Solving Sparse Symmetric Path Problems on a Network of Computers

Robert Manger and Goranka Nogo

Department of Mathematics, University of Zagreb, Croatia

We present an optimized version of a previously studied distributed algorithm for solving path problems in graphs. The new version is designed for sparse symmetric path problems, i.e. for graphs that are both sparse and undirected. We report on experiments where the new version has been implemented and evaluated with the PVM package.

Keywords: graph theory, path problems, distributed computing, parallel virtual machine (PVM), experiments, symmetric path problems, sparse graphs.

1. Introduction

Path problems [10] are a family of optimization and enumeration problems, which reduce to generation or comparison of paths in directed or undirected graphs. Some examples are: checking path existence, finding shortest or most reliable paths, listing all paths, etc.

Path problems can be solved in many ways, by sequential, parallel, general or specialized algorithms [2, 4, 6]. In [8] we have proposed a fairly general *distributed* algorithm which can be implemented on a network of computers and can be applied to all types of path problems. In [7] we have also studied optimized versions of the same algorithm that are faster but applicable only to certain classes of problems, e.g. symmetric or acyclic or sparse.

The aim of this paper is to present and evaluate yet another optimized version, which is faster but even more restricted in its applicability. Namely, the new version is designed for sparse symmetric path problems, i.e. those whose graphs are both sparse and undirected.

The paper is organized in the following way. Sections 2 and 3 give preliminaries about path problems and the general distributed algorithm. Section 4 introduces our new optimized version. Section 5 reports on experiments, where the new version has been implemented using the PVM package and tested on appropriate randomly generated problem instances. The final Section 6 gives conclusions.

2. Path problems

We use the algebraic approach to path problems developed in [1]. According to this approach, each particular problem in a (directed) graph with n nodes is represented by a suitable $n \times n$ *adjacency* matrix A. The entries of A are not necessarily numbers, but elements of a special set P, which is called a *path algebra*. The elements of P are manipulated by two binary operations, $\vee$ (join) and $\circ$ (multiplication), which are analogous to conventional addition and multiplication, respectively. The two operations satisfy the following properties: $\vee$ is idempotent, commutative, and associative; $\circ$ is associative, left-distributive, and right-distributive over $\vee$; there exist a zero element ϕ and a unit element ϵ such that (for any a) $\phi \vee a = a$, $\phi \circ a = \phi = a \circ \phi$, $\epsilon \circ a = a = a \circ \epsilon$.

The solution of the path problem given by A is obtained by computing the so-called *closure* matrix:

$$A^* = E \vee A \vee A^2 \vee A^3 \vee \dots . \quad (1)$$

Here, E denotes the unit matrix (ϵ on the diagonal, ϕ elsewhere). The matrix operations $\vee$ and

$\circ$ are derived from the corresponding scalar operations, similarly as in ordinary linear algebra. In all meaningful path problems the matrix A is *stable*, i.e. the closure A^* can be computed in a finite number of algebraic operations.

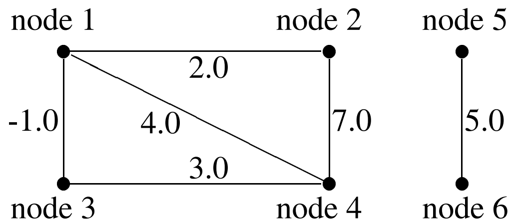

Fig. 1. A sample graph.

As an illustration, let us consider the graph in Figure 1 whose arcs are given "lengths". Suppose that we want to solve the *shortest distance* problem, i.e. we want to determine the length of a shortest path between any pair of nodes. Then the matrix A and its closure A^* are:

$$A = \begin{bmatrix} \infty & 2.0 & -1.0 & 4.0 & \infty & \infty \\ 2.0 & \infty & \infty & 7.0 & \infty & \infty \\ -1.0 & \infty & \infty & 3.0 & \infty & \infty \\ 4.0 & 7.0 & 3.0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 5.0 \\ \infty & \infty & \infty & \infty & 5.0 & \infty \end{bmatrix},$$

$$A^* = \begin{bmatrix} 0.0 & 2.0 & -1.0 & 2.0 & \infty & \infty \\ 2.0 & 0.0 & 1.0 & 4.0 & \infty & \infty \\ -1.0 & 1.0 & 0.0 & 3.0 & \infty & \infty \\ 2.0 & 4.0 & 3.0 & 0.0 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0.0 & 5.0 \\ \infty & \infty & \infty & \infty & 5.0 & 0.0 \end{bmatrix}.$$

P is here the set of real numbers including ∞, with the standard min as $\vee$, and the standard $+$ as $\circ$. The zero element in P is ∞, and the unit element is 0. The (i, j)-th entry of A is the length of the arc from node i to node j, while the (i, j)-th entry of A^* is the length of the shortest path from node i to node j.

As the second example, let us consider once again the same graph in Figure 1, but suppose now that we want to solve the *path existence* problem, i.e. we only want to determine which pairs of nodes are connected by a path and which are not. Then we can use a much simpler Boolean path algebra P, which consists of the values 0 and 1, with the operations max and min as $\vee$ and $\circ$, respectively. The corresponding matrices A and A^* have the same zero/non-zero structure as before: any ∞ is now replaced with 0 and any other value with 1. The (i, j)-th entry of A specifies the existence of an arc from node i to node j, while the (i, j)-th entry of A^* indicates the existence of a path from node i to node j.

As shown in [1, 10], many other types of path problems can be formulated and solved in a similar fashion. The general structure of the problem always stays the same. However, the set P changes, and the operations $\vee$ and $\circ$ can have different meanings.

Note that the two examples shown above are quite specific, namely their matrices $A, A^2, \ldots, A^*$ happen to be symmetric (which is generally not the case). We deal here with *symmetric* path problems. It is easy to prove that symmetric problems are exactly those where the involved graph is undirected and the operation $\circ$ is commutative.

3. Distributed algorithm

Our algorithm for solving path problems is in fact an algorithm that computes the closure A^* of a given stable $n \times n$ matrix A over an arbitrary path algebra P. The expression of the form (1) is evaluated relatively quickly, by *iterative squaring and updating* of a suitable matrix B. The algorithm starts with $B = E \vee A$, and stops when B stabilizes.

Our algorithm is distributed in the sense that it consists of a ring of m concurrent processes, where $1 \leq m \leq n/2$. Each process communicates with its predecessor and its successor along the ring. There is no shared memory, but each process has its own private memory. Figure 2 refers to a ring of $m = 4$ processes.

To enable distributed computing, the algorithm maintains two copies of the matrix B, denoted by $R = [r_{ij}]$ and $C = [c_{ij}]$. The matrix R is divided into $2m$ blocks, so that every block consists of a roughly equal number of adjacent rows. Similarly, C is divided into $2m$ blocks of columns. The range of column indices assigned to one particular block of C is the same as the range of row indices assigned to the *corresponding* block of R. The blocks of R and C are distributed among processes, so that one process

keeps exactly two blocks of R and the corresponding two blocks of C.

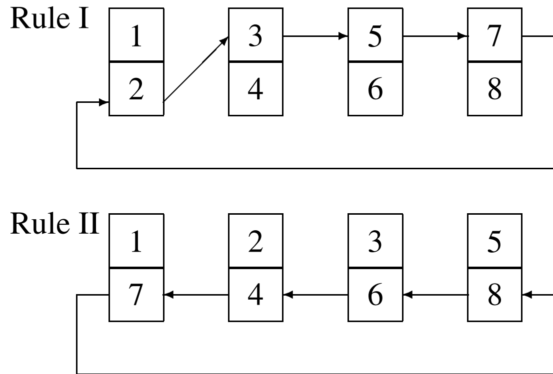

Fig. 2. Block exchange rules for $m = 4$.

As previously explained, the algorithm consists of iterations. However, due to distributed computing, each iteration is further divided into $2m-1$ smaller parts called *phases*. In one phase a process generates row-column pairs from its locally available blocks. In each phase all possible pairs from non-corresponding blocks are generated. In the first phase within an iteration, additionally, all possible pairs from the corresponding blocks are also generated. For each generated pair, consisting of, say, the i-th row of R and the j-th column of C, the process computes the "inner product". The obtained value is used to update both the j-th element of the i-th row of R and the i-th element of the j-th column of C; thus:

$$r_{ij} := c_{ij} := \bigvee_{l=1}^{n} r_{il} \circ c_{lj}. \qquad (2)$$

After any phase the processes exchange blocks in order to form new combinations of blocks for the next phase. Any block of C moves together with the corresponding block of R. The exact block exchange procedure is shown in Figure 2, for the case where $m = 4$. Rules I and II are alternately applied. Each index in Figure 2 denotes one block of R together with the corresponding block of C. Arrows in Figure 2 indicate block moves. As proved in [3], the chosen procedure assures that through a single iteration each block of R meets each of the non-corresponding blocks of C exactly once.

It has been proved in [8] that for a stable matrix A the algorithm terminates after a finite number of iterations, with the final matrices R and C equal to A^*.

4. Optimized version

Optimization of the described distributed algorithm is based on the idea that the used internal data structures should be made more compact. Since instances of those data structures are exchanged among processes, their smaller size implies smaller communication costs. Another positive side effect of data compaction is that some unnecessary algebraic operations are automatically skipped. The price we pay for this improvement of performance is a loss of generality.

In [7] we have already designed a version for dense symmetric path problems. That version simply stores only one copy of the matrix B, for instance C, and skips R. Thanks to symmetry, each row of R, whenever needed, can be replaced by the corresponding column of C. The blocks of C are represented as ordinary arrays.

In [7] we have also designed a version suitable for sparse unsymmetric problems. The version uses again two matrices R and C as described in the original algorithm. However, the blocks of R and C are represented by a special data structure that exploits sparsity and explicitly stores only non-zero entries.

In this paper we are proposing a new optimized version, which combines the features of the two versions mentioned above. Thus the new version stores only one copy of B, for instance C, and represents the blocks of C by the special data structure. Obviously, the new version works correctly only for a symmetric problem, and becomes really efficient if that problem is at the same time sparse.

The data structure used in the new version is based on *linked lists* [6, 9]. One linked list corresponds to one column of the block of C. Each record within a list contains the value and the row index of a non-zero entry from that column. The pointer within a record points to the next non-zero entry from the same column.

Figure 3 shows a possible instance of the used data structure, provided that the considered block

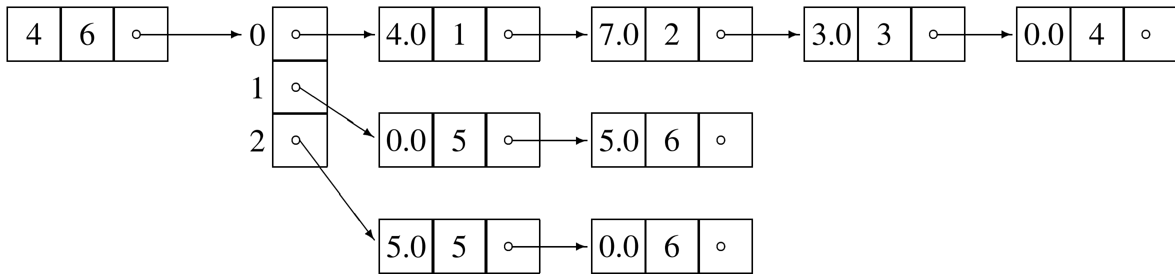

Fig. 3. Linked lists representation of a block of C.

of C consists of the 4-th to 6-th column of B, and B corresponds to the graph in Figure 1. The starting record also includes the original range of column indices for columns within the block. Note that ϕ in our example is ∞, not 0.0. In order to assure portability of data structures among processes, an actual implementaton could use a tabular variant of linked lists, i.e. record components could be put into arrays, and pointers replaced by array indices.

Computation in the new optimized version is done essentially in the same way as in the original algorithm, provided that rows of (nonexistent) R are replaced by corresponding columns of C. More precisely, an inner product (2) is now computed according to the following formula:

$$c_{ji} := c_{ij} := \bigvee_{l=1}^{n} c_{li} \circ c_{lj}. \qquad (3)$$

The formula (3) is evaluated only for $i \leq j$ and not for all (i, j) pairs. Note also that, due to the used data structure, (3) implies a more complicated algorithm than for ordinary arrays. Namely, computing an inner product means traversing two linked lists simultaneously, and updating an entry means finding the appropriate record within a linked list or inserting a new record. Since zero elements are not explicitly stored, trivial algebraic operations are automatically avoided thus additionally reducing computational effort.

5. Experiments

The original algorithm, its two previously mentioned older optimized versions, and the considered new optimized version have all been implemented as separate C programs. The program code has been designed so that it can easily be adjusted to solve various types of path problems (path existence, shortest distances, etc). In order to monitor performance, the programs have been extended so that they count their own iterations and record their own execution time. By execution time we mean the real physical time, which comprises computation, communication, and possible network delays.

Each program has been built by using the *PVM package* [5] as a set of concurrent processes, which can be allocated to different computers, and which exchange data through the network in a ring-like fashion. The programs have been tested on the same virtual parallel machine, assembled of four UNIX computers, as already described in [7, 8]. We have been able to run any program with up to eight truly concurrent processes.

The performance of the new version has been measured on 40 randomly generated path existence problems and 40 randomly generated shortest distance problems, all with undirected and more or less sparse graphs. In order to get relative measures of performance, all 80 problems have also been solved by the original algorithm and by the two older versions. The obtained results are summarized in Tables 1 and 2.

Each of the Tables 1 and 2 shows the results for the corresponding 40 problems of a certain type (path existence or shortest distances). The problems within the same table have further been divided into groups of 10 according to their size (n = 128 or 256) and their graph density (percentage of non-zeros in A - 1% or

prob-lem size n	graph den-sity	# of proce-sses m	# of itera-tions	exec-ution time	speedup vs orig algorithm	speedup vs D-S version	speedup vs S-U version	speedup vs sequen execution
128	1%	1	3.8	0.7	15.00	6.86	1.71	1.00
		2	4.0	0.6	11.17	6.83	1.50	1.17
		4	3.8	0.5	7.60	5.60	1.40	1.40
		8	3.8	2.0	2.75	1.95	1.35	0.35
	10%	1	3.0	5.7	1.44	0.68	2.09	1.00
		2	3.0	4.3	1.12	0.70	1.58	1.33
		4	3.0	2.6	1.08	0.81	1.42	2.19
		8	3.0	3.1	1.39	0.97	1.42	1.84
256	1%	1	3.4	31.4	2.56	1.17	2.19	1.00
		2	3.1	18.7	2.02	1.08	1.86	1.68
		4	3.2	11.8	1.78	1.11	1.60	2.66
		8	3.1	8.3	1.39	1.13	1.23	3.78
	10%	1	2.9	52.5	1.29	0.60	2.16	1.00
		2	3.0	33.4	1.08	0.59	1.85	1.57
		4	3.0	20.8	0.94	0.59	1.60	2.52
		8	3.0	18.7	0.59	0.49	1.21	2.81

Table 1. Experimental evaluation of the new version, path existence problems.

prob-lem size n	graph den-sity	# of proce-sses m	# of itera-tions	exec-ution time	speedup vs orig algorithm	speedup vs D-S version	speedup vs S-U version	speedup vs sequen execution
128	1%	1	3.8	0.9	34.67	17.89	1.89	1.00
		2	4.0	0.7	25.71	14.14	1.86	1.29
		4	3.9	0.6	17.83	10.50	1.67	1.50
		8	3.9	2.7	2.81	2.26	1.22	0.33
	10%	1	4.0	23.2	1.78	0.91	1.96	1.00
		2	4.0	14.0	1.59	0.84	1.91	1.66
		4	4.0	9.1	1.36	0.81	1.71	2.55
		8	4.0	8.6	1.02	0.80	1.37	2.70
256	1%	1	4.0	116.3	2.72	1.32	2.07	1.00
		2	4.0	63.6	2.66	1.25	2.13	1.83
		4	4.0	33.9	2.64	1.26	2.12	3.43
		8	4.0	22.5	2.27	1.17	1.94	5.17
	10%	1	4.0	221.6	1.61	0.78	2.07	1.00
		2	4.0	113.4	1.63	0.78	2.09	1.95
		4	4.0	60.1	1.60	0.78	2.03	3.69
		8	4.0	36.0	1.56	0.76	2.05	6.16

Table 2. Experimental evaluation of the new version, shortest distance problems.

10%). Each particular problem has been solved with different numbers of processes ($m = 1, 2, 4$ or 8). Entries in Tables 1 and 2 are, in fact, average values computed over a whole group of 10 similar problems. A table row comprises the number of iterations and the total execution time for a given group and a given number of processes. The time is expressed as an absolute value (seconds), and as four relative values (speedups).

The "speedup vs original algorithm" shows how many times the considered new version is faster than the original algorithm running with the

same number of processes. The "speedup vs D-S version" compares in a similar way the new version and the older "dense-symmetric" version, while the "speedup vs S-U version" does an analogous comparison with respect to the "sparse-unsymmetric" version. Finally the "speedup vs sequential execution" compares the new version configured with more processes and the same version configured with only one process. Through different speedups we see different effects of optimization (data compaction) and of distributed computing.

6. Conclusions

The considered general distributed algorithm can be optimized for certain smaller classes of path problems. In a previous paper we studied versions designed for dense symmetric and sparse unsymmetric problems, respectively. In this paper we have introduced a version for sparse symmetric problems.

The experiments have shown that for a symmetric problem the best performance is always achieved by either of the two "symmetric" versions. If the involved graph is *very* sparse, the new "sparse-symmetric" version runs considerably faster than the older "dense-symmetric" version. On the other hand, if the graph is only *moderately* sparse, then the performance of the new version becomes similar, or even worse, compared to the older version. This is in fact not surprising since the solution of a moderately sparse problem can be dense.

References

[1] CARRÉ B., *Graphs and Networks*, Oxford: Oxford University Press; 1979.

[2] CRICHLOW J.M., *An Introduction to Distributed and Parallel Programming*, Englewood Cliffs NJ: Prentice-Hall; 1997.

[3] EBERLEIN P.J., PARK H., Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures, *Journal of Parallel and Distributed Computing* 1990; 8: 358–366.

[4] GAYRAUD T., AUTHIE G., A parallel algorithm for the all pairs shortest path problem, in: Evans D.J., Joubert G.R., Liddell H., editors, *Parallel Computing '91. Advances in Parallel Computing 4*. Amsterdam: North-Holland 1992. pp. 107–114.

[5] GEIST A., BEGUELIN A., DONGARRA J., JIANG W., MANCHEK R., SUNDERAM V., *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*, Cambridge MA: The MIT Press; 1994.

[6] HOROWITZ E., SAHNI S., RAJASEKARAN S., *Computer Algorithms / C++*, New York: Computer Science Press; 1997.

[7] MANGER R., NOGO G., Optimized versions of a distributed algorithm for solving path problems, *Central European Journal of Operations Research – CEJOR* 2000; 8: 109–123.

[8] MANGER R., Solving path problems on a network of computers, *Informatica* 2002; 26: 91–100.

[9] PREISS B.R., *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*, New York: John Wiley and Sons; 1999.

[10] ROTE G., Path problems in graphs, *Computing Supplement* 1990; 7: 155–189.

Received: June, 2003
Accepted: September, 2003

Contact address:
Robert Manger, Goranka Nogo
Department of Mathematics, University of Zagreb
Bijenička cesta 30, 10000 Zagreb, Croatia
e-mail: manger@math.hr, nogo@math.hr

ROBERT MANGER received the BSc (1979), MSc (1982), and PhD (1990) degrees in mathematics, all from the University of Zagreb. For more than ten years he worked in industry, where he obtained practical experience in programming, computing, and designing information systems. Dr Manger is presently an associate professor in the Department of Mathematics at the University of Zagreb. He has published over 30 scientific and 10 professional papers. His current research interests include: parallel and distributed algorithms, combinatorial optimization, soft computing and database systems. Dr Manger is a member of the Croatian Mathematical Society, Croatian Society for Operations Research and IEEE Computer Society.

GORANKA NOGO received the BSc (1981), MSc (1985), and PhD (1998) degrees in mathematics, all from the University of Zagreb. She has been with the Department of Mathematics at the University of Zagreb since 1983 where she is presently an assistant professor. During last years her work is centered around the theory of complexity, parallel algorithms and network flow problems. She has a practical experience in programming and computing. Dr Nogo is a member of the Croatian Mathematical Society and the Mathematica Reference Center.