Front. Earth Sci. 2019, 13(2): 317–326
https://doi.org/10.1007/s11707-018-0725-9

RESEARCH ARTICLE

A fast and simple algorithm for calculating flow accumulation matrices from raster digital elevation

Guiyun ZHOU (✉)[1,2], Hongqiang WEI[2], Suhua FU[3,4]

1 Center for Information Geoscience, University of Electronic Science and Technology of China, Chengdu 611731, China
2 School of Resources and Environment, University of Electronic Science and Technology of China, Chengdu 611731, China
3 State Key Laboratory of Soil Erosion and Dryland Farming on the Loess Plateau, Institute of Soil and Water Conservation, Chinese Academy of Sciences, Yangling 712100, China
4 Faculty of Geographical Science, Beijing Normal University, Beijing 100875, China

© Higher Education Press and Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract Calculating the flow accumulation matrix is an essential step for many hydrological and topographical analyses. This study gives an overview of the existing algorithms for flow accumulation calculations for single-flow direction matrices. A fast and simple algorithm for calculating flow accumulation matrices is proposed in this study. The algorithm identifies three types of cells in a flow direction matrix: source cells, intersection cells, and interior cells. It traverses all source cells and traces the downstream interior cells of each source cell until an intersection cell is encountered. An intersection cell is treated as an interior cell when its last drainage path is traced and the tracing continues with its downstream cells. Experiments are conducted on thirty datasets with a resolution of 3 m. Compared with the existing algorithms for flow accumulation calculation, the proposed algorithm is easy to implement, runs much faster than existing algorithms, and generally requires less memory space.

Keywords flow accumulation, flow direction, DEM, GIS

1 Introduction

The automatic extraction of drainage networks from raster digital elevation models (DEMs) is required in many scenarios such as soil erosion modeling, hydrological process simulation, and geomorphological analysis (Fu et al., 2011; Nobre et al., 2011; Yamazaki et al., 2012; Buchanan et al., 2014; Bai et al., 2015). A widely used method for extracting drainage networks from DEMs is based on the simulation of surface flow (Jenson and Domingue, 1988; Wang and Liu, 2006; Zhou et al., 2016). The method is composed of multiple steps which include removing depressions, assigning flow directions, and calculating the flow accumulation matrix. Among these procedures, calculating the flow accumulation matrix is an important step. The flow accumulation of a cell is equal to the number of cells that drain to it (O'Callaghan and Mark, 1984). Flow accumulation is an essential input for many hydrological and topographic analyses such as stream channel extraction, stream channel ordering, and sub-watershed delineation (Bai et al., 2015; Su et al., 2015; Barnes, 2017).

Some algorithms derive the flow accumulation matrix directly from a DEM (Arge et al., 2003; Bai et al., 2015). These algorithms generally require cells to be sorted based on their elevation values and have $O(N\log N)$ time complexity. They start from the highest cells and gradually move to lower cells. The algorithms encounter problems in flat areas, where flow directions cannot be determined based solely on the values of the neighboring cells. These algorithms access cells in an order based on their elevations and can result in random scattered data swapping between memory and the hard drive when they are applied to massive DEMs that do not fit in the main memory (Su et al., 2015).

It is more common to derive the flow accumulation matrix from a flow direction matrix rather than directly from a DEM. There are two methods for flow direction determination from a DEM: the single-flow direction method and the multiple-flow direction method. In the single-flow direction method, each cell only drains to one neighboring cell. The $D8$ method, which uses the direction of steepest descent as the flow direction of a cell, is the most widely adopted single-flow direction method (O'Callaghan and Mark, 1984; Garbrecht and Martz, 1997; Nardi et al., 2008; Barnes et al., 2014). In the multiple-flow

Received January 24, 2018; accepted July 8, 2018

E-mail: zhouguiyun@uestc.edu.cn

direction method, each cell can flow to more than one neighboring cell (Freeman, 1991; Quinn et al., 1991; Qin and Zhan, 2012). Because the *D*8 method is the most widely used method for determining flow direction, this study focuses on calculating the flow accumulation matrix from the flow direction matrix that is derived using the single-flow *D*8 method. The assignment of flow directions in depressions and flat areas in a DEM must be treated with special algorithms when the *D*8 method is used (Garbrecht and Martz, 1997; Wang and Liu, 2006; Barnes et al., 2014).

With the advent of airborne LiDAR (Light Detection and Ranging) technology, DEMs have become increasingly large. It is common for a DEM to contain billions of cells. The time required to calculate flow accumulation matrices of massive DEMs using conventional methods is becoming prohibitively long. In recent years, new algorithms have been proposed for calculating flow accumulation matrices. In this study, we propose a fast and simple algorithm to calculate the flow accumulation matrix. The remainder of the paper is organized as follows. Section 2 gives an overview of the algorithms for flow accumulation calculation from single-flow flow direction matrices. Our proposed algorithm is presented in Section 3. Section 4 presents the experimental results and compares the proposed algorithm with existing algorithms. Section 5 concludes the paper.

2 Overview of algorithms for flow accumulation calculation

In this section, we give an overview of the algorithms for calculating flow accumulation from single-flow flow direction matrices. All the algorithms produce the same flow accumulation matrix for the same input flow direction matrix. For convenience, Table 1 lists a group of symbols that are used in the pseudocode of the algorithms. Among the symbols, the symbol of NextCell(*c*) represents a function that returns a Boolean value and is used for tracing the immediate downstream cell of input cell *c*. If the input cell *c* drains towards the outside of the DEM or to a NODATA cell, the function returns a false value. Otherwise, the function returns a true value and cell *c* is updated to point to the downstream cell to which it drains.

2.1 NIDP-based algorithms

These types of algorithms are based on the concept of the number of input drainage paths (NIDP). The NIDP of a cell *c* is the number of neighboring cells that drain to *c*. Cells with an NIDP of zero are usually located on ridges, and cells with an NIDP greater than one are the intersection cells of more than one drainage path. The pseudocode for calculating the NIDP matrix from a flow direction matrix is shown in Algorithm 1 (Fig. 1).

The earliest version of the algorithm is proposed by O'Callaghan and Mark (1984). Their algorithm initializes the flow accumulation matrix with the value of one and starts from cells with an NIDP of zero. Suppose *c* is a cell with an NIDP value of zero. The flow accumulation of the immediate downstream cell *n* of *c* is increased by the accumulation value of *c*. The NIDP value of *n* is decreased by one. This iterative process stops when all cells have an NIDP value of zero. The number of iteration steps depends

Table 1 Symbols used in the pseudocodes

Symbol	Description
FlowDir	The input flow direction matrix
FlowAccu	The output flow accumulation matrix
NextCell(*c*)	A function returning a Boolean value. If the input cell *c* drains towards the outside of the DEM or it drains to a NODATA cell, the function returns a false value. Otherwise, the function returns a true value and cell *c* is updated as the downstream cell to which it drains
NIDP	The matrix giving the number of immediately adjacent cells that flow into each cell
c, *n*	Cells in matrices

```
1:   Initialize NIDP with the value of zero;
2:   for each cell c in FlowDir {
3:        if ( c is NODATA cell) {
4:             NIDP(c)=NODATA;
5:             continue;
6:        }
7:        n=c;
8:        if (!NextCell(n)) continue;
9:        NIDP (n)++;
10:  }
```

Fig. 1 Algorithm 1: compute the NIDP matrix from FlowDir matrix.

on the length of the longest drainage path. In the worst case, the time complexity is $O(N^2)$, where N is the number of cells. On average, the length of the longest drainage path is expected to be the magnitude of $N^{0.5}$ and the average time complexity of the algorithm is $O(N^{1.5})$. Their algorithm is implemented in ArcGIS™ hydrology toolset (Choi, 2012) and is widely used (Ortega and Rueda, 2010).

Yao and Shi (2015) proposed an alternating scanning scheme for this algorithm, which reduces the time complexity of the algorithm to $O(N\log N)$. Wang et al. (2011) proposed an improved version of this algorithm. Their algorithm, referred to as Wang's algorithm in this study, uses a plain queue to record the starting cell in each iteration step. Initially, Wang's algorithm pushes all cells with NIDP values of zero into the queue. When a cell c is popped off the queue, the accumulation value of its immediate downstream cell n is increased by the accumulation value of c and the NIDP of n is decreased by one. If the NIDP of n becomes zero, n is pushed into the queue. The algorithm stops when the queue becomes empty. Wang's algorithm has a time complexity of $O(N)$. The pseudocode of Wang's algorithm is shown in Algorithm 2 (Fig. 2).

Jiang et al. (2013) proposed another improved version of the original algorithm by O'Callaghan and Mark (1984). The improved algorithm, referred to as Jiang's algorithm in this study, does not require the creation of an NIDP matrix by using the flow accumulation matrix to store the NIDP information. A cell whose NIDP value is m is assigned the value of $-1-m$ in the flow accumulation matrix. A cell whose value is -1 in the flow accumulation matrix is a cell without any input drainage path and is pushed into a stack. When a cell is popped off the stack, the accumulation values of its upstream neighbors have been computed and its accumulation value can be computed by iterating over all of its neighboring cells that drain to it. Jiang's algorithm does not require the creation of an NIDP matrix but the accumulation value of each cell needs to be calculated by iteration over all of its neighboring cells. Jiang's algorithm has a time complexity of $O(N)$. The pseudocode of Jiang's algorithm can be found in Jiang et al. (2013).

2.2 Traversal algorithm

This algorithm traverses each cell within a flow direction matrix row by row and column by column. When a cell c is traversed, the accumulation values of all downstream cells of c are increased by one. The time complexity of the algorithm depends on the length of the longest drainage path. As discussed in Section 2.1, the length of the longest drainage path is expected to be the magnitude of $N^{0.5}$ and the average time complexity of the method is $O(N^{1.5})$.

2.3 BTI-based algorithm

Su et al. (2015) propose an algorithm to use basin tree indices (BTI) to guide the calculation of the flow accumulation matrix. Their algorithm starts from the outlet cells that drain to the outside of the DEM and builds basin trees by tracing the drainage paths from the outlet cells to the source cells of each basin. The flow accumulation matrix is then calculated by tracing the trees from the leaves to the roots. The time complexity of the algorithm is $O(N)$. The pseudocode of this algorithm is shown in Algorithm 3 (Fig. 3). To avoid the repeated reallocation of the arrays for storing the trees, an array of size N is pre-allocated for storing all of the basin trees in our implementation of the algorithm in Section 4.

2.4 Recursive algorithm

This algorithm computes the accumulation value of a cell c by recursively computing the accumulation values of all of its neighboring cells that drain to it (Freeman, 1991; Choi, 2012). This is the first algorithm for flow accumulation computation that has a time complexity of $O(N)$. The pseudocode of this algorithm is shown in Algorithm 4 (Fig. 4). Usually, the program that implements the

```
1:  Initialize FlowAccu with the value of one;
2:  Let Q be an empty queue;
3:  Compute NIDP from FlowDir using Algorithm 1;
4:  for each cell c in NIDP {
5:      if (NIDP(c)==0) push c into Q;
6:  }
7:  while (Q is not empty) {
8:      pop cell c from Q;
9:      n=c;
10:     if (!NextCell(c)) continue;
11:     FlowAccu(c)+=FlowAccu(n);
12:     NIDP (c)--;
13:     if (NIDP (c)==0) push c into Q;
14: }
```

Fig. 2 Algorithm 2: compute the FlowAccu matrix from FlowDir matrix using Wang's algorithm.

```
1:  Initialize FlowAccu with the value of one;
2:  Let Outlet be an array of cells;
3:  //Find all outlet cells
4:  for each cell c in FlowDir {
5:      if (!NextCell(c)) push c into Outlet;
6:  }
7:  Let BTI be an array of arrays of the size of Outlet. Each element of BTI is
8:      one empty basin tree;
9:  //Build the basin tree
10:  for (i=0 to Outlet.length) {
11:      c =Outlet[i];
12:      push c into BTI[i];
13:      j=0;
14:      while (j<BTI[i].length)
15:          c= BTI[i][j];
16:          for (each neighboring cell n of c) {
17:              if (n flows to c) push n into BTI[i];
18:          }
19:          j++;
20:      }
21:  }
22:  //calculate flow accumulation
23:  for (i=0 to Outlet.length) {
24:      for (j=BTI[i].length-1; j>=0; j--) {
25:          n=c= BTI[i][j];
26:          if (!NextCell(c)) continue;
27:          FlowAccu(c)+=FlowAccu(n);
28:      }
29:  }
```

Fig. 3 Algorithm 3: compute the FlowAccu matrix from the FlowDir matrix using the BTI-based algorithm.

```
1:  Initialize FlowAccu with the value of zero;
2:  for each cell c in FlowAccu {
3:      //call the recursive function to calculate accumulation value
4:      ComputeAccu(FlowAccu, FlowDir, c);
5:  }
6:  //recursive function for calculating flow accumulation
7:  Function int ComputeAccu (FlowAccu, FlowDir, c)
8:      if (FlowAccu(c) <1 ) {
9:          FlowAccu(c)=1;
10:         for each neighboring cell n that drains to c {
11:             FlowAccu(c)+= ComputeAccu(FlowAccu, FlowDir, n);
12:         }
13:     }
14:     return FlowAccu(c);
15:  End Function
```

Fig. 4 Algorithm 4: compute the FlowAccu matrix from FlowDir matrix using the recursive algorithm.

recursive algorithm needs to determine the maximum size for the call stack during compilation and this size must be sufficiently large to process input data, which is a challenging issue to deal with when the size of the input data is unknown.

2.5 Iterative scanning algorithm

Zhang et al. (2013) computed the flow accumulation matrix using an iterative procedure. Each iteration step includes a forward and a reverse traversal of the accumulation matrix. During each traversal, the accumulation value of a cell is compared with the sum of the accumulation values of all of its neighboring cells that drain to it, and the accumulation value of the cell is updated as the sum if the sum is greater than the accumulation value of the cell. The iteration stops when there are no changes in the accumulation values of all cells. Each iteration step has

a time complexity of $O(N)$. The number of iterations depends on the length of the longest drainage path. In the worst case, the time complexity is $O(N^2)$.

3 A fast and simple algorithm for flow accumulation calculation

In this section, we present a new algorithm for calculating flow accumulation from single-flow flow direction matrices. We compare the time complexity and memory requirement of our algorithm with those of the four existing algorithms that also have $O(N)$ time complexity.

3.1 The proposed algorithm

The four existing algorithms, including Wang's algorithm, Jiang's algorithm, the BTI-based algorithm, and the recursive algorithm, have $O(N)$ time complexity. Other algorithms for flow accumulation calculation have higher time complexity, run substantially slower, and are not considered further in this study. In this section, we propose our algorithm, which also has $O(N)$ time complexity. Compared with the above algorithms, the proposed algorithm has a smaller constant coefficient before the time complexity, allowing for faster computation.

In our algorithm, we define three types of cells within the flow direction matrix: source cells, interior cells, and intersection cells. A source cell does not have any neighboring cells that drain to it and its NIDP value is zero. An interior cell has only one neighboring cell that drains to it and its NIDP value is one. An intersection cell has more than one neighboring cell that drains to it and its NIDP value is greater than one.

The proposed algorithm initializes the flow accumulation matrix with the value of one. It first calculates the NIDP matrix from the flow direction matrix. The algorithm then traverses each cell within the flow direction matrix row by row and column by column, similar to the traversal algorithm. When a source cell c is encountered, the algorithm traces all downstream cells of c until it encounters an intersection cell i. During the tracing, the accumulation value of a given cell is added to the accumulation value of its immediate downstream cell. An interior cell has only one neighboring cell that drains to it and its final accumulation value is obtained when the tracing is done. The accumulation value of the intersection cell i is updated from this drainage path. However, cell i has other unvisited neighboring cells that drain to it and its final accumulation value cannot be obtained after the first round of tracing. The algorithm decreases the NIDP value of i by one. Cell i is visited again when other drainage paths that pass through it are traced. Once all of the drainage paths that pass through it are traced, cell i is treated as an interior cell and the final accumulation value of i is obtained correctly, and the last tracing process can continue the tracing after cell i is treated as an interior cell. The pseudocode of the proposed algorithm is shown in Algorithm 5 (Fig. 5).

A worked example of the proposed algorithm is given in Fig. 6. A synthetic DEM with a size of 3 rows by 4 columns, including its flow direction matrix, is shown in Fig. 6(a). The algorithm first computes the NIDP matrix (Fig. 6(b)) and initializes the flow accumulation matrix with 1 (Fig. 6(c)). The algorithm then traverses the NIDP matrix from top to bottom and from left to right. It encounters the first source cell H. The algorithm starts the first round of tracing from H and all downstream cells of H including D, C, and F, are traced (Fig. 6(d)). The accumulation value of each traced cell is increased by the accumulation of its immediate upstream cell. Because F is an intersection cell, the tracing stops and the NIDP value of F is decreased by 1. F is treated as an interior cell hereafter. The algorithm keeps traversing the remaining cells in the NIDP matrix and encounters the second source cell J. The algorithm starts the second tracing from J and all downstream cells including I, E, and A are traced (Fig. 6(e)). After the second tracing is done, the NIDP

```
1:  Initialize FlowAccu with the value of one;
2:  Compute NIDP from FlowDir using Algorithm 1;
3:  Let nAccu be an interger ;
4:  for each cell c in FlowDir {
5:      if (NIDP (c)!=0) continue;
6:      n=c;
7:      nAccu=0;
8:      do {
9:          FlowAccu(n)+= nAccu;
10:         nAccu= FlowAccu(n);
11:         if (NIDP (n)>=2) {
12:             NIDP (n)--;
13:             break;
14:         }
15:     } while (NextCell(n))
16: }
```

Fig. 5 Algorithm 5: compute the FlowAccu matrix from the FlowDir matrix using the proposed algorithm.

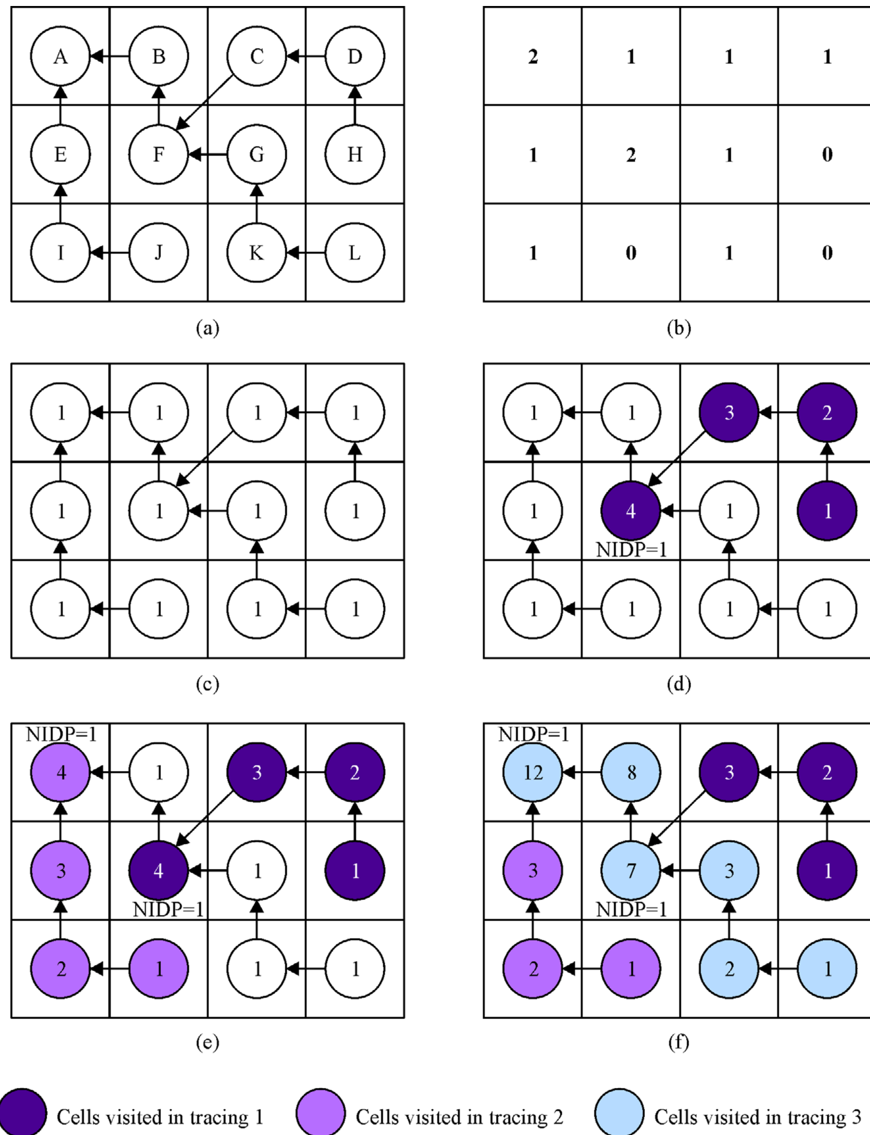

Fig. 6 A worked example of the proposed algorithm. (a) A 3×4 DEM with flow directions. (b) Initial NIDP matrix. (c) The flow accumulation matrix is initialized with one. (d) Cells H, D, C, and F are processed during the first round of tracing. The NIDP value of F is decreased by 1 and F is treated as an interior cell hereafter. (e) Cells J, I, E, and A are processed during the second round of tracing. The NIDP value of A is decreased by 1 and A is treated as an interior cell hereafter. (f) Cells L, K, G, F, B, and A are processed during the third round of tracing. The flow accumulation values of all cells are calculated after the tracing.

value of A is decreased by 1 and A is treated as an interior cell hereafter. In Fig. 6(f), the third tracing starts from cell L and all downstream cells including K, G, F, B, and A are traced. Note that cell F is being treated as an interior cell and that the third tracing does not stop when F is encountered. After the third tracing is done, no source cells remain and the traversing process is complete. The final flow accumulation matrix is obtained and shown in Fig. 6(f).

3.2 Time complexity analysis

Similar to the traversal algorithm, our algorithm utilizes the NIDP matrix. Compared with the four other algorithms that have a time complexity of $O(N)$, our algorithm has a smaller constant coefficient before the time complexity. Our algorithm visits source and interior cells only once, and intersection cells are visited the same number of times as their initial NIDP values. Wang's algorithm visits the cells the same number of times as our algorithm does but it requires a queue to hold the source cells. The manipulation of the queue incurs a performance loss. Jiang's algorithm requires the manipulation of a stack and calculating the accumulation value of each cell needs to access all of its neighboring cells. The BTI-based algorithm requires two passes to process the cells and each cell is visited at least

twice. The recursive algorithm is a process of depth-first search for the basin trees. It is a two-pass process. The first pass traces the drainage path from a starting cell to the leaves and the second pass computes the accumulation value from the leaves to the starting cell. In this regard, the recursive algorithm is similar to the BTI-based algorithm. Unlike the BTI-based algorithm, however, the recursive algorithm does not build the basin trees explicitly. The running time of a recursive algorithm is highly dependent on the optimization applied to it by the compiler. Different compilers may generate machine codes with considerable differences in running times.

3.3 Memory requirement analysis

All algorithms require an input flow direction matrix and an output flow accumulation matrix. Suppose that a DEM has N cells in it. Our algorithm requires an NIDP matrix. Because the NIDP value varies between 0 and 8, the NIDP matrix requires N bytes. Wang's algorithm also requires an NIDP matrix. In addition, it requires a queue Q to hold source cells. Each cell that is pushed into Q has its row index and column index information, which requires 8 bytes for storage in the most general cases, where the number of rows and columns may exceed 70,000 and a 4-byte unsigned integer is needed to store the row index and column index separately. For our test DEMs in Section 4, about 33.71% of the total number of the cells averaged across all the test DEMs are source cells. On average, the initial space required by the queue is more than 2.5N bytes. Jiang's algorithm does not use the NIDP matrix. Instead it uses a stack and requires at least 2.5N bytes as well. The BTI-based algorithm requires additional memory space to store the basin trees. Similar to Wang's algorithm, each cell needs 8 bytes for its row index and column index. The additional total memory space required by the BTI-based algorithm is at least 8N bytes. The recursive algorithm requires the call stack to track the calls. The memory requirement of a recursive algorithm depends on many factors, including the compiler and the platform on which the algorithm runs, and it is difficult to make a quantitative estimate of its memory requirement.

4 Experimental results

The five flow accumulation algorithms with $O(N)$ time complexity, including Wang's algorithm, Jiang's algorithm, the BTI-based algorithm, the recursive algorithm, and our proposed algorithm, are implemented in C++. The source code is available for download on GitHub. The 3-m LiDAR-based DEMs of thirty counties in the state of Minnesota, USA are downloaded from the FTP site operated by the Minnesota Geospatial Information Office. The first 30 counties in Minnesota in alphabetic order are chosen for the experiments to avoid selection bias. The dataset is freely available for download by any user. On average, each county contains approximately 3.96×10^8 cells in the 3-m DEM. Because of the large number of cells to be processed, we do not use ArcGIS™ to derive the flow direction matrices. Instead, we use the algorithm proposed by Wang and Liu (2006) to fill the depressions and derive the flow direction matrices for all tested counties. The algorithms are tested on both Linux and Windows operating systems. The Linux system is a CentOS 6.5 operating system with an Intel Xeon E5-2403 1.80 GHz processor. All algorithms running on the Linux system are compiled using GCC 4.8.3 with O3 optimization. The Windows system is a 64-bit Windows 7 operating system with an Intel Xeon E5-2620 2.0 GHz processor and 56 GB RAM. All algorithms are compiled using Microsoft Visual Studio 2012 with the default optimization settings, such as Maximizing speed and Streaming SIMD Extensions.

All five algorithms produce the same flow accumulation matrix for each tested DEM. Each flow direction matrix is completely loaded into the main memory for processing and the loading time is excluded from the total running time. Each algorithm is run multiple times and the average running time is used for analysis. Figures 7 and 8 plot the running times per 100 million cells of the five algorithms on Linux and Windows systems, respectively. On the Linux system, the average running times per 100 million cells are 14.36 seconds for Wang's algorithm, 31.37 seconds for Jiang's algorithm, 40.65 seconds for the BTI-based algorithm, 31.18 seconds for the recursive algorithm, and 10.40 seconds for our proposed algorithm. On the Windows system, the average running times per 100 million cells are 14.42 seconds for Wang's algorithm, 15.90 seconds for Jiang's algorithm, 18.95 seconds for the BTI-based algorithm, 10.87 seconds for the recursive algorithm, and 5.26 seconds for our proposed algorithm. There are three points that are worth noting about the running times. First, on both systems, our algorithm runs the fastest for all tested DEMs. The speed-up ratios of our proposed algorithm over the second fastest algorithm is about 28% and 51% on the Linux and Windows systems, respectively. Second, the relative rankings of the five algorithms are different on the two systems. For example, on average, the recursive algorithm runs the second fastest in our experiment on the Windows system, whereas it runs the third fastest (almost as fast as the Jiang's algorithm) on the Linux system. In addition, on both systems, Wang's algorithm is faster than the BTI-based algorithm. This finding is different from the findings of Su et al. (2015), who report that the BTI-based algorithm is much faster than Wang's algorithm, which is called the improved flow number matrix algorithm in their study. The difference in the running times may be caused partly by different details of the implementations. For example, Wang et al. (2011) originally use two collection structures for storing the

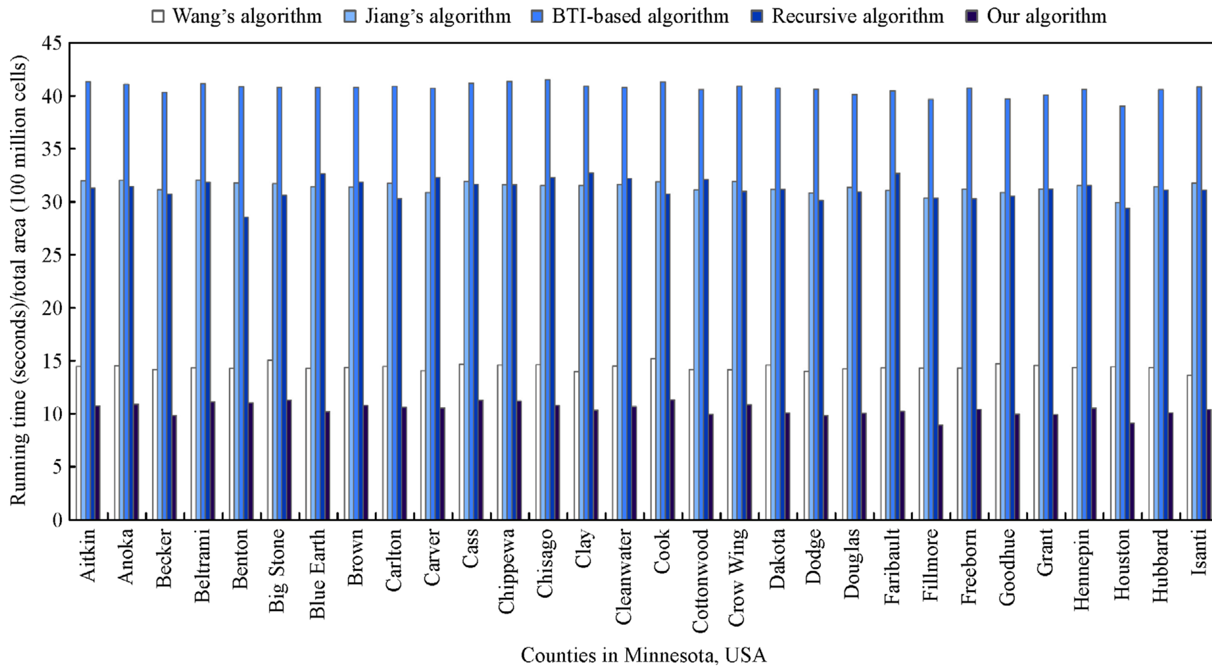

Fig. 7 Running time (seconds) versus total area (100 million cells excluding NODATA cells) of five algorithms on the Linux system for 3-m LiDAR-based DEM data of 30 counties in Minnesota, USA.

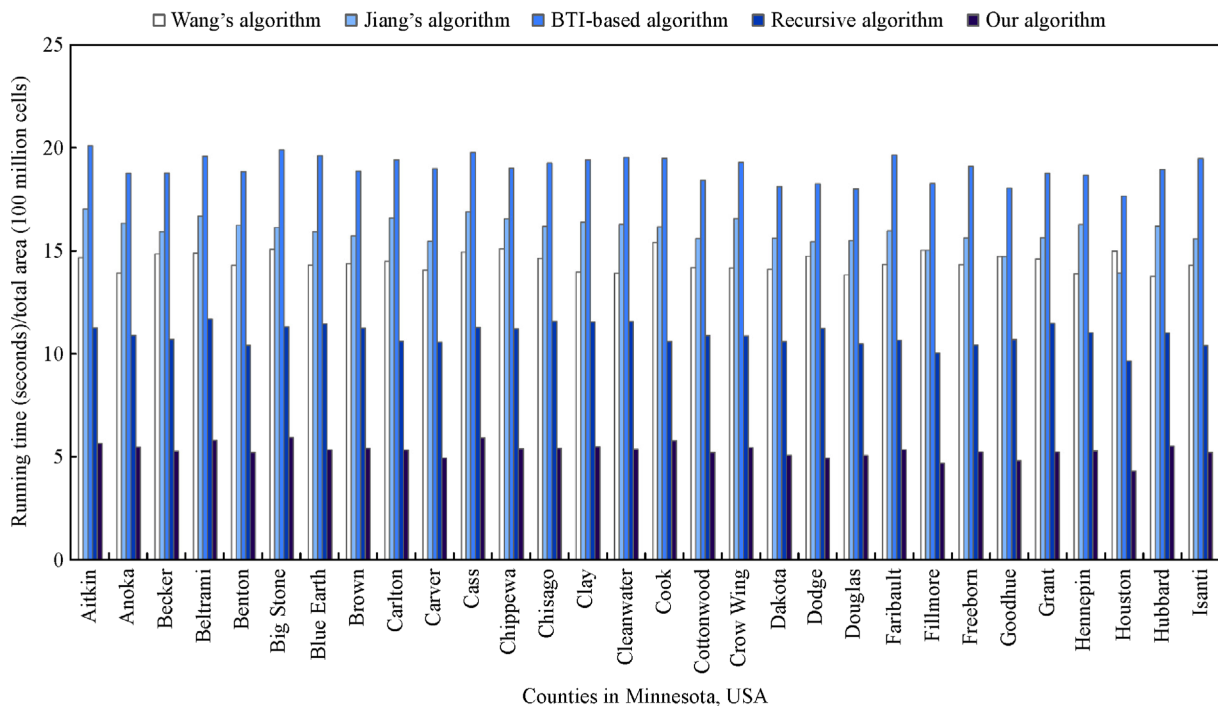

Fig. 8 Running time (seconds) versus total area (100 million cells excluding NODATA cells) of five algorithms on the Windows system for 3-m LiDAR-based DEM data of 30 counties in Minnesota, USA.

source cells in two consecutive iteration steps and copy elements from one collection structure to the other structure. For the BTI-based algorithm, we pre-allocate a matrix of the same size as the input DEM to avoid the repeated reallocation of the dynamic arrays for storing the basin trees. Third, in terms of the absolute amount of running times, with the exception of the Wang's algorithm, other four algorithms generally run slower on the Linux

system than on the Windows system. This may partially be attributed to the fact that the main frequency of the processor on which the Linux system runs is lower than that of the processor on which the Windows system runs.

It is clear that the running times of the algorithms for calculating flow accumulations are subject to such settings as the hardware configurations, operating systems, compilation optimization options, and implementation details. Because our proposed algorithm has a smaller constant coefficient before the time complexity of $O(N)$ than other four algorithms, our proposed algorithm is expected to run the fastest as long as similar settings are adopted for all algorithms.

5 Conclusions

In this study, we provide an overview of existing algorithms for flow accumulation calculations from single-flow direction matrices and propose a fast and simple algorithm for calculating flow accumulation matrices. All algorithms for flow accumulation calculations that have $O(N)$ time complexity are implemented in C ++. Experiments are conducted on thirty LiDAR-based DEMs with a resolution of 3 m.

Compared with the four existing algorithms for flow accumulation calculations that have $O(N)$ time complexity, our algorithm runs substantially faster, requires less space than all non-recursive algorithms, does not require a collection structure, and is easy to understand and implement.

Our proposed algorithm is only applicable to single-flow flow direction matrices. In future work, we will adapt the algorithm to make it applicable to multiple-flow flow direction matrices.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant No. 41671427) and the Fundamental Research Funds for the Central Universities (ZYGX2016J148). We thank the anonymous referees for their constructive criticism and comments.

References

Arge L, Chase J, Halpin P, Toma L, Vitter J, Urban D, Wickremesinghe R (2003). Efficient flow computation on massive grid terrain datasets. GeoInformatica, 7(4): 283–313

Bai R, Li T, Huang Y, Li J, Wang G (2015). An efficient and comprehensive method for drainage network extraction from DEM with billions of pixels using a size-balanced binary search tree. Geomorphology, 238: 56–67

Barnes R (2017). Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters. Environ Model Softw, 92: 202–212

Barnes R, Lehman C, Mulla D (2014). An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. Comput Geosci, 62: 128–135

Buchanan B P, Nagle G N, Walter M T (2014). Long-term monitoring and assessment of a stream restoration project in central New York. River Res Appl, 30(2): 245–258

Choi Y (2012). A new algorithm to calculate weighted flow-accumulation from a DEM by considering surface and underground stormwater infrastructure. Environ Model Softw, 30: 81–91

Freeman T G (1991). Calculating catchment area with divergent flow based on a regular grid. Comput Geosci, 17(3): 413–422

Fu S, Liu B, Liu H, Xu L (2011). The effect of slope on interrill erosion at short slopes. Catena, 84(1–2): 29–34

Garbrecht J, Martz L W (1997). The assignment of drainage direction over flat surfaces in raster digital elevation models. J Hydrol (Amst), 193(1–4): 204–213

Jenson S K, Domingue J O (1988). Extracting topographic structure from digital elevation data for geographic information system analysis. Photogramm Eng Remote Sensing, 54(11): 1593–1600

Jiang L, Tang G, Liu X, Song X, Yang J, Liu K (2013). Parallel contributing area calculation with granularity control on massive grid terrain datasets. Comput Geosci, 60: 70–80

Nardi F, Grimaldi S, Santini M, Petroselli A, Ubertini L (2008). Hydrogeomorphic properties of simulated drainage patterns using digital elevation models: the flat area issue. Hydrol Sci J, 53(6): 1176–1193

Nobre A D, Cuartas L A, Hodnett M, Rennó C D, Rodrigues G, Silveira A, Waterloo M, Saleska S (2011). Height above the nearest drainage — a hydrologically relevant new terrain model. J Hydrol (Amst), 404 (1–2): 13–29

O'Callaghan J F, Mark D M (1984). The extraction of drainage networks from digital elevation data. Comput Vis Graph Image Process, 28(3): 323–344

Ortega L, Rueda A (2010). Parallel drainage network computation on CUDA. Comput Geosci, 36(2): 171–178

Qin C Z, Zhan L (2012). Parallelizing flow-accumulation calculations on graphics processing units—from iterative DEM preprocessing algorithm to recursive multiple-flow-direction algorithm. Comput Geosci, 43(0): 7–16

Quinn P, Beven K, Chevallier P, Planchon O (1991). The prediction of hillslope flow paths for distributed hydrological modelling using digital terrain models. Hydrol Processes, 5(1): 59–79

Su C, Yu W, Feng C, Yu C, Huang Z, Zhang X (2015). An efficient algorithm for calculating drainage accumulation in digital elevation models based on the basin tree index. IEEE Geoscience and Remote Sensing Letters, 12(2): 424–428

Wang L, Liu H (2006). An efficient method for identifying and filling surface depressions in digital elevation models for hydrologic analysis and modelling. Int J Geogr Inf Sci, 20(2): 193–213

Wang Y, Liu Y, Xie H, Xiang Z (2011). A quick algorithm of counting flow accumulation matrix for deriving drainage networks from a DEM. In: Proceedings on the Third International Conference on Digital Image Processing

Yamazaki D, Baugh C A, Bates P D, Kanae S, Alsdorf D E, Oki T (2012). Adjustment of a spaceborne DEM for use in floodplain hydrodynamic modeling. J Hydrol (Amst), 436–437: 81–91

Yao Y, Shi X (2015). Alternating scanning orders and combining

algorithms to improve the efficiency of flow accumulation calculation. Int J Geogr Inf Sci, 29(7): 1214–1239

Zhang H, Yang Q, Li R, Liu Q, Moore D, He P, Ritsema C J, Geissen V (2013). Extension of a GIS procedure for calculating the RUSLE equation LS factor. Comput Geosci, 52: 177–188

Zhou G, Sun Z, Fu S (2016). An efficient variant of the priority-flood algorithm for filling depressions in raster digital elevation models. Comput Geosci, 90: 87–96