm.

1 Introduction

The minimal fill problem arises out of applications to performing Gaussian elimination on sparse matrices. For a symmetric matrix, rows can be mapped to vertices, and the order in which rows are chosen as pivots will correspond to a vertex ordering of the graph. Pivoting on a row can make zeros in the original matrix change into nonzero values; these new nonzero elements are called fill. In the graph model, this fill process corresponds to making the neighbors of the pivot vertex into a clique. The fill created by an ordering is defined by a process of successively removing a vertex v, and making all remaining neighbors of v into a clique.

Ideally, we would like to choose an ordering which minimizes the fill over all possible orderings. However, Yanakakis showed that this "minimum fill" problem is NP-complete [15]. This has led to a great deal of research on attempts to create orderings which have good properties, though these will not in general find the minimum fill.

One important direction of research has been an attempt to create minimal, rather than minimum, fill. A fill created by an ordering α is minimal if there is no other ordering β such that the fill created by β is a proper subset of the fill created by α.

For many minimization problems, finding a minimal (i.e., the solution is not a proper superset of any other solution) rather than a minimum solution can be accomplished in a very simple greedy fashion. By contrast, it is not obvious how to create a minimal fill in polynomial time. A number of polynomial algorithms, such as [1,8,11] for the problem have been developed, but all of these take $\Theta(mn)$ time, and can have running times proportional to n^3 on dense graphs.

We break this $\Theta(n^3)$ barrier, using fast matrix multiplication to get a running time of $O(n^{2.69})$ for the minimal fill problem. In particular, we take one of the most famous algorithms for minimal fill, the LEX-M algorithm of [11], and show that the ordering produced by LEX-M can be found in $O(n^{2.69})$ time.

2 Definitions

Given an ordering $\sigma = v_1, v_2, \ldots, v_n$ of the vertices of a graph $G = (V, E)$, we construct a sequence of graphs $G = G_0, G_1, G_2, ..., G_{n-1}, G_n = G_f$ as follows. For $1 \leq i \leq n$ let G_i be the graph obtained from G_{i-1} by adding edges between every pair of vertices v_j, v_k such that $j, k > i$ and v_j and v_k are neighbors of v_i in G_{i-1}. The graph $G_f = G_n$ will be called the filled graph for the ordering σ, and the fill caused by σ is the set of edges which are in G_f but not in G.

The fill caused by an ordering σ corresponds to zero entries which become nonzero during Gaussian elimination of a symmetric matrix using σ as pivot ordering. An early paper noting the correspondence between this "elimination game" on a graph and creating nonzero entries during Gaussian elimination is [9]. For this reason, the problem of choosing a vertex ordering which minimizes the amount of fill has been studied carefully. The minimum fill problem was one of the original open problems in [4], and was shown to be NP-complete in [15]. The special case of determining whether a graph has a perfect elimination order, i.e. an ordering which causes no fill, is equivalent to the graph having no chordless cycles of length greater than 3. Graphs without chordless cycles of length greater than 3 are called chordal graphs; algorithms for recognizing chordal graphs and producing a perfect elimination ordering in linear time can be found in [11,14].

The minimal fill problem studied here requires finding an ordering of the vertices such that the fill produced by this ordering does not properly contain the fill produced by any other ordering of the vertices.

The paper will make use of the "path lemma" of Rose, Tarjan and Lueker.

Path Lemma [11,10]: Consider any ordering of the vertices of a graph. There is an edge (v, w) of the fill graph produced by this ordering if and only if there is a path P from v to w such that every internal vertex of P precedes both v and w in the ordering.

Our algorithm will maintain a partition of the vertices into sets, where the order between sets is fixed, while the order of vertices within sets is not yet determined. A fill path between v and w will be a path from v to w which uses only intermediate vertices which are in sets which come before the sets containing v and w in the ordering; the path lemma tells us that any vertices connected by a fill path will be adjacent in the filled graph.

We let n^α denote the best known time bound to multiply two n by n matrices. Currently $\alpha = 2.376..$

3 Algorithm

Algorithm LEX-M for solving the minimal fill problem [11] works as follows. At each step, the vertices of G are partitioned into an ordered list of vertex subsets; the initial partition places all vertices in a single set. Any vertex v from the last current subset is chosen to be removed from the subset and placed in the output list. We then use v to partition each subset in the current list as follows.

Let S be an arbitrary subset in the list. Let precede(S) be the set of vertices which are in subsets which come before S in the current partition list. Let $S_2 \subseteq S$ be the set of vertices which are reachable from v either directly by an edge, or reachable from v on a path P using only intermediate vertices from precede(S). Let S_1 be S - S_2. For each S, we modify the partition by splitting S into S_1, S_2, placing S_1 immediately before S_2. An implementation showing that this partitioning with respect to v can be done in O(m) time is given in [11], leading to an O(nm) algorithm for finding a minimal fill ordering.

The algorithm differs from LEXBFS, a well known algorithm originally designed for recognizing chordal graphs, in that LEXBFS divides each set into neighbors and nonneighbors of v in G, while LEX-M divides into neighbors and nonneighbors of v in a graph which has been augmented by edges which are known to be contained in the fill. We modify LEX-M by computing this 'known fill' periodically, rather than generating it each time when a vertex is placed on the output list.

In our implementation of the LEX-M algorithm, the ordered list of vertex subsets of the original implementation is restructured in an ordered list of 'groups'. Each group is an ordered collection of consecutive vertex subsets, and the groups form a partition of the vertex subsets. Thus, initially, there is one group containing a single set, where this set consists of all vertices in G. Additionally, our implementation maintains the invariant that each group will contain at most one large subset, where large will mean that its cardinality is greater than f(n), where f is a function to be chosen later. For a group T, let precede(T) be the set of vertices which are in groups which come before T in the current partition list. We will maintain for all vertices x, y known fill edges which come from a path from x to y in which all intermediate vertices precede both the group containing x and the group containing y; this fill is computed whenever a new group is created. Fill edges incident to a vertex w which uses paths involving intermediate vertices from the same group as w will be generated only when a new vertex is placed on the output list, as in the original implementation of the LEX-M algorithm.

Theorem 1: Let T be a group. All fill between the vertices in T and vertices of $V(G)$ - precede(T) which uses only vertices of precede(T) as intermediate vertices on fill paths can be computed in O(n^α) time.

Proof: Find the connected components of the subgraph H induced by precede(T). For each x, t such that t is in T and x is G - precede(T), there is a fill path from x to t satisfying the conditions of the theorem if and only if x and t have neighbors in a common component of H. Create a matrix M with a row for each vertex of G - precede(T), and a column for each connected component of H. There is a 1 in position $M[x, h]$ if x has an edge to a vertex in component h, and a 0 otherwise. It is clear that t and x have edges to a common component if and only if $MM^T[t, x]$ is nonzero, so the fill from T using only intermediate vertices in precede(T) can be computed in O(n^α) time.

□

We define G' to be a graph consisting of the vertices and edges of G, and all fill edges (x, y) where intermediate vertices on fill paths come from groups which do not contain x or y. The algorithm of [11] could also use the edges of G' rather than G in its computation, but does not compute any of these edges when a set is split into subsets.

When a vertex v is chosen for the output list, we will divide each subset of the current partition list using the following technique. For each subset S, partition S into S_1, S_2, where S_2 contains the vertices of S which are either adjacent to v in G', or vertices which have a path from v in G' using only intermediate vertices which are in the group containing S, and which come from sets which precede S in the partition list.

Theorem 2: Suppose the subsets are ordered as in LEX-M before v is placed in the output list. Every subset of vertices is split by the above procedure exactly as in LEX-M.

Proof: Let S_1, S_2 be the partition created by LEX-M. Consider any path P $vx_0x_1..x_ks_2$ to s_2 in S_2 found during LEX-M. Let $x_i..x_j$ be any maximal consecutive set of vertices of P which come from groups which precede S. There will be an edge from x_{i-1} to x_{j+1} in G', since edges of G' are updated whenever a new group is formed, which implies that the path from v to s_2 will also be found by the algorithm.

Consider any path $vy_0y_1..y_js_2$ found by the new algorithm. Each edge (u, v) of G' which is not in G was created when a new group was formed, because there was a path from u to v using intermendiate vertices only in earlier groups. Therefore, for any edges of G' not in G, we can insert the path using only vertices which precede this group. Thus, the path from v to s_2 will also be found by LEX-M.

The sets S_2 found by the two algorithms are identical, and we will find the same partition list using both algorithms.

$\square$

Theorems 1 and 2 give us the outline of an algorithm. Whenever a new group T is created, we use matrix multiplication to compute all fill (t, x) where t is in T and x is in G - precede(T), and all intermediate vertices used in the fill path are in precede(T), and add the fill edges to G'. As in LEX-M, we select a vertex to go last, and split all subsets, but in our implementation we use only intermediate vertices within a single group for computing fill edges at this time.

We have yet to define a group precisely (and thus to determine when a new group is created), and to present an algorithm for splitting subsets of vertices inside groups using paths which involve intermediate vertices from the same group as the subset of vertices being split.

Let f(n) be any function. Keep track of the size T_g of the group when the group was created. If a step of the algorithm causes the size of the largest subset in the group to be at most T_g - f(n), we decompose the group into three subgroups consisting of T_1 = all vertices sets of T before the largest subset, S = the largest subset, T_2 = all vertex subsets after the largest subset. Note that this decomposition rule guarantees the invariant of our algorithm that each group contains at most one vertex subset of size larger than f(n), since splitting all subsets of a group may create at most two large subsets (from splitting the unique large subset), and the two large subsets would be immediately placed in different groups. In addition, we will create three new groups as above if the size of the largest set becomes smaller than f(n); this step is not strictly necessary to achieve our time bound, but no further group subdivision is necessary on groups of size smaller than f(n).

The intention of the splitting process above is to ensure the following properties. We want to make sure that the total number of groups created by the algorithm is $\mathrm{O}(n/f(n))$. We also want to make sure that in any group, no more than f(n) vertices are outside the largest subset of the group; this will eventually allow us to take time proportional to the square of these set sizes in a crucial step. As noted above, the process guarantees that the second property remains true throughout the algorithm, and the following lemma shows that the first property is also true.

Lemma 1: There are $\mathrm{O}(n/f(n))$ new groups created during a run of the algorithm.

Proof: Each time a group is created, either at least f(n) vertices are placed in subgroups which are of size smaller than f(n), or at least two subgroups of the group are of size at least f(n). The first type of split can clearly occur at most $n/f(n)$ times, since none of the f(n) vertices placed in small subgroups will ever be split again into smaller groups.

We can view the process of splitting into groups as a tree, where subgroups created become children of the subgroups they were formed from. All internal nodes correspond to subgroups of size at least f(n). We know from above that the number of groups of size at most f(n) is $\mathrm{O}(n/f(n))$. The number of groups of size greater than f(n) which are not subdivided into smaller groups is clearly $\mathrm{O}(n/f(n))$, since these groups must be vertex disjoint. Therefore, the number of leaves of the tree of subgroups is $\mathrm{O}(n/f(n))$. Since internal nodes all have at least 2 children, the number of internal nodes is also $\mathrm{O}(n/f(n))$. Therefore, there are $\mathrm{O}(n/f(n))$ steps of the algorithm which create new groups. $\square$

Each time a group is created, we spend $\mathrm{O}(n^\alpha)$ time to find fill involving this group. Thus, the total time spent in creating groups is $\mathrm{O}(n^{1+\alpha}/f(n))$.

We now describe how to split subsets when v is selected to be placed in the output list. Part of the algorithm will use spanning trees for particular subgroups of groups, which will be maintained dynamically; i.e., the trees will be maintained as edges are added and deleted. Using algorithms such as those in [6] for maintaining these spanning trees, this will not be a bottleneck step of our algorithm. It is easier to understand which edges are used in these spanning trees when the need for them arises; we will assume in Theorem 3 that these trees are available, and discuss the cost of maintaining these trees as part of Theorem 4.

Theorem 3: When v is chosen to be the next vertex in the output list, the subsets created by LEX-M can be found in $\mathrm{O}(nf(n))$ time.

Proof: Consider each group T separately. Let S be the largest subset in T.

Let T_1 be the set of those vertices of T which precede S, and let T_2 be the set of those vertices of T which come after S. Note that $|T_1|$ and $|T_2|$ are at most $\mathrm{f}(n)$.

We will use the standard method, as in [11], to split all subsets in T_1; this will take time proportional to $|V(T_1)|^2$, and is thus $\mathrm{O}(f^2(n))$. We note that this could easily be reduced to time which is linear in the size of the subgraph induced by T_1, but this would not affect the overall time complexity of the algorithm. For completeness, we describe the steps below.

Mark all neighbors of v (in G'). We will maintain a set R (for reachable) of vertices which are reachable from v using only vertices in subsets which have already been considered. Go through subsets of T_1 from the first set in the ordering to the last; let X be the current subset. Divide X into marked vertices, which are placed in X_2, and unmarked vertices, which are placed in X_1. Place all vertices of X_2 on a queue of vertices and add these vertices to R. While there are vertices on this queue, let q be the next vertex on the queue. For each vertex t_1 in T_1, if q is adjacent to t_1 and t_1 comes after X in the ordering, mark t_1. If q is adjacent to t_1 and t_1 does not come after X in the ordering, add t_1 to the queue and to R if t_1 is not already in R.

We consider each possible edge (i, j) twice (when i is added to R, and when j is added), so the work to split all sets is proportional to the square of the number of vertices in T_1.

To split S, we find the connected components in the subgraph of G' induced by T_1, and find which components have edges incident to v in G'. For each vertex in these components, we mark the vertices of S which are neighbors of this vertex in G'; marked vertices (as well as those vertices of S which are neighbors of v in G') are placed in S_2, and unmarked vertices in S_1. The total time is clearly $\mathrm{O}(f^2(n) + |S|f(n))$.

The following technique to split T_2 is similar to the splitting of T_1 but requires us to use certain complex routines involving dynamic maintenance of connected components to avoid taking time proportional to the square of the size of S.

We use algorithms such as [6] which perform dynamic maintenance of spanning trees to avoid taking $|S|^2$ time; the time complexity of these algorithms will be discussed in Theorem 4. Specifically, we maintain spanning trees of the subgraph of G' induced by $T_1 \cup S$ for each group T. Given these dynamically maintained spanning trees, we take O(f(n)+ $|S|$) time to divide the subgraph induced by $T_1 \cup S$ into connected components.

Given the components, we work in similar fashion to the splitting of T_1. We start by marking all neighbors of v in G', and all vertices in components of the subgraph induced by $T_1 \cup S$ which contain one of these vertices. We make a set of vertices reachable from v, initially consisting of the marked vertices of subgraph induced by $T_1 \cup S$, and mark all vertices of T_2 adjacent to a marked vertex of subgraph induced by $T_1 \cup S$.

We then step through the sets of T_2 from the first set in the ordering to the last. We divide the current set X into a subset of marked vertices, and a subset of unmarked vertices, placing the set of unmarked vertices before the set of marked vertices. We add the marked set to the reachable vertices, and make sure that each neighbor of these vertices becomes either marked (if it is in a later set) or added to the reachable set, as in the splitting of T_1 except that we add an entire connected component to the reachable set when a vertex of $T_1 \cup S$ is marked. The total running time of this algorithm is O(f(n)(f(n)+$|S|$)). Therefore, the total time to split a single group when v is chosen is O($f^2(n)$ + f(n)$|S|$).

Since there are O($n/f(n)$) groups, the total time taken to refine all sets when v is chosen is O(n/f(n)*($f^2(n)$)+f(n)($|S_1| + |S_2| + .. + |S_k|$), where S_i is the largest set in group T_i, and is thus O($nf(n)$) in total. □

Theorem 4 The minimal fill found by algorithm LEX-M of a graph can be computed in O($n^{2.69}$) time.

Proof: We first address the time needed for dynamic maintenance of spanning trees, which was deferred earlier. We add edges between a pair of vertices (v, w) only once, when the edge is first added to G'. We delete an edge only once (either when the pair is split into different groups, or when one of the vertices is placed in T_2), since when a new group is created the set $T_1 \cup S$ is a subset of a previous set for which a spanning tree was maintained. Therefore, there are at most n^2 calls to add/delete edges in the spanning tree. There are dynamic spanning tree algorithms which take polylogarithmic time per operation [6], so the overall time spent maintaining spanning trees is clearly o($n^{2.69}$).

Combining theorems 1, 3, and lemma 1 above, the total time taken is O(n^2f(n)+$n^{1+\alpha}/f(n)$). If we use the O($n^{2.376}$) matrix multiplication bound of [3], this balances using $f^2(n) = n^{1.376}$, i.e. f(n) = $n^{.688}$ and the time is O($n^{2.688}$). If we use Strassen's algorithm [12], generally considered more practical than [3] due to the large constants in the latter algorithm, and assume that matrix multiplication takes O($n^{2.81}$) time, the equation balances at $f^2(n) = n^{1.81}$ or f(n) = $n^{.905}$ i.e. the time is O($n^{2.91}$).

More generally, if we use O(n^k) time for matrix multiplication, the time bound for computing LEX-M in this way becomes O($n^{(3+k)/2}$). $\square$

Corollary 1: The clique cutset decomposition of a graph can be computed in O($n^{2.69}$) time.

Proof: At the end of the algorithm above, G' will contain exactly those edges which are either in G, or added to G as fill caused by the ordering produced by LEX-M. For each vertex v of the ordering, let C_v be the neighbors of v in G' which are after v in the ordering. [13] shows that a clique separator decomposition of G can be found by identifying which sets C_v are cliques in G.

Create a graph H by adding two vertices v_1, v_2 to G for each vertex v of G, making v_1 and v_2 adjacent to all vertices in C_v. It is not hard to see that C_v is a clique in G if and only if the number of paths of length 3 from v_1 to v_2 in H is equal to $|C_v|(|C_v|$-1). This information is easily obtained from the cube of the adjacency matrix of H, and thus we can determine which sets C_v are cliques of G using two extra matrix multiplications. $\square$

4 Conclusions and Open Problems

We have shown that a minimal fill ordering of a graph can be found in O($n^{2.69}$) time; this ordering can also be used to find a clique cutset decomposition in the same time bound.

The paper [2] shows that although LEX-M creates a minimal fill, the number of fill edges added tends to be larger than those for some other minimal fill algorithms. It would be interesting to see whether the other algorithms could also use matrix multiplication to reduce their time complexities on dense graphs.

We note in this context that independently from us, the recent paper [5] has shown that using a completely new algorithm, a minimal fill can be computed in O(n^αlog$n$) time.

It would also be interesting to show that computing a minimal fill of a graph is at least as hard as some other well-known problem (such as determing whether a graph has a triangle) for which the best known algorithms require matrix multiplication; for examples of this type of bound, see [7].

We would like to thank an anonymous referee for a careful reading which corrected errors in an earlier draft of this paper.

References

[1] A. Berry, A Wide-Range Efficient Algorithm for Minimal Triangulation, 10th Symposium on Discrete Algorithms, 860-861, 1999

[2] A. Berry, J.-P Bordat, P. Heggernes, G. Simonet, Y. Villanger, A Wide-Range Algorithm for Minimal Triangulation from an Arbitrary Ordering, Research Report LIMOS/RR-03-02, Université Clermont-Ferrand II

[3] D. Coppersmith, S. Winograd, Matrix Multiplication via Arithmetic Progressions, Journal of Symbolic Computing 9, 1-6, 1990

[4] M.R. Garey, D.S. Johnson, Computers and Intractability - A Guide to the Theory and Practice of NP-Completeness, W.H. Freeman, San Francisco, 1979

[5] P. Heggernes, J.A. Telle, Y. Villanger, Computing Minimal Triangulations in $\mathrm{O}(n^{\alpha}\log n) = \mathrm{o}(n^{2.376})$, SODA 2005, to appear

[6] J. Holm, K. de Lichtenberg, M. Thorup, Poly-logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-edge, and Biconnectivity, STOC 98, 79-89

[7] D. Kratsch, J. Spinrad, Between $\mathrm{O}(nm)$ and $\mathrm{O}(n^{\alpha})$, 14th Annual ACM-SIAM Symposium on Discrete Algorithms, 709-716, 2003

[8] T. Ohtsuki, A Fast Algorithm for Finding an Optimal Ordering for Vertex Elimination on a Graph, SIAM Journal on Computing 5, 133-145, 1976

[9] S. Parter, The use of Linear Graphs in Gauss Elimination, SIAM Review 3, 1190139, 1961

[10] D.J. Rose, R.E. Tarjan, Algorithmic Aspects of Vertex Elimination on Directed Graphs, SIAM Journal on Applied Mathematics 34, 176-197, 1978

[11] D. Rose, R.E. Tarjan, G. Lueker, Algorithmic Aspects of Vertex Elimination on Graphs, SIAM Journal on Computing 5, 266-283, 1976

[12] V. Strassen, Gaussian Elimination is not Optimal, Numerische Mathematik 14, 354-356, 1969

[13] R.E. Tarjan, Decomposition by Clique Separators, Discrete Mathematics 55, 221-232, 1985

[14] R.E. Tarjan, M. Yannakakis, Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs, SIAM Journal on Computing 13, 566-579, 1984

[15] M. Yannakakis, Computing the Minimum Fill-in is NP-complete, SIAM Journal on Algebraic Discrete Methods 3, 351-358, 1982