JOURNAL OF THE AMERICAN WATER RESOURCES ASSOCIATION
AUGUST AMERICAN WATER RESOURCES ASSOCIATION 2004

A FAST RECURSIVE GIS ALGORITHM FOR COMPUTING STRAHLER STREAM ORDER IN BRAIDED AND NONBRAIDED NETWORKS[1]

Alexander Gleyzer, Michael Denisyuk, Alon Rimmer, and Yigal Salingar[2]

ABSTRACT: Stream ordering is a useful property of every river network, having a wide range of applications. A method for determining stream orders that quickly and easily addresses various network topologies and magnitudes is therefore needed. This paper introduces a general recursive stream ordering framework for vector hydrography. It also presents a linear, O(n), stream ordering procedure for braided river networks, which is a major improvement to the existing quadratic, $O(n^2)$, procedure. The discussion includes results and interpretations, and the appendices present procedure pseudocodes and thorough line by line explanations.
(KEY TERMS: Strahler stream order; geographic information system; recursive algorithm; software; drainage; braided networks.)

Gleyzer, Alexander, Michael Denisyuk, Alon Rimmer, and Yigal Salingar, 2004. A Fast Recursive GIS Algorithm for Computing Strahler Stream Order in Braided and Nonbraided Networks. Journal of the American Water Resources Association (JAWRA) 40(4):937-946.

INTRODUCTION

This paper describes an algorithmic framework that was developed to create the fastest possible procedure for vector stream network ordering using the Strahler method. In the process of upgrading the vectored hydrographic layer of Israel's National Geographic Information Systems (GIS) database (prepared by the Survey of Israel) it was found that the stream ordering module became too time consuming. The detailed hydrographic data were required for a hydrological and chemical fate model that was integrated into a GIS of the Upper Catchment of the Jordan River.

Stream ordering helps to create a hierarchy for streamflow networks, which usually consist of thousands of tributaries. It is applied in various types of streamflow models [e.g., LASCAM (Sivapalan and Viney, 1994), MIKE 11 (DHI Software, 2003) and Eagle Point Watershed Modeling (Eagle Point Software Corporation, 2004]. In particular, it may be used for hydrological problems such as identification of areas vulnerable to flooding and in studies of chemical and ecological systems of drainage basins (Cole and Wells, 2003).

Among several stream ordering methods such as those of Horton, Strahler, Scheidegger, Woldenberg, and Shreve (Doornkamp and King, 1971), only the methods developed by Strahler and Shreve are widely used today. In this work, the method developed by Strahler was used. Strahler's stream ordering method (Strahler, 1957) assigns an order of k = 1 to all streams that have no tributaries. Thereafter, proceeding downstream, when two (or more) streams of the same order k meet, they form a stream with an order of k + 1. However, when a stream of order k is met by one of a lower order, no change occurs in the downstream order. Figure 1a demonstrates a simple stream network with Strahler orders.

[1]Paper No. 03043 of the *Journal of the American Water Resources Association* (JAWRA) (Copyright © 2004). **Discussions are open until February 1, 2005.**

[2]Respectively, Software Engineer and GIS Applications Specialist, STAV-GIS Ltd., 20179 Teradyon, Misgav, Israel; Researcher, Kinneret Limnological Laboratory (KLL), 14950 Migdal, Israel; and Managing Director, STAV-GIS, Ltd., 20179 Teradyon, Misgav, Israel (E-Mail/Salingar: stav_gis@netvision.net.il).

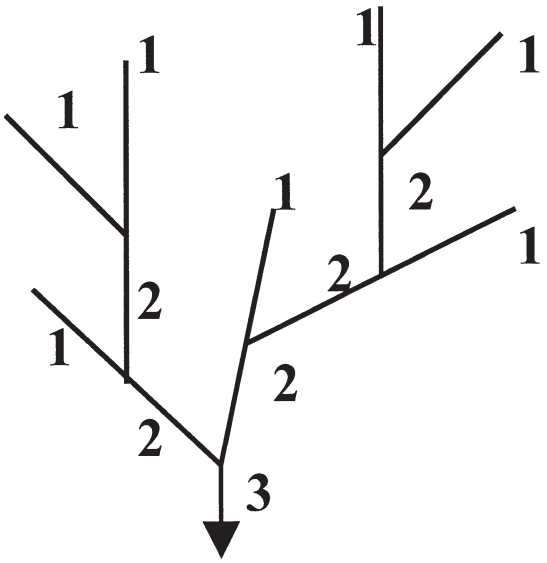

Figure 1a. Stream Network With Strahler Orders.

DATA PREPARATION

An 'arc' in GIS is defined as an ordered string of vertices (x, y coordinate pairs) that begin at one location and end at another. Connecting the arc's vertices creates a line. The vertices at each endpoint of an arc are called 'nodes.' A 'pour point' is defined as a selected endpoint along the streamflow network, into which that network flows.

To implement a stream ordering algorithm, the stream input data have to meet several criteria. First, all arcs in the processed stream network must be connected, so that a continuous path to the pour point can be obtained. Second, all arcs and nodes of the network must have unique identification (ID) numbers. This is a built-in property of GIS software, which assigns unique identifiers to different geometric features belonging to a common layer. Also, the network must maintain an 'arc-node topology' (representing connectivity between arcs and nodes), meaning that each arc should hold the ID numbers of its endpoint nodes. The network's pour point must be determined beforehand, as it is also used as an input for the ordering procedure.

In practice, only the network's arc-node topology is required for algorithm input, since its spatial representation plays no role in stream ordering patterns. This provides the option of applying an ordering procedure unrelated to burdensome graphics manipulations, dramatically increasing the procedure's overall execution time.

STRAHLER STREAM ORDERING ALGORITHMIC FRAMEWORK

According to its definition, it is evident that Strahler's stream order has a recursive nature, since each arc's order depends on the orders of its inflow arcs. This property has been used in various GIS tools that provide stream ordering (Lu *et al.,* 1996; Tarboton, 2000, 2002). However, those are ready to use software packages, which do not reveal the actual algorithm behind them. Hence, a formal and thorough algorithmic definition of the recursive stream order computation framework is of interest.

The proposed algorithmic framework can be easily implemented by hydrologists. It provides a 'skeleton' for additional stream network computations and schemes such as Strahler segments (presented in this section) and braided networks (presented in the following section). In this section, such a framework is presented, which will be further developed into an innovative solution for dealing with braided stream networks.

To perform stream network analysis the network could be regarded simply as a *graph* (either *directed* or *undirected*, depending on the procedure's application). Graphs are common data structures, and as such can be analyzed by various recursive traversal algorithms, possessing good running time properties. Network structure in GIS could be viewed as a graph, where nodes are also graph 'nodes' and arcs are graph 'edges.' For stream networks in particular, statistical graph theory was used (e.g., Scheidegger, 1968, to describe Horton's law of stream numbers).

Each graph can be represented either by an *adjacency matrix* or a set of *adjacency lists*. An adjacency matrix contains an equal number of rows and columns according to the number of graph nodes. Every *edge* from node i to node j is denoted by the Boolean value of *true* at the matrix location (i, j), and, in the case of an undirected graph also at the location (j, i). All other matrix 'cells' are assigned the value of *false*. This representation is good for richly populated graphs, where an edge is present for almost every node pair. Alternatively, *adjacency lists* hold a list of adjacent nodes for each graph node (i.e., a list of nodes to which there is an edge from that specific node). Thus, a list entry exists only for actual edges present in the graph. The second notation (with slight modifications) was preferred in this case because of the sparsely populated nature of stream networks.

Since each network's pour point is determined prior to applying stream order and the presented framework deals with simple nonbraided networks (braided networks are defined in the following section), the network could be regarded as an undirected graph. As

visiting and ordering arcs (i.e., edges) and not nodes are of interest, adjacency lists that contain adjacent (flowing both in and out) arcs for each stream node were used. The complete algorithm pseudocode as well as a thorough line by line description is given in Appendix A.

An additional network property that is also derived during this recursive river network traversal is the *Strahler river segment* (Scheidegger, 1970). A river segment is a continuous path within a network bearing the same stream order. Thus, every network has a set of unique segments for each stream order. Consider, for example, the stream network depicted in Figure 1b. This network contains arcs with Orders 1 through 3. It has seven river segments of Order 1, two river segments of Order 2 (one depicted with dense round dots and another with a dashed line), and one segment of Order 3 (depicted with a dash/dot line). The general rule is that when arcs of two (or more) streams meet at a node, their respective river segments (regardless of their order) come to an end, and a new downstream segment of a greater order is initiated. The exception is in the case where only one arc bears the maximal stream order among those streams. In such a case, the downstream arc (into which all upstreams flow) does not change its order and consequently continues the river segment of the upstream arc bearing the same maximal stream order.

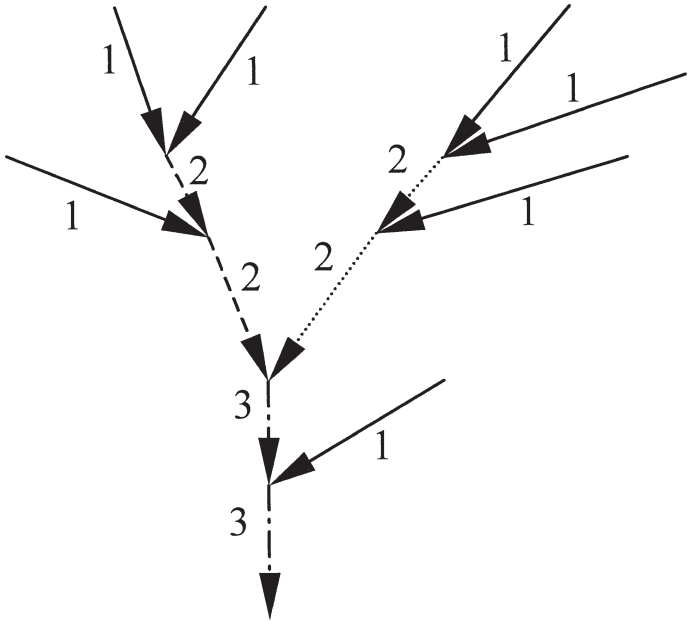

Figure 1b. Stream Network With Orders and Segments.

The stream ordering procedure assigns each arc an ID of its river segment, which is unique within a certain stream order of the processed network. For instance, if a network has five river segments of Order 3, then every arc having the Order 3 will be assigned a segment ID from 1 to 5. If that network also has eight river segments of Order 2, then every arc with Order 2 will be assigned a segment ID from 1 to 8. In this way, the identification of a unique river segment of any order in the network can be performed using a composite 'order segment ID' key.

The overall running time of the presented framework algorithm is a linear function of the number of arcs n in the network, O(n), which is certainly expected from this kind of a recursive network (graph, tree) traversal. Note that the particular computed property – Strahler segments – did not add to the traversal time, so both the stream order and the segments are obtained as a linear function of the number of arcs.

STRAHLER STREAM ORDERING ALGORITHM FOR BRAIDED RIVER NETWORKS

After the general recursive stream ordering framework was completed, it could be used as a 'skeleton' for independently developed functions, intended to deal with more complex stream networks, such as the braided networks.

In a braided network, a stream can split into a number of downstream flows, and these flows can merge again further downstream. Alternatively, there can be 'channels' linking different 'branches' of the same network. All these and similar cases introduce cycles in the respective undirected graphs representing them.

Consider the braided river network depicted in Figure 2 (a modified version of the network in Figure 1). If the arrows are ignored, this network can be regarded as an undirected graph, and a number of braided elements can easily be recognized as they create cycles in the graph. However, the stream order along these elements does not change in most cases since they originate from the same upstream arc. For instance, all arcs between nodes *a* and *b* have the stream Order of 1, because they all come from the same first order arc.

Several methods to analyze such cases, while maintaining the running time properties of the nonbraided (framework) solution, were considered. For example, the biconnectivity mechanism (Aho *et al.,* 1974) was employed. Such a mechanism could be used to identify *biconnected components* in the undirected graph, which in this case are the braided parts of the stream network. After that, it has been postulated that each biconnected component has the same stream order for all its arcs. However, this proved to be true only for

biconnected components 'bounded' by a maximum of two *articulation points* (as in the above mentioned example, between nodes *a* and *b*). If an arc exists that 'joins' the component at an additional articulation point (e.g., Node *d*, in addition to Nodes *c* and *e*), then only the component's upstream parts (for that node) reliably maintain their stream order, while the downstream ones are affected by the joining arc and can change their order (as is the case of arc *de*).

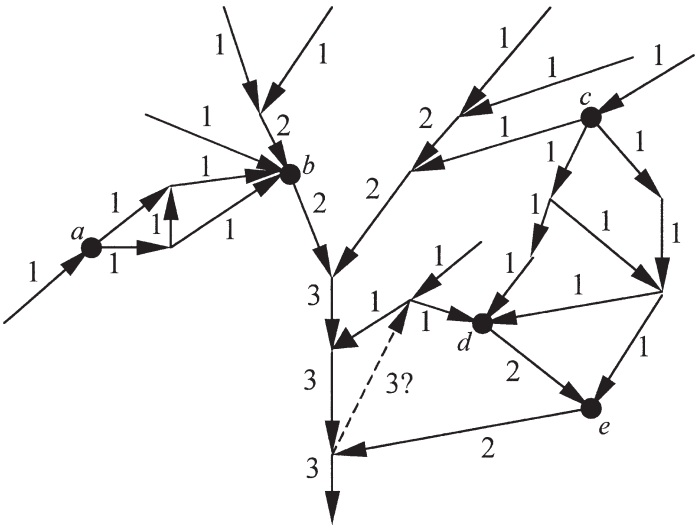

Figure 2. Braided Stream Network With Orders.

Consequently, the braided part cannot be treated as one unit for stream ordering. Therefore, the conclusion is that braided networks must be treated as directed graphs. This introduces another data preparation activity ascertaining that all arcs appear in the network with correct downstream direction depicted in their *from* nodes and *to* nodes.

In the course of developing the new stream ordering procedure, the principle of Lanfear's algorithm (Lanfear, 1990) was used. Lanfear suggested an iterative ordering procedure that could also deal with braided networks. In this procedure another property was added to each arc, containing the upstream node at which the stream order value of the arc originated. Considering this new property, the arc's order does not follow the general rule (described in the Introduction) and does not increase if all arcs of the same maximal order originate from the same node. The complete braided network stream ordering algorithm, built upon the previously described framework, is given in Appendix B. This braided algorithm also has a linear running time, similar to the simpler case described earlier.

DISCUSSION AND CONCLUSIONS

The proposed general recursive stream ordering framework possesses good linear running time properties. By its nature, it resembles the well known and widely used *Depth First Search* (*DFS*) graph traversal algorithm first introduced by Tarjan (1972). The DFS algorithm is also recursive, and it also accesses the graph's furthermost nodes (or, in this case, those farthest upstream), reachable from a certain node before proceeding to that node's 'siblings.' Eventually, the recursive traversing of a stream network, starting at its pour point, can be generalized and reused for other stream network properties whose definitions are also recursive, as in the case of Strahler's stream order (e.g., a Shreve stream ordering algorithm). Consequently, the suggested framework can be easily adapted for those properties as well.

Despite the resemblance to DFS, it has been shown that it is useful to have a clearly and formally defined algorithmic framework, thoroughly describing an allegedly simple task such as a recursive stream network traversal. Based on this framework, additional tasks could be easily introduced (e.g., Strahler segments) while maintaining the overall integrity and running time.

This framework also allowed the development of a faster linear procedure for braided networks stream ordering (see previous section). Two tests were performed to compare between Lanfear's algorithm and the proposed recursive braided procedure. For the first test, a tool developed by Hornby (2003), who implemented Lanfear's algorithm, was used. The running time for a 22,000 arc network was very long, just as Hornby stated: "Large networks (1000+ polylines) can take a long time to compute and may need to be run overnight." The second test was more objective. To fairly evaluate the two procedures, both were coded in the same environment, using the same programming language and identical data structures (i.e., dictionaries, lists, arrays, etc.), and run on the same machine. While processing the same 22,000 arc network, it took Lanfear's algorithm more than 220 seconds to complete the ordering, while the suggested new recursive algorithm reached satisfactory results for the same network within 22 seconds. When another 26,000 arc network was processed, the suggested algorithm solved it in 26 seconds, whereas Lanfear's algorithm took 600 seconds. The results clearly show the linear running time nature of the suggested algorithm and the fast growing nature of Lanfear's. After closely analyzing both algorithms it was found that the running time of Lanfear's algorithm depends asymptotically on n^2 [$O(n^2)$], which is certainly less efficient and more time consuming than the proposed

linear procedure. The suggested preprocessing procedures (which prepare the required data structures) are also linear, and in both mentioned examples they took about five seconds to complete. Quality control for the resulting stream ordering was performed visually by symbolizing the arcs according to their stream order, revealing that all arcs were attributed by the appropriate stream order. The resulting stream orders calculated by the suggested algorithm were also compared to those assigned by Lanfear and found to be identical.

To implement algorithms based on the proposed framework, the selected programming language needs to support recursive procedure calls. All the 'dictionaries' mentioned above are actually known as *associative arrays* or *hashes*. These are data structures that are not sequential and continuous like regular arrays and that can use noninteger values for indexing. Most programming languages in use today have some kind of representation for these data structures.

The proposed procedure obtained two important results. First, instead of closed and completed **software** implementations, which are in this case limited and awkward, a general, effective, and efficient recursive stream ordering **algorithm** is formalized and thoroughly described. Second, the existing stream ordering procedure for braided stream networks was significantly improved, from nearly quadratic to linear running time.

APPENDIX A
STRAHLER RECURSIVE STREAM ORDERING ALGORITHM FRAMEWORK
(with an additional property of Strahler segments)

Note: Throughout Appendices A and B, all procedure names, data structures, and variables start with capitals, while pseudocode keywords (like **if** or **do**) start with small letters and appear in bold. All data structures and variables that are global (external) to the procedure (i.e., maintain the same copy over the recursive calls) are shown in *italic*.

The algorithm employs a short preprocessing procedure, which builds node and arc adjacency 'dictionaries,' which are sets of adjacency lists compiled for each node and arc in the network. This procedure assumes that the input data is organized as described above in the Data Preparation section. Table A1 describes the preprocessing pseudocode for the adjacency dictionaries.

The MakeDictionaries procedure receives a stream network as its argument and consists of a loop, which scans all arcs in the network (Lines 1-4). Two dictionaries are updated at every loop step. At the end of the procedure, the *NodesPerArc* dictionary contains a list of arc endpoint IDs (Arc's FromNodeID and Arc's ToNodeID) for every network arc, while the *ArcsPerNode* dictionary contains a list of adjacent arcs (regardless of direction) for every network node. The running time of this procedure is linear, O(n) (where n is the number of arcs in the network) since every arc is processed only once.

The ordering procedure uses the above dictionaries to access the appropriate nodes and arcs at each recursion level. It starts with the 'pour arc' (i.e., the arc directly connected to the pour point) and recursively scans its upstream arcs, while deriving their stream orders. After that, it determines the pour arc's order according to its upstream orders. The stream ordering algorithm pseudocode is presented in Table A2.

The StreamOrdering procedure assumes that an empty dictionary (global to the procedure itself) named *StreamOrders* has been created prior to execution. It receives two arguments: ArcID – containing the ID of the current pour arc and DirectionNodeID – containing the ID of the pour arc's node in the upstream direction. (This is needed due to the use of an undirected graph for stream network representation, since it is eventually necessary to 'guide' the procedure to follow the arc upstream and not downstream.) The first two lines include the recursion's

TABLE A1. **MakeDictionaries** Preprocessing Procedure.

```
MakeDictionaries(Network)
1       for each Arc ∈ Network
2               do NodesPerArc[Arc's ID] ← (Arc's FromNodeID, Arc's ToNodeID)
3                  ArcsPerNode[Arc's FromNodeID] ← ArcsPerNode[Arc's FromNodeID] ∪ Arc's ID
4                  ArcsPerNode[Arc's ToNodeID] ← ArcsPerNode[Arc's ToNodeID] ∪ Arc's ID
```

TABLE A2. **StreamOrdering** Procedure.

```
StreamOrdering(ArcID, DirectionNodeID)
1   if |ArcsPerNode[DirectionNodeID] | = 1
2       then StreamOrders[ArcID] ← 1
3   else
4       for each Arc ∈ ArcsPerNode[DirectionNodeID]
5           do if Arc ≠ ArcID
6               then (FromNodeID, ToNodeID) ← NodesPerArc[Arc]
7                   if  FromNodeID ≠ DirectionNodeID
8                       then UpstreamOrders[Arc] ← StreamOrdering (Arc, FromNodeID)
9                       else UpstreamOrders[Arc] ← StreamOrdering (Arc, ToNodeID)
10          SegmentIDs[ UpstreamOrders[Arc] ] ← SegmentIDs[ UpstreamOrders[Arc] ] + 1
11      MaxOrder ← 0
12      MaxOrderCount ← 0
13      for each Order ∈ UpstreamOrders
14          do if Order > MaxOrder
15              then MaxOrder ← Order
16                  MaxOrderCount ← 1
17              else if Order = MaxOrder
18                  then MaxOrderCount ← MaxOrderCount + 1
19      if  MaxOrderCount > 1
20          then StreamOrders[ArcID] ← MaxOrder + 1
21          else  StreamOrders[ArcID] ← MaxOrder
22              SegmentIDs[StreamOrders[ArcID] ] ← SegmentIDs[StreamOrders[ArcID] ] – 1
23  Segments[ArcID] ← SegmentIDs[StreamOrders[ArcID] ]
24  return StreamOrders[ArcID]
```

stop condition, which in this case is a source arc (i.e., an arc that has no arcs in its upstream direction). The *ArcsPerNode* dictionary will contain only one arc for the upstream direction node of a source arc – it would be the source arc itself.

According to definition, a source arc is assigned a stream order of 1. Lines 3 through 22 deal with non-source arcs. Line 4 starts a loop over all upstream direction node adjacent arcs. Line 5 makes sure that the procedure does not follow the current arc (indexed by ArcID) since it is also present in the direction node's adjacency list. Line 6 retrieves the endpoint nodes of each upstream arc to determine the upstream arc's direction node ID at line 7. (The *NodesPerArc* dictionary contains both the ID of the current direction node and that of the upstream arc, since both are the current arc's endpoints.) After this is determined, an appropriate recursive call is made either at Line 8 or Line 9. In contrast with *StreamOrders*, UpstreamOrders should be a local dictionary, maintaining a separate copy for each recursive call. After all upstream orders have been derived, the algorithm determines the correct stream order of the current arc at Lines 11 through 21. For this purpose, it counts the number of upstream arcs having the maximum stream order at Lines 13 through 18. According to definition, an arc's stream order is incremented only if there is more than one upstream arc

with the maximal stream order. Otherwise the arc's stream order remains the same as the maximal order of its upstream arcs. This logic is depicted at Lines 19 through 21.

To derive river segments, the stream ordering procedure assumes that a *SegmentIDs* array, global to the procedure, has been created prior to execution. This array will hold the currently used segment ID for each stream order in the network, while incrementing these IDs each time a new segment of a certain order is started. Since the network's maximum stream order is not known in advance, one can either create an array with a number of elements greater than any possible stream network order, or, alternatively, use some dynamic array data structure, which inserts new elements as soon as they are accessed. In any case, all *SegmentIDs* array elements should be initialized to 0 or 1 (the first segment ID in each network). Also, an empty *Segments* dictionary, global to the procedure, is assumed to be created before execution of the procedure. This dictionary will hold the actual segment ID for each network arc. Line 10 increments segment IDs of all upstream orders, since those streams have reached a junction. Line 22 executes in case the current arc maintains the order of one of the upstream arcs (which is the only one with maximum stream-order), bringing that specific segment ID back to where it was before execution of Line 10. Line 23 saves the derived segment ID for the current arc in the *Segments* dictionary.

The StreamOrdering procedure returns the stream order of the current arc (pointed by ArcID) up the recursion tree. After all recursive calls have been completed and the algorithm's execution has ended, the *StreamOrders* and *Segments* dictionaries will contain the proper ordering and segment information for each stream network arc.

The running time of the StreamOrdering procedure is linear, O(n) (where n is the number of arcs in the network), since it processes each network arc only once, which is correct for both loops (Lines 4 through 10 and Lines 13 through 18).

APPENDIX B
STRAHLER RECURSIVE STREAM ORDERING ALGORITHM FOR BRAIDED NETWORKS
(with an additional property of Strahler segments)

Since dealing with directed graphs now, the preprocessing procedure should create an *InflowingArcsPerNode* dictionary (instead of an *ArcsPerNode* dictionary) that would contain for every node only those arcs that flow into that same node. It should also create a *FromNodesPerArc* dictionary (instead of a *NodesPerArc* dictionary) to contain the *from-* nodes of each arc. The modified preprocessing procedure pseudocode is given in Table B1.

The stream ordering algorithm is also modified to accommodate the possibility of braided network parts. The Lanfear property of order originating node is added. For this purpose, an *OriginatingNode* dictionary is created prior to execution in the above preprocessing procedure (Line 4) and initialized to the arcs *from-* nodes. Each arc also now has a corresponding value in a *Visited* dictionary, indicating whether this arc has already been visited in some previously executed recursive call or by its predecessor in the recursion tree. This can happen due to the nature of the braided network, where the same upstream arc could be reached by a number of its downstream predecessors. In this way, when the recursive calls return back to the split node, the algorithm would know not to follow upstream arcs already explored from 'sibling' paths. The modified stream ordering algorithm pseudocode is depicted in Table B2.

The ordering procedure assumes that a *Visited* dictionary, global to the procedure, has been created prior to execution and initialized to the Boolean value of *false* for each arc. It also assumes (similar to the case of a nonbraided ordering algorithm) that an external dictionary (global to the procedure itself) named *StreamOrders* has been created and initialized to 0 for each arc. First, the procedure marks the current arc as *Visited*, so that it will not be accessed more than once. Lines 2 and 3 contain the recursion's stop

TABLE B1. **MakeDictionaries** Preprocessing Procedure (braided).

```
MakeDictionaries(Network)
1     for each Arc ∈ Network
2         do FromNodesPerArc[Arc's ID] ← Arc's FromNodeID
3             InflowingArcsPerNode[Arc's ToNodeID] ← InflowingArcsPerNode[Arc's ToNodeID] ∪ Arc's ID
4             OriginatingNode[Arc's ID] ← Arc's FromNodeID
```

TABLE B2. **StreamOrdering** Procedure (braided).

```
StreamOrdering(ArcID)
1   Visited[ArcID] ← true
2   if | InflowingArcsPerNode[ FromNodesPerArc[ArcID] ] | = 0
3       then StreamOrders[ArcID] ← 1
4   else
5       for each Arc ∈ InflowingArcsPerNode[ FromNodesPerArc[ArcID] ]
6           do if not Visited[Arc]
7               then UpstreamOrders[Arc] ← (StreamOrdering(Arc), OriginatingNode[Arc])
8               else  UpstreamOrders[Arc] ← (StreamOrders[Arc], OriginatingNode[Arc])
9       MaxOrder ← 0
10      MaxOrderCount ← 0
11      for each (Order, Origin) ∈ UpstreamOrders
12          do if Order > MaxOrder
13              then MaxOrder ← Order
14                   MaxOrderCount ← 1
15                   MaxOrderOrigin ← Origin
16              else if  Order = MaxOrder
17                   then if Origin ≠ MaxOrderOrigin
18                        then MaxOrderCount ← MaxOrderCount + 1
19      if MaxOrderCount > 1
20          then StreamOrders[ArcID] ← MaxOrder + 1
21              OriginatingNode[ArcID] ← FromNodesPerArc[ArcID]
22          else StreamOrders[ArcID] ← MaxOrder
23              OriginatingNode[ArcID] ← MaxOrderOrigin
24  if SegmentIDsPerOriginatingNode[ OriginatingNode[ArcID] ] = nil
25      then SegmentIDs[ StreamOrders[ArcID] ] ← SegmentIDs[ StreamOrders[ArcID] ] + 1
26          SegmentIDsPerOriginatingNode[ OriginatingNode[ArcID] ] ← SegmentIDs[ StreamOrders[ArcID] ]
27  Segments[ArcID] ← SegmentIDsPerOriginatingNode[ OriginatingNode[ArcID] ]
28  return StreamOrders[ArcID]
```

condition, which is the case of an arc with no inflowing arcs. Otherwise, Lines 5 through 8 would proceed with a loop over all inflowing arcs. If an upstream arc has not been visited yet (as determined at Line 6), then Line 7 initiates a recursive procedure call, which determines the upstream arc stream order. At this point, the local *UpstreamOrders* dictionary contains both the order and the order's origin node. If the arc has already been visited, its order is obtained from the global *StreamOrders* dictionary. The order's origin node is obtained in both cases from the global *OriginatingNode* dictionary. Lines 11 through 18 find the maximum order among upstream arcs and the number of unique streams bearing that maximum order. Line 15 saves the origin of the current maximum order. Lines 17 and 18 check whether an

additional upstream arc with the current maximum order has a different origin and, in such case, the maximum order count is increased. If all maximum order upstream arcs come from the same origin, the maximum order count stays at 1 as the loop terminates. Lines 19 through 23 establish the current arc's order and its up to date order originating node property. If the order is to be increased (maximum order count equals 1), it is performed at Line 20, and Line 21 're-initializes' the arc's order originating node to its *from-* node (since the order changes here). Otherwise, Line 22 sets the current arc's order to the maximum order of its upstream arc(s) and Line 23 sets the arc's order originating property to the order origin of its maximum order upstream arc(s). The latter equals the order origin of the arc's single maximum order upstream arc, or, in the case of a braided upstream part – to that braided part's order origin.

Strahler segment determination in this new braided version is very similar to the nonbraided one. Here, also, an external *SegmentIDs* array keeps count of the current segment ID for each encountered stream order. However, for the actual segment ID assignment, Lanfear's idea of order's origin (as described above) is employed. When several streams meet at a junction, they can either initiate a new segment (in case there is more than one stream with the same maximum order of different origin) or else a previous segment can continue (in case there is only one stream with the maximum order or more than one stream with the maximum order but of the same origin). In the first case, the downstream arc order's origin will also be 'initialized' to contain this arc's *from-* node, while in the latter case, the downstream arc order's origin will simply 'repeat' that of its upstream order originating segment. This means that each order originating node uniquely identifies each segment in the network, since a new segment is initiated only when a new order's origin is 'initialized.'

This new approach appears in Lines 24 through 27. The procedure now assumes that an empty external dictionary *SegmentIDsPerOriginatingNode* has been created prior to execution. Line 24 checks whether no Strahler segment exists beginning with the order originating node of the current arc. If this is the case, then Line 25 generates a new segment ID for the current stream order and Line 26 assigns that segment ID to the current order originating node. Line 27 saves the appropriate segment ID for the current arc, based on the arc's order originating node.

The new braided version can also easily deal with multiple drainage outlets. To address such cases, the procedure must be executed separately for each pour point (outlet) present in the network. When the procedure first reaches the 'split node,' where the network splits into multiple drainage paths, it will not follow the other paths, since they do not flow into that split node, but rather out of it. Then, in successive calls for other outlets, the procedure can use already determined stream orders of the split node's upstream arcs instead of repeating that upstream part's traversal. Of course, all external dictionaries should be shared among the calls to maintain consistency among them. In any case, the aforementioned geomorphologists intended to apply their ordering schemes for permanent natural stream channels (Horton, 1945) and most of the splits occur in man made situations like channels or dams.

The new braided ordering procedure's running time is still linear (as its framework 'skeleton' is) because, as stated, every network arc is accessed only once during the procedure's execution.

Some arcs can still remain unaccounted for during execution of the ordering procedure. This will happen to arcs with an incorrectly entered direction (for example, the dashed arc in Figure 2). While dealing with this arc, the procedure will determine that every upstream arc of this arc is already *Visited*. In this way, every such an arc will not be assigned a stream order (leaving it at 0), giving the user a useful error indication. If those incorrectly assigned arcs still need to be ordered (in the same way – according to their upstream arcs; e.g., so that the dashed arc in Figure 2 gets the order of 3), a simple iterative (nonrecursive) procedure can easily scan those arcs and deal with them. Such a procedure could be constructed with a slightly modified nonrecursive part of the original procedure (Lines 9 through 27) while substituting locally obtained order and origin values (i.e., stored in UpstreamOrders) by those already correctly derived and saved in the *StreamOrders* and *OriginatingNode* dictionaries by the original recursive procedure. This kind of supplementary procedure is obviously also linear, so it will not change overall running time properties.

ACKNOWLEDGMENTS

This research was supported by a grant (GLOWA - Jordan River) from the German Bundesministerium fuer Bildung und Forschung and the Israeli Ministry of Science and Technology.

LITERATURE CITED

Aho, A.V., J.E. Hopcroft, and J.D. Ullman, 1974. The Design and Analysis of Computer Algorithms. Bell Telephone Laboratories, Inc., Addison-Wesley, Reading, Massachusetts.

Cole, T.M. and S.A. Wells, 2003. CE-QUAL-W2: A Two-Dimensional, Laterally Averaged, Hydrodynamic and Water Quality Model, Version 3.1. *Available at* http://www.cee.pdx.edu/w2. *Accessed on* June 17, 2004.

DHI Software, 2003. MIKE 11. Danish Hydraulic Institute (DHI) Water and Environment, Hørsholm, Denmark.

Doornkamp, J.C. and Cuchlaine A.M. King, 1971. Numerical Analysis in Geomorphology: An Introduction. Edward Arnold, London, United Kingdom.

Eagle Point Software Corporation, 2004. Eagle Point Watershed Modeling. Dubuque, Iowa.

Hornby, D., 2003. Create Strahler Stream Order for Braided River Polyline Themes. *Available at* http://arcscripts.esri.com/details.asp?dbid=11781. *Accessed on* June 17, 2004.

Horton, R.E., 1945. Erosional Development of Streams and Their Drainage Basins: Hydrophysical Approach to Quantitative Morphology. Bulletin of the Geological Society of America 56:275-370.

Lanfear, K.J., 1990. A Fast Algorithm for Automatically Computing Strahler Stream-Order. Water Resources Bulletin 26(6):977-981.

Lu M., T. Koike, and Norio Hayakawa, 1996. A Distributed Hydrological Modelling System Linking GIS and Hydrological Models. *In:* Application of Geographic Information Systems in Hydrology and Water Resources Management. Int. Assoc. Hydrological Sci., 1996, pp. 141-148, Wallingford, United Kingdom.

Scheidegger, A.E., 1968. Horton's Law of Stream Numbers. Water Resources Research 4:655-658.

Scheidegger, A.E., 1970. Theoretical Geomorphology (Second Revised Edition). Springer-Verlag, Berlin, Germany.

Sivapalan, M. and N.R. Viney, 1994. Large Scale Modeling to Predict the Effects of Land Use Changes. Water J. 21(1):33-37.

Strahler, A.N., 1957. Quantitative Analysis of Watershed Geomorphology. American Geophysical Union Transactions 38:913-920.

Tarboton, D. G., 2000. TARDEM: A Suite of Programs for the Analysis of Digital Elevation Data. Department of Engineering, Utah State University. *Available at* http://www.engineering.usu.edu/cee/faculty/dtarb/tardem.html. *Accessed on* June 17, 2004.

Tarboton, D. G., 2002. Terrain Analysis Using Digital Elevation Models (TauDEM). Department of Engineering, Utah State University. *Available at* http://moose.cee.usu.edu/taudem/taudem.html. *Accessed on* June 17, 2004.

Tarjan, R.E., 1972. Depth First Search and Linear Graph Algorithms. SIAM Journal of Computing 1:146-160.