A TOPOLOGICAL APPROACH TO DYNAMIC GRAPH CONNECTIVITY *

John H. REIF **

Aiken Computation Laboratory, Division of Applied Sciences, Harvard University, Cambridge, MA 02138, U.S.A.

A *dynamic connectivity problem* consists of an initial graph, and a sequence of operations consisting of graph modifications and graph connectivity tests. The *size* n of the problem is the sum of the maximum number of vertices and edges of the derived graph, plus the number of operations to be executed. Each graph modification is a deletion of either an edge or an isolated vertex. Each graph connectivity test is to determine if there exists a path in the current graph between two given vertices (the vertices can vary for distinct tests). The best previously known time for this dynamic connectivity problem was $\Omega(n^2)$.

Our main result is an $O(ng+n \log n)$ time algorithm for the dynamic connectivity problem in the case of the maximum genus of the derived graph being g.

Keywords: Graph algorithm, connectivity, dynamic algorithm, planar graph

1. Introduction

1.1. Previous graph connectivity algorithms

Tarjan [15] and Hopcroft and Tarjan [10] have given linear time depth-first search algorithms for graph connectivity and biconnectivity, and directed graph strong connectivity. Although these algorithms are asymptotically optimal for a given single input graph, they may be inefficient to apply in the case of dynamic connectivity problems where the graph undergoes repeated modifications, and we require connectivity tests on each of the derived graphs. Dynamic graph connectivity in the case of only edge additions can be solved in $O(n\alpha(n, n))$ time for any graph by use of the disjoint set union algorithm [16]. The algorithm of Frederickson [6] can be used to solve the dynamic graph connectivity problem for planar graphs with edge insertions and deletions in $O((\log n)^2)$ time per update. However, there was previously no efficient solution to the dynamic connectivity problem, in the case of general graphs, except to execute the known depth-first search algorithms on each of the derived graphs. This requires quadratic time.

1.2. Dynamic depth-first search

One approach to dynamic connectivity problems is to develop a dynamic algorithm for updating depth-first trees. However, Section 4 gives evidence that this cannot be done efficiently (i.e., in less than linear time per update in the worst case) since this would imply an efficient solution of a number of other complete dynamical problems.

* This work was supported by the National Science Foundation under Grant No. MCS-82-000269 and by the Office of Naval Research under Contract No. N00014-80-C-0647.

** Present affiliation: Department of Computer Science, Duke University, Durham, NC 27706, U.S.A.

Preprint of paper appearing in Information Processing Letters, Volume 25, Issue 1, 20 April 1987, Pages 65-70.

1.3. Our topological approach to dynamic connectivity

Our basic approach is to assume graph G to be embedded on an oriented surface. In this case, connectivity properties of G can be computed from topological properties of the embedding, such as the genus and the number of faces. Furthermore, modifications to the graph may result in changes in these topological properties, from which we can determine changes in the connectivity of G. We keep a special representation of the graph embedding and its dual which allows us to efficiently implement modifications to the graph embedding, and to efficiently update the connectivity properties of interest.

This paper is organized as follows: in Section 2 we define oriented graphs representing graph embeddings and describe efficient data structures for maintaining topological properties of these embeddings. Section 3 describes our dynamic graph connectivity algorithm with edge deletions. Finally, Section 4 considers complete dynamic problems.

2. Data structures for graph embeddings

Let $G = (V, E)$ be an undirected graph where V is a finite set of vertices and

$$E \subseteq \{\{u, v\} \mid u, v \in v\}.$$

An *embedding* of G onto an oriented surface can be specified [3] by giving for each vertex $v \in V$ a cyclic permutation $\theta(v)$ of the edges containing v. We call (G, θ) an *oriented graph*. Our data structure for (G, θ) will represent each cyclic permutation $\theta(v)$ by a doubly linked cyclic list, and each edge $\{u, v\} \in E$ will have pointers to the occurrence of $\{u, v\}$ in $\theta(v)$ and $\theta(u)$.

Let

$$\vec{E} = \{(u, v) \mid \{u, v\} \in E\} \cup \{(v, u) \mid \{u, v\} \in E\}$$

be the set of directed edges derived from $\vec{E}$. Let $\pi : E \to E$ be the permutation on $\vec{E}$ such that $\pi((u, v)) = (v, w)$ iff edge $\{v, u\}$ is immediately followed by edge $\{v, w\}$ in $\theta(v)$ (see Fig. 1). The

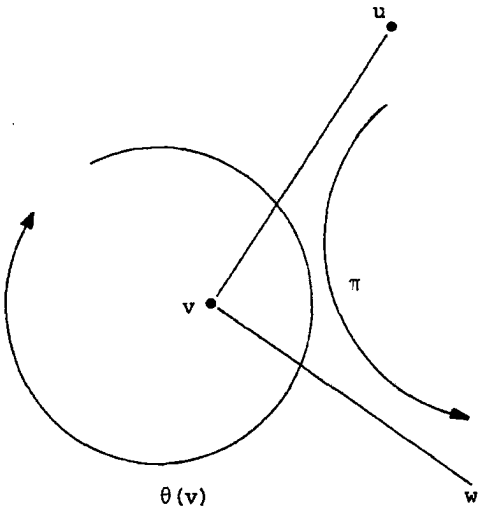

Fig. 1. The induced permutation π.

faces of the embedding are the orbits of π (see [17]).

Let F be the set of faces. Let c be the number of connected components of G. Euler's equation gives

$$|E| - |V| - |F| = 2(g - c),$$

where g is the genus of the embedding.

For each edge $\{u, v\} \in E$ we have a *dual edge*

$$D(\{u, v\}) = \{f, f'\}$$

consisting of the set of (not necessarily distinct) faces containing (u, v) or (v, u). We define the *dual graph*

$$D(G) = (F, D(E)),$$

$$\text{where } D(E) = \{D(\{u, v\}) \mid \{u, v\} \in E\}.$$

For each face $f \in F$, we define a cyclic permutation $D(\theta)(f)$ of the dual edges containing f, so that dual edge $D(\{v, u\}) = \{f, f'\}$ is immediately followed by $D(\{v, w\}) = \{f, f''\}$ iff directed edge (u, v) is immediately followed by (v, w) in f (see Fig. 2).

In addition to our data structure for oriented graph, (G, θ) will also maintain the dual oriented graph $(D(G), D(\theta))$ using a similar data structure. Furthermore, we keep links to and from each edge $e \in E$ and dual $D(e) \in D(E)$ (Guibas and Stolfi [7] also have described data structures for graph embeddings).

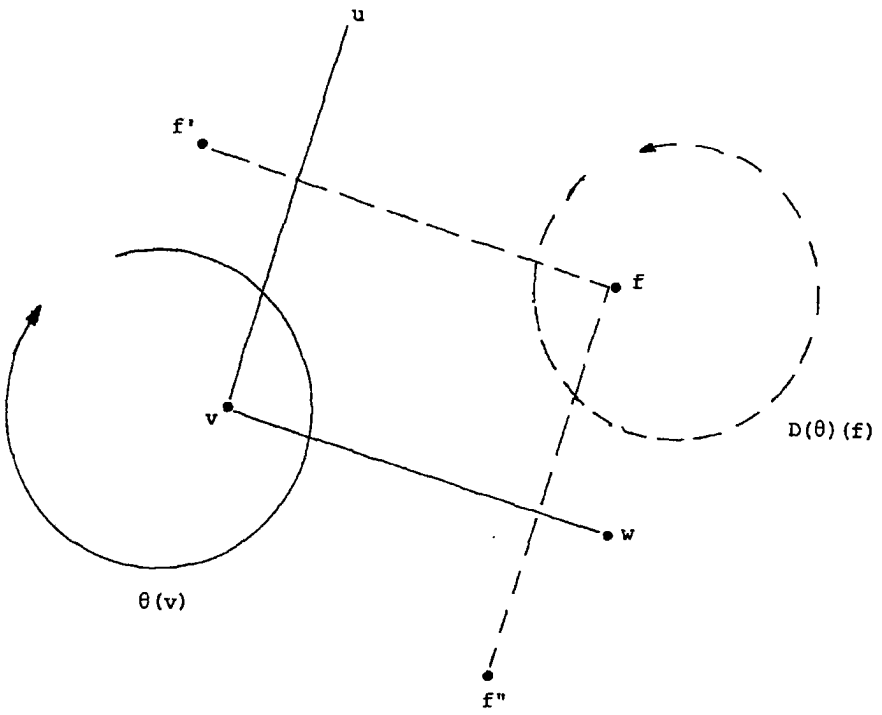

Fig. 2. Oriented graph (G, θ) is given by solid lines and oriented graph (G′, θ′) is given by dashed lines.

We shall also use a balanced binary tree to maintain a mapping *face*: $\vec{E} \to F$, so *face*((u, v)) is the unique face containing directed edge $(u, v) \in \vec{E}$.

3. An efficient algorithm for dynamic connectivity, with edge deletions

Given an initial oriented graph (G_0, θ_0) of genus g we wish to process a sequence of n operations where we may (i) delete an isolated vertex, (ii) delete an edge, or (iii) test if two given vertices are in the same connected component. Let n be the size of this problem, i.e., the sum of the number of edges and vertices of G_0 and the number of operations to be executed.

We shall begin by executing the O(n) time depth-first search algorithm of Tarjan [15] to determine the connected components of G_0. Also, in linear time we can construct the dual graph $D(G_0)$ and the mapping *face* as described in Section 2.

Suppose after processing some (but not all) of the required operations in order, we have the derived oriented graph (G, θ) of genus g, where G = (V, E). We assume that we have maintained the dual graph D(G) = (F, D(E)). We also assume to have maintained a connectivity mapping CC: $V \to V$ such that $\forall u,v \in V$, CC(u) = CC(v) iff u is in the same connected component of G as v and, furthermore, CC(v) is in the same connected component as v.

Given this information, we require constant time to delete an isolated vertex or to test if given vertices are in the same connected component.

Our key problem is to efficiently delete an edge {u, v}. In particular, we must appropriately determine the resulting oriented graph (G′, θ′) and its dual (D(G′, D(θ′)), and also update the connectivity mapping CC. By application of Euler's equation we have the following lemma.

3.1. Lemma. *If* D({u, v}) *consists of two distinct faces, then* (G′, θ′) *has the same genus as* (G, θ), *and the same connected components but one less face. If* D({u, v}) *consists of a single face and there is no path in* G *from* u *to* v *avoiding edge* {u, v}, *then* {G′, θ′) *has the same genus as* (G, θ) *but* (G′, θ′) *has an additional face and connected*

component. Otherwise, (G', θ') *has the same connected components as* G *but there is an additional face in* (G', θ') *and the genus of* (G', θ') *is one less than the genus of* (G, θ).

The oriented graph representation described in Section 2 allows us to efficiently perform deletion operations. Suppose $D(\{u, v\}) = \{f_1, f_2\}$ are distinct faces. We call this a *type* 1 *deletion*. In this case, we need not modify CC since u, v will remain in the same component of G'. However, we must merge f_1 into f_2 to form a new face f' such that if originally

$$D(\theta)(f_1) = (e_0 = \{u, v\}, e_1, \ldots, e_k, e_0, \ldots)$$

and

$$D(\theta)(f_2) = (e_0 = \{u, v\}, e'_1, \ldots, e'_k, e_0, \ldots),$$

then the resulting cyclic permutation is

$$\theta(v)(f') = (e_1, \ldots, e_k, e'_1, \ldots, e'_k \ldots)$$

(see Fig. 3). This is accomplished by first deleting e_0 from $D(\theta)(f_1)$ and $D(\theta)(f_2)$ and then performing an insertion operation on the corresponding doubly linked lists. We can update the mapping *face* appropriately using the union algorithm for balanced binary trees described in [1].

Next suppose $D(\{u, v\})$ consists of a single face f. We assume that for each vertex $w \in V$ there are Boolean variables $visit_1(w)$, $visit_2(w)$ which are initially *false*. We perform simultaneously a depth-first search DFS_1 starting at vertex u of the connected component containing u, and a depth-first search DFS_2 starting at vertex v of the connected component containing V. (We do not allow DFS_1 and DFS_2 to traverse edge $\{u, v\}$.) We alternately execute one step of DFS_1, and then one step of DFS_2. We immediately terminate both DFS_1 and DFS_2 if either terminates or if they visit a common vertex.

The case where DFS_1 and DFS_2 visit no common vertex is called a *type* 2 *deletion*. Without

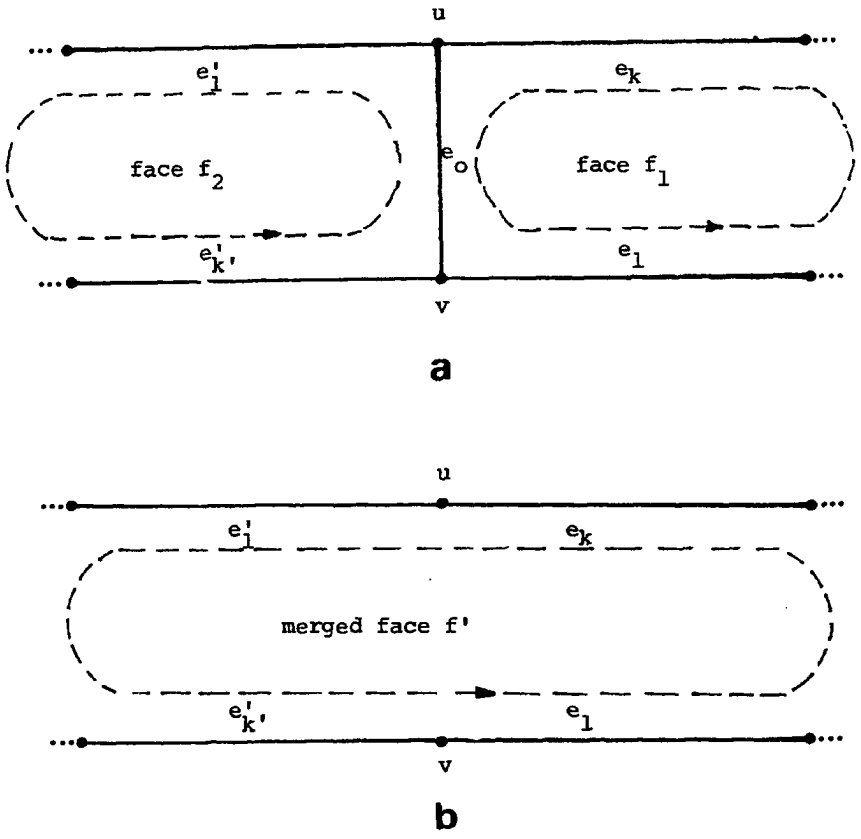

Fig. 3. (a) Oriented graph (G, θ). (b) Oriented graph (G', θ') resulting from a type 1 deletion of edge {u, v}.

loss of generality we can assume that DFS_1 terminates before DFS_2. In this case, Lemma 3.1 states that u and v are in distinct connected components. Hence, we can reset CC(w) = u for each vertex w visited by DFS_1. Also, we must appropriately modify face f. If, originally,

$$D(\theta)(f) = (e_0 = \{u, v\}, e_1, \ldots, e_{i-1}, e_i = \{u, v\}, e_{i+1}, \ldots, e_k, e_0, \ldots)$$

then the resulting (G', θ') has two induced faces f', f'' where

$$D(\theta)(f') = (e_1, \ldots, e_{i-1}, e_1, \ldots)$$

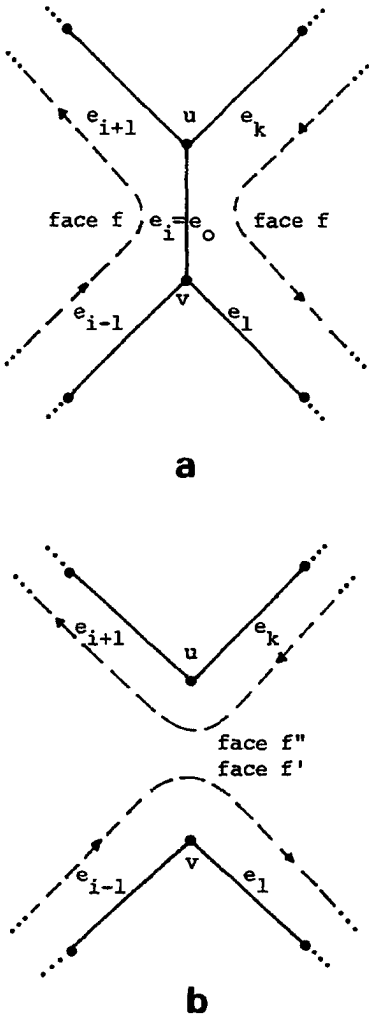

Fig. 4. (a) Oriented graph (G, θ). (b) Oriented graph (G', θ') resulting from a type 2 or 3 deletion of edge {u, v}.

and

$$D(\theta)(f'') = (e_{i+1}, \ldots, e_k, e_{i+1}, \ldots)$$

(see Fig. 4). Again, these can be constructed by a constant number of linked list operations plus $O(\log(|V_1| + |V_2|))$ steps to compute *face*((u, v)) and to appropriately modify *face*. The cost of a type 2 deletion is thus

$$O(\min\{|V_1|, |V_2|\}) + \log(|V_1| + |V_2|),$$

where $V_1, V_2 \subseteq V$ are the connected components of G' containing u, v respectively.

The case where DFS_1 and DFS_2 visit a common vertex is called a *type* 3 *deletion*. By Lemma 3.1, the genus decreases by 1, and connectivity mapping CC need not be modified since u and v are on the same connected component of G'. The cost of a type 3 deletion is O(n). Face f is modified just as in the immediately previous case.

After these type 2 or 3 deletion modifications, we reset $visit_1(w)$ and $visit_2(w)$ to *false* for all vertices w visited by DFS_1 and DFS_2.

3.2. Theorem. *Execution of our algorithm costs* $O(gn + n \log n)$ *total time for the dynamic (deletion) connectivity problem of size* n *and genus* g.

Proof. We have only constant time cost for each vertex deletion and connectivity test. We require only a constant number of steps on each type 1 or any deletion operation. We have a cost of O(n) steps for each type 3 deletion operation but notice that the genus drops by 1. Thus, the total cost for all type 3 deletion operations is O(gn). On the type 2 deletion operations, the genus remains the same, but the number of connected components increases by 1. Suppose H is the forest of derived connected components, where each node of H corresponds to a connected component of a derived graph, and each internal node is weighted by the minimum size of its two immediate successor components plus the logarithm of the sum of their sizes. Then, the total cost of processing all the type 2 deletion operations is at most a constant factor times the sum of all the weights of the internal nodes of H. It is easy to show that the worst-case cost is when H is a balanced binary tree and hence the total cost of all type 2 deletion

operations is O(n log n). This implies the total cost of all operations to be O(ng + n log n) as claimed. □

4. Complete dynamic problems

There are a large number of problems with linear time sequential RAM algorithms on a single input instance, but which seem to require a complete recomputation in the worst case if a single symbol of the input is modified. Many of these problems are complete in deterministic polynomial time with respect to log space TM reductions.

Examples are:

(1) the acceptance of a linear time TM,
(2) path system [2],
(3) the Boolean circuit evaluation problem [12],
(4) unit resolution [11], and
(5) depth-first search numbering of a graph [14].

In fact, all the above problems have the following properties:

(P1) the known reductions between these problems can be done in linear time by a sequential RAM.

Furthermore,

(P2) (given a suitable encoding) each of these reductions is *constant time updatable*: if we modify one symbol of input to an already computed reduction, then the reduction (but not necessarily the problem) can be recomputed in constant time on a sequential RAM.

Consider the dynamic problem of processing a sequence of n single bit modifications to an input instance of any of the above problems of size n, where the problem satisfies (P1) and (P2). It follows from (P1) and (P2) that if any of the resulting dynamic problems can be solved in $t(n) = o(n^2)$ time, then all these dynamic problems can be solved in O(t(n)) time.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, MA, 1974).

[2] S.A. Cook, An observation of time-storage trade-off, J. Comput. Systems Sci. 9 (1974) 308–316.

[3] J. Edmonds, A combinatorial representation for polyhedral surfaces, Amer. Math. Soc. Notices 7 (1960) 646.

[4] S. Even and Y. Shiloach, An on-line edge deletion problem, J. Assoc. Comput. Mach. 28 (1981) 1–4.

[5] I. Fillotti, G. Miller and J.H. Reif, On determining the genus of a graph in $O(v^{O(g)})$ steps, 11th Symp. on Theory of Computing (1979) 27–37.

[6] G.N. Frederickson, Data structures for on-line updating of minimum spanning trees, Proc. 15th Ann. ACM Symp. on Theory of Computing (1983) 252–257.

[7] L.J. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, Proc. 15th Symp. on Theory of Computing, Boston, MA (1983) 221–234.

[8] J.E. Hopcroft and R.E. Tarjan, Efficient planarity testing, J. ACM 21 (1973) 549–568.

[9] J.E. Hopcroft and R.E. Tarjan, Dividing a graph into triconnected components, SIAM J. Comput. 2 (3) (1973).

[10] J.E. Hopcroft and R.E. Tarjan, Efficient algorithms for graph manipulation, Comm. ACM 16 (6) (1973) 372–378.

[11] N.D. Jones and W.T. Laaser, Complete problems for deterministic polynomial time, Theoret. Comput. Sci. 3 (1977) 105–117.

[12] R.E. Ladner, The circuit value problem is log space complete for P, SIGACT News 7 (1) (1975) 18–20.

[13] G. Miller, An additivity theorem for the genus of graph, MIT Tech. Rept., 1983.

[14] J.H. Reif, Depth-first search is inherently sequential, Inform. Process. Lett. 20 (5) (1985) 229–234.

[15] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.

[16] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. Assoc. Comput. Mach. 22 (1975) 215–225.

[17] A. White, Graphs, Groups and Surfaces (North-Holland/American Elsevier, New York, 1973).

The parallel computation of minimum cost paths in graphs by stream contraction

V. Pan *

Department of Mathematics and Computer Science, Lehman College, CUNY, Bronx, NY 10468, USA and Computer Science Department, SUNY Albany, Albany, NY 12222, USA

J. Reif **

Computer Science Department, Duke University, Durham, NC 27706, USA

Abstract

We accelerate by a factor of $\log n$ and with no increase of the processor bound our previous parallel algorithm for path algebra computation in the case of the minimum cost path computation in an n-vertex graph and, more generally, wherever the path algebra has an order relation defined by its $\oplus$ operation. The acceleration is obtained by means of a novel technique of stream contraction.

Keywords: Parallel algorithms, computational complexity, path algebras

1. Introduction. General stream contraction technique and the main result for path algebra

It has been well recognized that numerous important computations of paths in an n-vertex graph $G=(V, E)$ can be both unified and simplified by means of reducing them to the evaluation (over a fixed dioid, also called semiring or path algebra) of the quasi-inverse A^* given an $n \times n$ matrix $A=A(G)$ associated with the graph G (see [1,2,4,6], for details).

In the present paper we propose a novel general technique of *stream contraction* for the acceleration of parallel algorithms by means of their systolic rearrangement and demonstrate its power by accelerating the recent parallel algorithms of [6] for several important path algebra computations. To show how this technique of stream contraction works, we will now describe two algorithms. The first generalizes the algorithm of [6] and the second represents its new acceleration. We assume that A, X and W (with subscripts) are matrices related to each other through the functions f, g, p and q; d and $K(h)$ for $h=0, 1, \ldots, d$ are fixed positive input integers, $A_{0,0}$ is an input matrix and $A_{h,K(h)}=W_{h,K(h)}$, for $h=0, 1, \ldots, d$, are the out-

* Supported by NSF Grants DCR 85-07573, CCR8805782 and CCR 9020690 and by PSC CUNY Awards No. 668541, No. 669290 and No. 661340.

** Supported by Air Force Contract AFOSR-87-0386, ONR Contract N00014-87-K-0310, DARPA/ISTO Contract N00014-88-K-0458, ARO Contract DAAL03-88-K-0195, and NASA/CESDIS Subcontract 550-63 NAS 5-30428 URSA.

Preprint of paper appearing in *Information Processing Letters*, Vol. 40, October 25,1991, pp. 79-83.

put matrices. For path algebra computations, the matrices represent the input, output and auxiliary graphs, and the output defines the paths and their properties.

Algorithm 1.

for $h = 0, 1, \ldots, d$ **do**
  for $k = 0, 1, \ldots, K(h) - 1$ **do**
    $X_{h,0} = p(A_{h,0})$
    $W_{h,0} = f(X_{h,0})$
    $W_{h,k+1} = q(W_{h,k})$
    $A_{h+1,0} = g_h(W_{h,K(h)}, A_{h,0})$
    $A_{h,K(h)} = A_{h,0}$
  end for
end for

Algorithm 2.

for $h = 0, \ldots, d$ **do**
  $X_{h,0} = p(A_{h,0})$
  $W_{h,0} = f(X_{h,0})$
  $A_{h+1,0} = g_h(W_{h,0}, A_{h,0})$.
end for
for $h = 0, \ldots, d$ **do**
  for $k = 0, \ldots, K(h) - 1$ **do**
    $W_{h,k+1} = q(X_{h,k}, W_{h,k})$
    $A_{h+1,k+1} = g_h(W_{h,k+1}, A_{h,k})$
    $X_{h,k+1} = p(A_{h,k+1})$
  end for
end for

Applying Algorithms 1 and 2, for appropriate functions f, g, p, q and $K(h)$, to certain path algebra computations, we will arrive at the same desired output (see below). If we assume that each evaluation of f, g, p or q can be performed in a single time-step on a single processor, then Algorithm 1 takes $\sum_{h=0}^{d-1} K(h)$ time-steps on a single processor and cannot run faster even if more processors are available, whereas Algorithm 2 takes $3(d + 1 + \max_h K(h))$ steps using $d + 1$ processors because the second "for" loop (for $h = 0, \ldots, d$) can be performed concurrently.

In the particular case of the algorithm of [6] as Algorithm 1, we achieve a speedup of $c \log n$ for a positive constant c; our acceleration technique requires no increase of the asymptotic processor bounds and applies to a large and important class of the minimum cost path computations in the planar graphs or planar digraphs G having no negative cost cycles. Furthermore, we may allow negative cost cycles in G provided that we compute paths consisting of at most $n = |V|$ edges over dioids (semirings) S whose operation $+$ defines an order relation, so that $a + b \leqslant a$, for any pair of the elements of S [1,2,4,6]. Moreover, similarly to [6], we may relax the assumptions that the input graph or digraph G is planar; what we actually need is just an $s(n)$-separator family defined by a fixed separator tree of G, where $s(n) = O(n)$; in particular, for planar graphs, $s(n) = O(\sqrt{n})$.

With appropriate modifications, the stream contraction technique can be applied to many other computational problems, in particular, see [3].

Hereafter, for simplicity we assume that G is an undirected graph (the extension to digraphs G is straightforward, see [6]).

2. The "old" algorithm

We first recall that the algorithms of [6] compute the matrix A^* by means of computing its recursive factorization (given by equations (8) and (9) of [6]),

$$A_0 = PAP^{\mathrm{T}}, \qquad A_h = \begin{bmatrix} X_h & Y_h^{\mathrm{T}} \\ Y_h & Z_h \end{bmatrix}, \qquad (*)$$

$$A_{h+1} = Z_h \oplus Y_h X_h^* Y_h^{\mathrm{T}},$$

$$A_h^* = \begin{bmatrix} I & X_h^* Y_h^{\mathrm{T}} \\ 0 & I \end{bmatrix} \begin{bmatrix} X_h^* & 0 \\ 0 & A_{h+1}^* \end{bmatrix} \begin{bmatrix} I & 0 \\ Y_h X_h^* & I \end{bmatrix}, \qquad (**)$$

where P denotes a permutation matrix, $h = 0, 1, \ldots, d$, $d = O(\log n)$, $n = |V|$, the number of vertices of the input graphs $G = (V, E)$, and where the choice of P and the partition of the $n_h \times n_h$ matrix A_h into its blocks X_h, Y_h, Z_h and Y_h^{T} depends on the separator structure of the graph and is computed in the nested dissection algorithm of [5], which is the basis of the algorithm of [6]. The algorithm of [6] consists in recursive

evaluation of X_h^* and A_{h+1}, for $h = 0, 1, \ldots, d$, where in the case of the minimum cost paths with no negative cost cycles the quasi-inverse X_h^* (given X_h) is computed by using the matrix equations $X_h^* = (I \oplus X_h)^i$ for all $i \geqslant s(n_h) - 1$ (see [1–3,6]), and then the matrices A_{h+1} (and thus X_{h+1}) are computed by using (*). The computation is effective because n_h decreases as a geometric progression as h grows and X_h is a block diagonal matrix with diagonal blocks of smaller size (of the size of the separators), as these are well-known properties of the generalized nested dissection algorithms, to which class the algorithm of [6] belongs. On the other hand, the evaluation of X_{h+1}^* only starts when A_{h+1} and thus its submatrix X_{h+1} has been computed, and therefore, only when the quasi-inverse X_h^* has been evaluated (see (*)). Therefore, the parallel time bound of the algorithms of [6] equals the sum of the respective bounds for computing X_h^* for all h, $h = 0, 1, \ldots, d$, d being of the order of $\log n$. In the present paper, we start computing X_{h+1}^* before we end computing X_h^* and this way speed up the parallel computations.

To reconcile this algorithm with Algorithm 1 of the Introduction, we set

$$A_{0,0} = A_0, \qquad X_{0,0} = X_0,$$

$$p(A_h) = X_h \quad \text{(according to (*))},$$

$$f(X) = I \oplus X, \qquad q(W) = W^2,$$

$$g(W, A_h) = Z_h \oplus Y_h W Y_h^{\mathrm{T}} \quad \text{(according to (*))}.$$

Then we observe that $W_{h,k+1} = (I \oplus X_h)^{2^k}$ and, therefore, $W_{h,K(h)} = X_h^*$ since $2^{K(h)} \geqslant s(n_h) - 1$, $A_{h,K(h)} = A_{h,0} = A_h$ for all h (as desired).

3. The new algorithm

Our new improvement relies on a systolic rearrangement of the algorithm of [6], so that computing X_{h+1}^* (and therefore, A_{h+1}^*) starts before the matrix X_h^* has been evaluated.

In our new accelerated factorization, we define a sequence of matrices $A_{h,k}$; we proceed recursively in h, for $h = 0, 1, \ldots, d$, and in k, for $k = 0, 1, \ldots, d + \lceil \log_2 n \rceil$, starting with $A_{0,0} = A_0$. For all h and k, we let

$$A_{h,k} = \begin{bmatrix} X_{h,k} & Y_{h,k}^{\mathrm{T}} \\ Y_{h,k} & Z_{h,k} \end{bmatrix}, \tag{1}$$

$$W_{h,0} = I \oplus X_{h,0},$$
$$A_{h+1,0} = Z_{h,0} \oplus Y_{h,0} W_{h,0} Y_{h,0}^{\mathrm{T}}, \tag{2}$$

$$W_{h,k+1} = (X_{h,k} \oplus W_{h,k})^2, \tag{3}$$

$$A_{h+1,k+1} = Z_{h,k} \oplus Y_{h,k} W_{h,k+1} Y_{h,k}^{\mathrm{T}}. \tag{4}$$

We first compute the matrices $W_{h,0}$, $A_{h+1,0}$, for all h, by using (2); then we recursively apply (3) and (4) in order to compute first the matrices $W_{h,1}$, $A_{h+1,1}$, for all h, then the matrices $W_{h,2}$, $a_{h+1,2}$, for all h, and so on, ending with $W_{h,k}$ and $A_{h+1,k}$ for $k = d + \lceil \log_2 n \rceil$. Finally, we set $X_h^* = W_{h,k}$, $A_h = A_{h,k}$, for $k = h + \lceil \log_2 n \rceil$ and for all h, which defines the desired recursive factorization (*), (* *).

In an alternate version of this algorithm, we end the computation with $k = \lceil \log_2(s(n_0) - 1) \rceil$ and arrive at (*), (* *) by setting $X_h^* = W_{h,k}$, $A_h = A_{h,k}$ for $k = \lceil \log_2(s(n_h) - 1) \rceil$ and for all h (see Remark 2 below). (We give two versions of the algorithm and of the correctness proof to demonstrate more fully some variations of the stream contraction technique.)

To reconcile the latter algorithm (in its second version) with Algorithm 2 of the Introduction we set

$$X_{h,k} = p(A_{h,k}) \quad \text{(according to (1))},$$

$$f(X) = I \oplus X, \qquad q(X, W) = (X \oplus W)^2,$$

$$g_h(W, A_{h,k}) = Z_{h,k} \oplus Y_{h,k} W Y_{h,k}^{\mathrm{T}}$$

according to (1) and (4)).

In the next section (see Remark 7, Theorem 1 and Lemma 5) we will prove that $A_h = A_{h,K(h)}$, $W_{h,K(h)} = X_h^*$ (as desired).

In the resulting algorithm, we avoid computing the auxiliary quasi-inverses X_h^*, and all the operations used are similar to ones used for computing A_{h+1} according to (*) provided that Z_h, Y_h and X_h are available cost-free. More precisely, this applies to computation of $A_{h+1,0}$ according to (2). The asymptotic complexity estimates for (2) are

thus the same as for the algorithm of [6] except that the parallel time is now less by the factor of $c\log n$ (for a positive constant c), since here we do not compute X_h^*. Next, for every k, compute $W_{h,k+1}$ and $A_{h,k+1}$ for all h according to (3) and (4). Then again, no quasi-inverses need be computed at these stages, and we only need to perform the matrix additions and multiplications. For every h, we perform $d+\lceil\log_2 n\rceil = O(\log n)$ such matrix operations by using $O(\log^2 n)$ time and

$$O\left(s(n_h)^3/\log n_h\right)$$

processors, under EREW PRAM, and $O(\log n)$ and $s(n_h)^3$ under a randomized CRCW PRAM, where A_h is an $n_h \times n_h$ matrix. We do this concurrently for all $O(\log n)$ values of h, and since n_h decreases as a geometric progression as h grows, we arrive at the overall asymptotic time and processor bounds

$$O(\log^2 n) \quad \text{and} \quad O\left(s(n)^3/\log n\right),$$

respectively, under EREW PRAM, and $O(\log n)$ and $O(s(n)^3)$, under a randomized CRCW PRAM, which improves the asymptotic complexity bounds of [6], as we claimed.

4. The correctness proof

The correctness of the algorithm is immediately implied by Lemma 5, by Remark 6 below and by the following theorem:

Theorem 1. *$A_{h,k} = A_h$, for all h, if $k = h + \lceil\log_2 n\rceil$.*

The theorem follows from the equations (*), (1)–(4), and Lemmas 3 and 5 below (see also Remark 6 below). To prove and even to state Lemmas 3 and 5, we will need some auxiliary definitions.

Definition 2. Hereafter, $(V)_{i,j}$ denotes the (i, j)-entry of a matrix V, $|p|$ denotes the number of distinct edges (that is, the length) of a path p, and $c(p)$ denotes the cost of a path p, defined as the sum of the given costs of all the edges of p. We will assume that all the graphs are complete, for we may interpret any absent edge as an edge of infinite cost.

Lemma 3. *For all h, k, i, and j, we have:*

$$(W_{h,k})_{i,j} \geqslant (X_h^*)_{i,j}, \qquad (A_{h,k})_{i,j} \geqslant (A_h)_{i,j}.$$

Moreover, if $k \geqslant 1$, then

$$(W_{h,k-1})_{i,j} \geqslant (W_{h,k})_{i,j}, \quad (A_{h,k-1})_{i,j} \geqslant (A_{h,k})_{i,j}.$$

Lemma 3 immediately follows from the definition of the recursive factorizations (*) and the equations (1)–(4) (here, we apply induction on k).

Definition 4. We will partition the set V of all the vertices of the graph G associated with the matrix A_0 into the subsets $V_0, V_1, \ldots, V_d$ associated with the matrices $X_0, X_1, \ldots, X_d$ of the recursive factorization (*). (In fact, this partition of the set V is computed in the nested dissection algorithm of [5].) Hereafter, let $\hat{V}_h = \bigcup_{g=0}^{h} V_g$ and let $G_A(\hat{V}_h)$ denote the maximum subgraph of G induced by $\hat{V}_h$ (such that $G_A(\hat{V}_h) = (\hat{V}_h, \hat{E}_h)$, where $\hat{E}_h$ denotes the set of all the edges of G with both endpoints in $\hat{V}_h$).

Lemma 5. *Let p denote a minimum cost path in $G_A(\hat{V}_h)$ between two vertices i and j in $\hat{V}_h$. Let*

$$|p| \leqslant 2^{\max(k-h,0)}. \tag{5}$$

Then

$$(W_{h,k})_{i,j} = (X_h^*)_{i,j} = c(p). \tag{6}$$

Remark 6. The assumption (5) always holds for $k \geqslant h + \lceil\log_2 n\rceil$ and for any minimum cost path p in G, since we suppose that $|p| \leqslant n$ (see the Introduction).

Let $k = d + \lceil\log_2 n\rceil - 1$, so that (5) holds, due to Remark 6. Let us deduce Theorem 1 from Lemmas 3 and 5. Indeed, Lemma 5 implies (6). Furthermore, from Lemma 3, it follows that for all h and k,

$$X_{h,k} \leqslant X_{h,0} = X_h, \qquad Y_{h,k} \leqslant Y_{h,0} = Y_h,$$
$$Z_{h,k} \leqslant Z_{h,0} = Z_h.$$

Substitute these inequalities and (6) into (4) and deduce that

$$A_{h+1,k+1} \leqslant Z_h \oplus Y_h X_h^* Y_h^{\mathrm{T}} = A_{h+1}.$$

Combine the latter inequality with $A_{h+1,k+1} \geqslant A_{h+1}$ due to Lemma 3 and arrive at Theorem 1. □

It remains to prove Lemma 5.

Proof of Lemma 5. Lemma 5 is obvious for $k \leqslant h$, in particular, for $k = 0$. Apply induction on k and always assume that $h < k$. First note that, by the definition of X_h^*, $(X_h^*)_{i,j} = c(p)$, and that, by virtue of Lemma 3, $(W_{h,k})_{i,j} \geqslant (X_h^*)_{i,j}$.

It remains to prove that $(W_{h,k})_{i,j} \leqslant c(p)$. Fix a positive k_0, suppose that the latter unequality holds for all h and for all $k < k_0$, and consider a path p satisfying the assumptions of Lemma 5 for $k = k_0$. Partition p into two subpaths p_1 and p_2 such that $p = p_1 p_2$; the two endpoints of p_s are denoted u_s and v_s for $s = 1, 2$, so that

$$u_1 = i,\ v_1 = u_2,\ v_2 = j \quad \text{where } i, u_2, j \in \hat{V}_h, \quad (7)$$

and

$$|p_s| \leqslant 2^{k_0 - 1 - h}, \quad (8)$$

for $s = 1, 2$.

Note that the induction hypothesis implies that

$$(W_{h,k_0-1})_{u_s,v_s} \leqslant c(p_s), \quad (9)$$

as long as (8) holds.

The inequalities (8) and (9) hold for $s = 1, 2$. On the other hand, (3) implies that

$$W_{h,k_0} = \left(X_{h,k_0-1} \oplus W_{h,k_0-1}\right)^2,$$

so that

$$(W_{h,k_0})_{i,j} \leqslant c(p_1) + c(p_2) \leqslant c(p).$$

This implies (6). □

This was a graph-theoretical proof. Here is an alternate algebraic proof. Recursively apply (3) and deduce that

$$W_{h,k} \leqslant W_{h,k-1}^2 \leqslant W_{h,0}^{2^k} = (I \oplus X_{h,0})^{2^k} = (I \oplus X_h)^{2^k}$$

for all h and k, but $(I \oplus X_h)^{s(n_h)-1} = X_h^*$, and $s(n_h) \leqslant n_h \leqslant n$. □

Remark 7. The latter proof implies that $W_{h,k} = X_h^*$ and $A_{h,k} = A_h$ already for

$$k = \lceil \log_2(s(n_h) - 1) \rceil \leqslant \lceil \log_2(s(n_0) - 1) \rceil.$$

References

[1] R.C. Backhouse and B.A. Carré, Regular algebra applied to path-finding problems, *J. Inst. Math. Applics.* **15** (1975) 161–186.

[2] B.A. Carré, An algebra for network routing problems, *J. Inst. Math. Applics.* **7** (1971) 273–294.

[3] H. Gazit and J. Reif, A randomized parallel algorithm for planar graph isomorphism, in: *Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architecture* (1990) 210–219.

[4] M. Gondran and M. Minoux, *Graphs and Algorithms* (Wiley/Interscience, New York, 1984).

[5] V. Pan and J. Reif, Fast and efficient parallel solution of sparse linear systems, Tech. Rept. 88–19, Computer Science Dept., SUNYA, 1988.

[6] V. Pan and J. Reif, Fast and efficient solution of path algebra problems, *J. Comput. System Sci.* **38** (1989) 494–510.