A Linear Time Implementation of the Reverse Cuthill-McKee Algorithm

by

W.M. Chan and Alan George
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-79-13

April 1979

Abstract

The Reverse Cuthill-McKee (RCM) algorithm is a method for reordering a sparse matrix so that it has a small envelope. Given a starting node, we provide an implementation of the algorithm whose run-time complexity is proved to be linear in the number of nonzeros in the matrix. Numerical experiments are provided which compare the performance of the new implementation to a good conventional implementation.

1. Introduction

Consider the system of linear equations

$$Ax = b \quad ,$$

where the N by N matrix A is sparse, symmetric and positive definite. If Cholesky's method is to be used to solve the system, the essential first step is to reorder the system; that is, to find a permutation matrix P with the aim of solving the equivalent system

$$(PAP^T)(Px) = Pb \quad ,$$

where P is chosen so that the Cholesky factor L of PAP^T has desirable properties. A common choice for P is one for which PAP^T has a small envelope [2], and one of the most widely used and effective algorithms for finding such a permutation is the Reverse Cuthill-McKee (RCM) algorithm, which we describe in Section 2.

In this paper we describe an implementation of the RCM algorithm which can be proven to execute in $O(|A|)$ time, where $|A|$ denotes the number of nonzeros in A. Since the number of inputs to the algorithm is $O(|A|)$, and the input must be read, it follows that our implementation is asymptotically optimal. Moreover, we provide numerical experiments comparing our implementation to a good existing implementation for which no such execution time bound can be provided. The experiments indicate that our new implementation is very competitive or superior.

2. Description of the RCM Algorithm

The algorithm operates on the unlabelled graph of the matrix A, so we begin by introducing a few graph theory notions. For our purposes, a graph $G = (X, E)$ consists of a finite nonempty set X of nodes together with an edge set E consisting of ordered pairs of nodes. The labelled graph of A is a graph having N nodes, labelled from 1 to N, with an edge set E consisting of edges such that $\{x_i, x_j\} \in E$ if and only if $a_{ij} = a_{ji} \neq 0$. The unlabelled graph of A is simply the graph obtained from A above with its labels removed. The RCM algorithm generates a labelling for this graph, and hence a (symmetric) reordering of A. We assume that the matrix A is irreducible, which implies that the graph G obtained from A is connected [1]. If not, the algorithm can be applied to each connected component separately.

The application of the RCM algorithm requires that a starting node be provided. There are a number of good (heuristic) algorithms for finding such nodes [4, 6], and we do not consider that problem here.

Given a starting node r, the algorithm is as follows.

Step 1. Set $x_1 \leftarrow r$.

Step 2. (Main loop) For $i = 1,2,\ldots,N$, find all the unnumbered neighbors of x_i and number them in increasing order of degree.

Step 3. (Reverse the ordering) The RCM ordering is given by $y_1, y_2, \ldots, y_N$, where $y_i = x_{N+1-i}$, $i = 1,2,\ldots,N$.

It is obvious that the complexity of the algorithm depends essentially on the way the sorting is done in Step 2. In the implementations known to the authors, a simple sorting algorithm is used, such as linear insertion. If α_i nodes are numbered during the i-th stage of Step 2, the time required is $O(\alpha_i^2)$. Under these circumstances Liu [6] showed that the execution time of the algorithm is $O(\alpha|E|)$, where $\alpha = \max\{\alpha_i \mid 1 \leq i \leq N\}$.

For the majority of sparse matrix problems, such an implementation is entirely satisfactory because the degrees of the nodes are small; although the sorting algorithm used is theoretically inefficient, because of its simplicity and the fact that the number of elements being sorted is small, in practise more efficient algorithms such as quick sort or merge sort are not as fast. However, occasionally one encounters problems where several nodes have very high degrees (a significant fraction of N), and in these cases the conventional implementations are quite slow; that is, the $O(N^2)$ bound for these problems manifests itself. We are proposing our implementation because it executes as fast as the conventional implementations, but does not degrade for problems which have nodes of high degree.

3. A Data Dependent Sorting Algorithm

In this section we develop a specialized sorting algorithm whose execution time is dependent on the data being sorted. Suppose we wish to sort N positive integers $p_1, p_2, \ldots, p_N$. For our purposes in this paper, we require an algorithm for sorting these numbers which has an execution time of $O(\sum_{k=1}^{N} p_k)$. While this may not in general be very impressive for a sorting algorithm, in the context of our application it is precisely what is required to achieve an optimal implementation of the RCM algorithm.

Since our sorting algorithm is basically a simple modification of the standard linear insertion sorting algorithm, we describe this standard algorithm first. The version shown below works from the "bottom up" to sort the N elements of the array P in increasing order.

```
Step 1.   $k \leftarrow N$
Step 2.   $k \leftarrow k-1$ ; If $k = 0$ , stop.
Step 3.   If $P(k) \le P(k+1)$ go to Step 2.
Step 4.   $TEMP \leftarrow P(k)$ ; $\ell \leftarrow k+1$
Step 5.   $P(\ell-1) \leftarrow P(\ell)$ ; $\ell \leftarrow \ell+1$
Step 6.   If $\ell > N$ or $P(\ell) \ge TEMP$
          then {$P(\ell-1) \leftarrow TEMP$ ; go to Step 2}
          else go to Step 5.
```

Now our objective is to produce an algorithm for sorting P whose execution time is $O(\sum_{k=1}^{N} P(k))$. It is clear that if P contains only distinct positive integers, this bound is immediate for the algorithm above. Moreover, the bound is sharp, since it is achieved when P contains the first N positive integers in descending order.

However, assume N is even, and consider the case when P contains 2 in the first $N/2$ positions, and 1 in the last $N/2$ positions. Then $\sum_{k=1}^{N} P(k)$ is $O(N)$, but the execution time is clearly $O(N^2)$, since a string of $N/2$ consecutive ones will be individually shifted up one position a total of $N/2$ times. The basic idea in our new algorithm is to arrange that when such replications of integers occur in the partially sorted list, these strings are effectively moved up one position (when required) by removing the bottom member of the string and adding one to the top of the string. We do this by recording information about the lengths of such strings in an array B, parallel to the array P.

In our description of the algorithm below, the steps correspond closely to those in the ordinary linear insertion algorithm described above, with changes added to maintain the state of the array B and to exploit the information contained in it.

```
Step 1.  $k \leftarrow N$ ;  $B(N) = 0$
Step 2.  $k \leftarrow k-1$ ;  If  $k = 0$ , stop
Step 3.  If  $P(k) < P(k+1)$
         then  {$B(k) \leftarrow 0$ ; go to Step 2}
         else  if  $P(k) = P(k+1)$
               then  {$B(k) \leftarrow B(k+1) + 1$ ; go to Step 2}
Step 4.  TEMP $\leftarrow P(k)$ ;  $\ell \leftarrow k+1$
Step 5.  $P(\ell-1) \leftarrow P(\ell + B(\ell))$ ;  $B(\ell-1) \leftarrow B(\ell)$
```

```
          $\ell \leftarrow \ell + B(\ell) + 1$
Step 6.   If  $\ell > N$  or  $P(\ell) \geq TEMP$
          then  {$P(\ell-1) \leftarrow TEMP$ ;
                 if  $P(\ell) = TEMP$
                      then   $B(\ell-1) \leftarrow B(\ell)+1$
                      else   $B(\ell-1) \leftarrow 0$    ;
                 go to Step 2}
          else  go to Step 5.
```

It should be apparent from the algorithm that if $P(m) = P(m+1) = \ldots = P(m+r)$ in the partially sorted list, then $B(m)$ has the value r . Then the string can be "moved up" one position by setting $P(m-1)$ to $P(m+r)$. Figure 1 illustrates the algorithm applied to an array P containing 8 arrays and the value of k , ℓ and TEMP <u>after</u> Step 2 has been executed, but <u>before</u> Step 3 has begun. The notation □ indicates that no assignment has been made to the variable.

	P	B
	3	□
	1	□
	2	□
	2	□
	1	□
	3	□
k →	2	□
	2	0

ℓ : □
TEMP : □
(i)

	P	B
	3	□
	1	□
	2	□
	2	□
	1	□
k →	3	□
	2	1
	2	0

ℓ : □
TEMP : □
(ii)

	P	B
	3	□
	1	□
	2	□
	2	□
k →	1	□
	2	1
	2	1
	3	0

ℓ : 9
TEMP : 3
(iii)

	P	B
	3	□
	1	□
	2	□
k →	2	□
	1	□
	2	1
	2	1
	3	0

ℓ : 9
TEMP : 3
(iv)

	P	B
	3	□
	1	□
k →	2	□
	1	0
	2	2
	2	1
	2	1
	3	0

ℓ : 6
TEMP : 2
(v)

	P	B
	3	□
k →	1	□
	1	0
	2	3
	2	2
	2	1
	2	1
	3	0

ℓ : 5
TEMP : 2
(vi)

	P	B
k →	3	□
	1	□
	1	0
	2	3
	2	2
	2	1
	2	1
	3	0

ℓ : 5
TEMP : 2
(vii)

	P	B
k →		
	1	1
	1	1
	2	3
	2	3
	2	2
	2	1
	3	1
	3	0

ℓ : 8
TEMP : 3
(viii)

Figure 1 States of the arrays (after execution of Step 2) sorting process.

4. Complexity of the RCM Algorithm

Lemma 1 Let P be an array of N positive integers. Then the time complexity to sort P in order using the modified linear insertion sorting algorithm described above is $O(\sum_{k=1}^{N} \min(k, P(k)))$.

Proof: Clearly Step 1 is executed only once, and Steps 2, 3 and 4 are executed exactly N times. Thus, the proof hinges on how many times Steps 5 and 6 are executed.

Now observe that for each $P(k)$, there are at most $P(k)$ distinct positive integers not greater than $P(k)$. Thus, Steps 5 and 6 will be executed at most $\min(k, P(k))$ times for each k , which proves the lemma.

$\square$

Remark. If the entries of P are non-negative integers instead, the sorting algorithm can be proved to be $O(\sum_{k=1}^{N} P(k) + N)$ by a similar method.

Lemma 2 In the RCM algorithm, the time required to sort the unnumbered neighbours $x_1, x_2, \ldots, x_\lambda$ of y is $O(\sum_{i=1}^{\lambda} |\mathrm{Adj}(x_i)|)$.

The proof is a direct consequence of Lemma 1. Here $\mathrm{Adj}(x_i)$ is the set of nodes adjacent to x_i in the graph, so $|\mathrm{Adj}(x_i)|$ is the degree of node x_i . Since x_i is a neighbour of at least one node y , $|\mathrm{Adj}(x_i)|$ is positive.

<u>Theorem 3</u> The time complexity of the RCM algorithm is $O(|E|)$.

<u>Proof</u>: Let the number of nodes in the graph be N . The starting node will be labelled as node 1. The unlabelled neighbours of each labelled node are numbered by the RCM algorithm described in Section 3 successively until all labelled nodes are exhausted.

In other words, except the starting node, each node in the graph is numbered exactly once as a neighbour of node i , for some i , $1 \le i \le N-1$. Thus, excluding the starting node r , we can partition the node set X according to when they were numbered

$$X = \sum_{i=1}^{N-1} P_i \cup \{r\} .$$

For each P_i , Step 2 of the algorithm in Section 2 is applied once, and the time required to number P_i is

$$O(\sum_{x \in P_i} |Adj(x)|) .$$

Thus, the time complexity for the numbering process for all the vertices is

$$O(\sum_{i=1}^{N-1} \sum_{x \in P_i} |Adj(x)|) .$$

But $\sum_{i=1}^{N-1} \sum_{x \in P_i} |Adj(x)| \le \sum_{x \in X} |Adj(x)| = 2|E|$ which proves the theorem.

□

5. Numerical Experiments and Conclusions

In this section we provide a few numerical experiments to illustrate the points made in Sections 1 and 2. In particular, we wish to demonstrate that our new implementation is as efficient as convention implementations; that is, we want to show that the gauranteed behaviour of the algorithm does not penalize its execution time.

Table 1 contains ordering times for two implementations of the RCM algorithm, applied to a sequence of L-shaped graph problems taken from [5]. These problems are sparse, and all node degrees are bounded by 7. The two implementations differ only in the way the sorting is performed in Step 2 of the RCM algorithm; the "original" implementation [3] uses linear insertion, while the "modified" algorithm uses the new sorting scheme described in Section 3.

The execution times shown in Table 1 are quite satisfactory in our opinion. Since the degree of the nodes for these problems are all small, we simply want to confirm that our new scheme is competitive with the conventional approach.

N	Original Implementation	Modified Implementation
265	.12	.11
406	.19	.17
577	.27	.24
778	.36	.33
1009	.49	.42
1270	.60	.53
1561	.73	.66
1882	.88	.80
2233	1.04	.95

Table 1 Ordering times for the original and modified implementations of the RCM algorithms, applied to a sequence of problems from [5].

6. References

[1] C. Berge, The Theory of Graphs and Its Application, John Wiley and Sons, Inc., New York, 1962.

[2] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices", Proc. 24th Nat. Conf., ACM Publ. p. 69, 1122 Ave. of the Americas, New York, N. Y. 1969.

[3] A. George and J.W.H. Liu, Computer Solution of Large Sparse Positive Definite Systems, to be published by Prentice Hall, Inc.

[4] A. George and J.W.H. Liu, "An efficient implementation of a pseudo-peripheral node finder", ACM Trans. on Math. Software, (to appear).

[5] A. George and J.W.H. Liu, "Algorithms for matrix partitioning and the numerical solution of finite element systems", SIAM J. Numer. Anal., 15 (1978), pp. 297-327.

[6] N.E. Gibbs, W.G. Poole, and P.K. Stockmeyer, "An algorithm for reducing the bandwidth and profile of a sparse matrix", SIAM J. Numer. Anal., 13 (1976), pp. 236-250.

[7] J.W.H. Liu, "On reducing the profile of sparse symmetric matrices", Report CS-76-07, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada (February 1976).