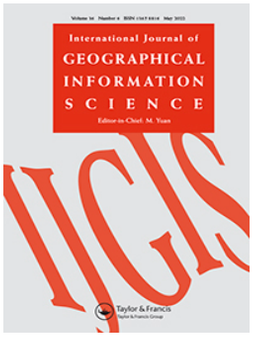


International Journal of Geographical Information Science

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tgis20

An efficient multiple scanning order algorithm for accumulative least-cost surface calculation

Yuanzhi Yao, Xun Shi & Zekun Wang

To cite this article: Yuanzhi Yao, Xun Shi & Zekun Wang (2022): An efficient multiple scanning order algorithm for accumulative least-cost surface calculation, International Journal of Geographical Information Science, DOI: 10.1080/13658816.2022.2052885

To link to this article: https://doi.org/10.1080/13658816.2022.2052885

INTERNATIONAL JOURNAL OF GEOGRAPHICAL INFORMATION SCIENCE
https://doi.org/10.1080/13658816.2022.2052885

Taylor & Francis
Taylor & Francis Group

RESEARCH ARTICLE

An efficient multiple scanning order algorithm for accumulative least-cost surface calculation

Yuanzhi Yao[a,b], Xun Shi[c] and Zekun Wang[d]

[a]School of Geographic Sciences, East China Normal University, Shanghai, P.R. China; [b]College of Forestry and Wildlife Sciences, Auburn University, Auburn, AL, USA; [c]Department of Geography, Dartmouth College, Hanover, NH, USA; [d]Department of Mechanical Engineering, Auburn University, Auburn, AL, USA

ABSTRACT

The least-cost surface (LCS) calculation is a compute-intensive problem conventionally solved by the queue-based Dijkstra's algorithm. Alternative raster-based scanning algorithms have also been proposed which use a moving window to scan the whole study area iteratively. Here we propose improvements to the raster-based algorithms. The main improvement is to implement multiple scanning orders (MSO) to replace the conventional single scanning order (SSO, typically from upper-left corner to lower-right corner, row by row). We compared the performance of different algorithms over different cost surfaces and with different numbers of source points. The comparison shows that a raster-based algorithm adopting MSO has a substantially better performance than a conventional raster-based algorithm using SSO. An MSO raster-based algorithm is generally comparable to the queue-based Dijkstra's algorithm, and surpasses the latter over a relatively simple cost surface (e.g. in which the cost is resampled) and/or when the number of source points is relatively large. Our empirical experiments suggest that MSO reduces the time complexity from to $\Theta(N^2)$ to $\Theta(\mathrm{NlogN})$. Additionally, we found that the MSO raster-based algorithm can be easily parallelized using shared-memory parallel programming.

ARTICLE HISTORY
Received 20 June 2021
Accepted 10 March 2022

KEYWORDS
Least-cost surface; geocomputation; spatial analysis

1. Introduction

A least accumulative cost passage surface, herein referred to as a *least-cost surface* (LCS), is a raster representation of the minimum travel cost of each and every location in an area to a certain destination location or locations (Douglas 1994). LCS is the basis for the raster-based shortest path analysis, and the latter has many applications in wildlife ecology (LaRue and Nielsen 2008, Gonçalves 2010), healthcare access studies (Brabyn and Skelly 2002, Elsheikh and Hassan 2016), and urban and regional planning (e.g. route selections for roads and pipelines) (Teng *et al*. 2011, Balbi *et al*. 2019). The process of deriving LCS is highly computationally intensive. There is a clear need for

CONTACT Xun Shi xun.shi@dartmouth.edu

efficient LCS algorithms given the increasing availability of high-resolution remotely sensed and topographic data.

A queue-based Dijkstra's algorithm (Xu and Lathrop Jr 1994, 1995, Soltani *et al*. 2002) has been widely accepted as the standard solution to the LCS problems. For example, it is the solution adopted by the *cost distance* tool in ArcGIS (Scott and Janikas 2010) and the Cumulative Cost Mapping tool in Google Earth (Gorelick *et al*. 2017). This algorithm takes a source raster and a cost raster as inputs, creates and maintains a queue for the candidate cells, and outputs a result raster. The source raster contains only labels of those cells designated as sources (thus it is not required to be a raster), and the cost raster holds the local cost of traversing the cell. The process includes three basic steps. First, all source cells, i.e. those cells labeled as sources in the source raster, are added to the candidate queue. In step 2, the algorithm finds all immediate neighboring cells of the first cell in the candidate queue, calculates their cumulative least costs to that first cell, and puts them into the candidate queue in the order from the smallest cumulative cost value to the largest. In step 3, it records the cumulative cost of that processed first cell in the candidate queue into the output raster (the cumulative cost for a source cell is 0), and removes it from the queue (i.e. this is a cell whose final least cost has been determined). The algorithm repeats steps 2 and 3, until the queue is empty, i.e. the least cost value of all cells have been determined and the LCS is generated. While Fredman and Tarjan (1987) have tried to optimize Dijkstra's algorithm for LCS, a fundamental issue they did not address was that the use of a sorted queue hampers the parallelization of the process. Basically, with Dijkstra's algorithm there is no static strategy to decompose the study domain. This is because the sorted queue in the algorithm needs to be constantly updated using the global data, and therefore even though one can divide the grid cells stored in the queue into sections and send them into multiple workflows, there will be intensive data exchanges between the main workflow and sub-workflows, resulting in low efficiency, especially when dealing with large data sets. A few previous studies attempting to parallelize the minimum searching process of the Dijkstra's algorithm achieved less than 10% speed improvement on average (Crauser *et al*. 1998, Jasika *et al*. 2012).

Alternative from a queue-based procedure, Collischonn and Pilar (2000) propose the use of raster to store information during the calculation. Their algorithm scans the entire study area to find recently updated cells, and updates their neighboring cells' accumulative least cost values. The scanning is repeated until no cell needs to be updated. The label to mark a cell that was updated during the last iteration is stored in a separate raster that has the same dimensions as those of the original cost raster. This label raster plays the role of the queues in aforementioned Dijkstra's algorithm. It effectively works in the way of a hash table, which saves the effort of queue operations, but potentially requires more memory. Besides the distinction between queue and raster, a more important difference between Dijkstra's algorithm and the raster-based algorithm is that the latter updates not only the neighbor of the cell that has the minimum cost value, but also the cells affected by the newly updated cells. In this way, the raster-based algorithm saves the effort of searching for the minimum-value cell in each iteration, but at a cost of calculating multiple intermediate values for most

cells. Essentially, the tradeoff is about intensive queue operations versus redundant updating operations.

In this study, we implemented an improved raster-based algorithm for the LCS calculation. The improvement to the raster-based algorithm is twofold. First, we implemented multiple scanning orders (MSO) for the raster-based method. A conventional raster scanning method employs a single scanning order (SSO), typically starting from the upper-left corner of the raster and ending at the lower-right corner, scanning the raster row by row. Yao and his colleagues (Yao *et al*. 2012, Yao and Shi 2015) found that for a directional raster operation, for example flow accumulation, applying multiple different scanning orders can considerably improve the calculation efficiency. LCS calculation is directional, and thus a process employing MSO should be more efficient than one using SSO. Second, we found that a restriction in Collischonn and Pilar (2000) algorithm limits the updating operation to the neighbors of only those cells updated in last iteration. Including neighbors of those cells updated in current iteration as well will not affect the final result, but will improve efficiency. Lastly, we tried to take the most important potential advantage of the raster method, i.e. parallelization. We parallelized the MSO algorithm based on shared-memory parallel programming (OpenMP API). We tested the performance of different algorithms over two cost surfaces of different types and with different numbers of source points, intending to compare them under different situations.

2. Improvements to the raster-based algorithm for least cost surface calculation

The raster-based algorithm for calculating a LCS proposed by Collischonn and Pilar (2000) scans the entire study area to find cells whose accumulative cost values have been updated in the previous iteration (the first iteration starts with those specified *source* cells), and calculates the accumulated least cost value for their eight neighboring cells in the 3×3 window. If the newly calculated value of a cell is smaller than the cell's old value, the cell will be updated with the new value and will be tagged in a separate raster for the next iteration. As noted in the Introduction, we implemented two improvements to this process, for which details are given here.

2.1. Implementing multiple scanning orders

Several studies have implemented multiple scanning orders (MSO), and suggested that MSO can improve the efficiency (Vincent 1993, Planchon and Darboux 2002). Yao and Shi (2015) developed an MSO algorithm for calculating flow accumulation from a DEM and demonstrated that it is much faster than an SSO process. They illustrated why MSO would be more efficient than SSO for a directional raster operation like the flow accumulation calculation. LCS calculation is also directional, and thus should also be able to take advantage of MSO. Following Yao and Shi (2015), we illustrate why MSO is more efficient than SSO in calculating LSC in Figure 1.

Figure 1a shows a simplistic scenario where a least cost path extends from north to south. This extending direction happens to exactly match the conventional scanning order (from upper-left to lower-right), and thus it takes only one scan of a conventional

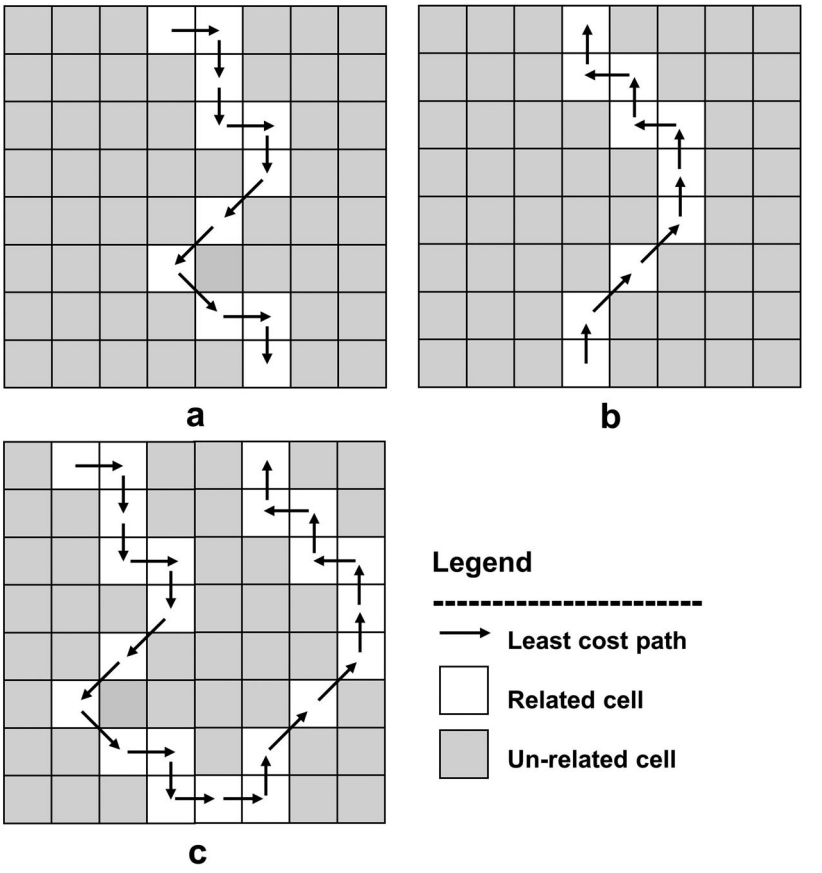


Figure 1. Illustrative least-cost paths for demonstrating how multiple scanning orders can improve efficiency of the least-cost surface calculation. (a) A least-cost path that can be processed by one scan iteration of the conventional upper-left-by-row (UL-row) scan order. (b) A least-cost path that can be processed by one scan iteration of a lower-right-by-row (LR-row) scan order. (c) A complex least-cost path that requires many iterations if use either of the scan orders in a and b, but needs only two iterations if first use the one in a, followed by the one in b.

operation to derive the path. On the other hand, the scenario illustrated by Figure 1b, in which the path is from south to north, would take the conventional scanning order many iterations to complete. This is because each such scan can only move one cell ahead along the path in one scanning iteration. However, a scan running from the lower-right corner to the upper-left corner would require only one iteration to complete. Neither order can handle the scenario in Figure 1c efficiently, but if both are run alternately it would take only two scans to derive the path. These simple illustrations show that if the scan happens to match the specific moving direction along the targeted path, the efficiency of the process will be high. In reality, the direction of a path can be complicated, and a strategy of achieving overall optimization is to apply scans of different orders. Yao and Shi (2015) listed eight basic scanning orders and the matched directions. Here, we adopt their naming convention. For example, the conventional order that runs from the

upper-left (*UL*) corner to the lower-right (*LR*) corner, row by row, is referred to as the *UL-row* order in the remainder of this paper.

2.2. Working on all recently updated cells

The algorithm proposed by Collischonn and Pilar (2000) works only on those cells updated in the immediately previous scanning iteration, and skips cells updated in current iteration. Instead, we let the operation update cells that have been processed in current iteration. Rosenfeld and Pfaltz (1966) have proved the two options to be mathematically equivalent, but the latter (the option we chose) is more optimal in terms of processing time. This is because those cells updated in current iteration will be the target of the immediate next neighbour-updating operation anyway, and it does not matter if they are processed in current iteration or are saved for the next.

The *sequential operation* increases the number of cells processed in one iteration, and thus is likely to reduce the total number of scanning iterations and improve efficiency of the entire calculation (Figure 2).

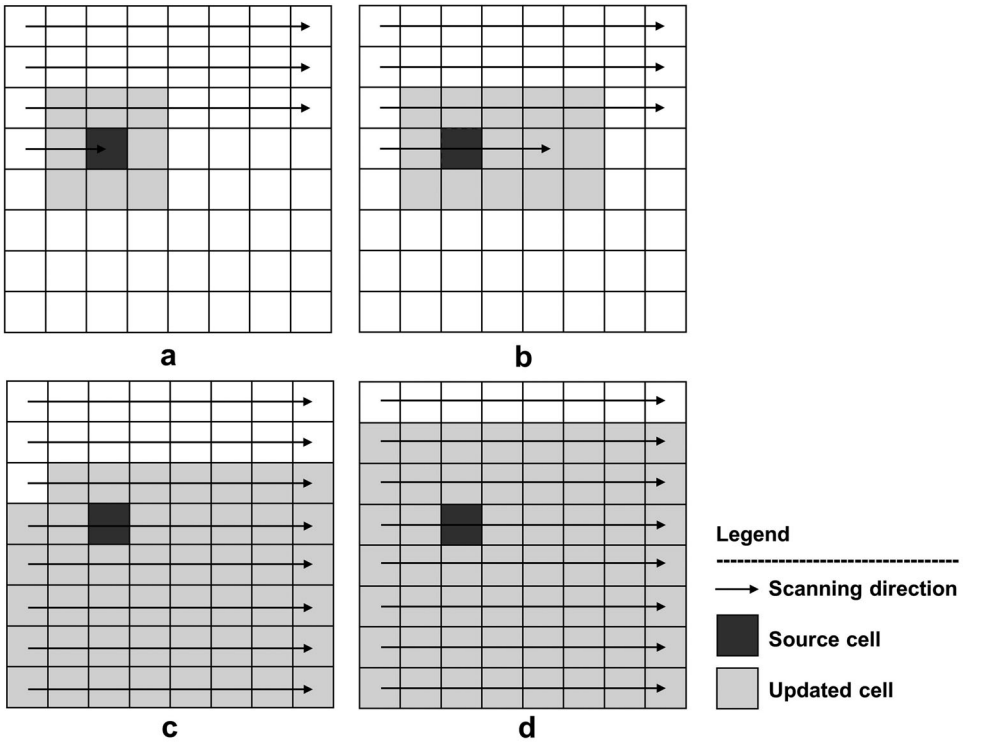


Figure 2. The basic process of a revised sequential algorithm for generating a least-cost surface. The dark grey cell surrounded by the light grey cells indicates the single destination cell in this area and light grey cells indicate the updated cells. a, b, and c illustrate the process of the first iteration. d shows the system at the end of the second iteration. Note that the cells may need further iterations to reach the final accumulative least cost value.

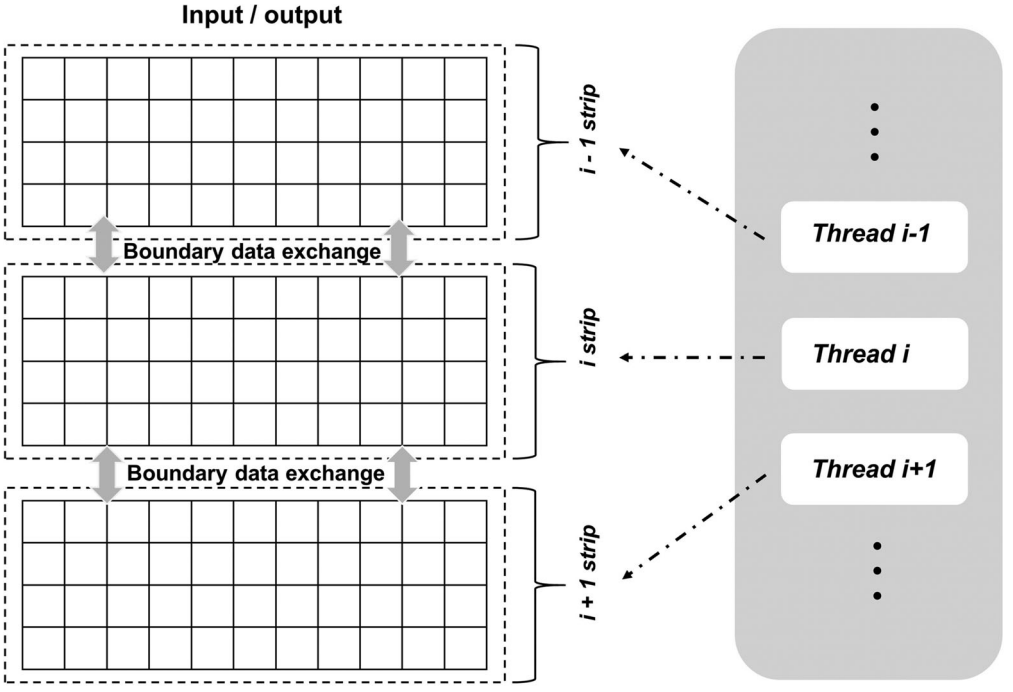


Figure 3. The domain decomposition for the parallel computing scheme. The whole data domain is divided into static sub-domains in the vertical direction, and the sub-domains are assigned to different threads. The adjacent sub-domains will exchange their boundary data between two consecutive iterations.

2.3. Parallelized MSO based on shared-memory parallel programming

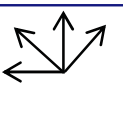
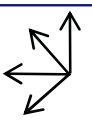
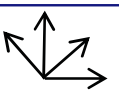
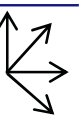
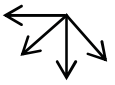
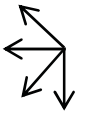
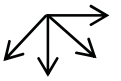
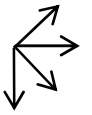
The sequential raster operation basis of the MSO algorithm directs a straightforward parallelization implementation. Here, we conducted OpenMP based shared-memory parallel programming (naming as MSO_OpenMP algorithm) on LCS calculation. This decomposes the data domain into static strips by rows, where the number of strips equals the number of available CPUs for computing (Figure 3). The MSO operation then requires several variables for the within-stripe scanning, involving the indexes of the start row and end row of each strip, and the numbers of rows within a strip. Each of the processers scans the assigned data strip until all the grid cells within the global data domain do not need any update.

It should be noted that, the operations of the rows, where i strip adjacent to $i+1$ strip, may result in incorrect value assignment if they simultaneously write to the same grid cell. However, the incorrectly assigned values will be updated during the next iteration because all the least cost values are deterministic in LCS calculation.

3. Complexity of the raster-based algorithms

Yao and Shi (2015) observed that the time complexity of SSO and MSO for calculating flow accumulation was determined by the longest streamline in the area, and they

Table 1. *Unconnected directions* of the eight basic scanning orders. One can compare this illustration with Table 1 in Yao and Shi (2015).

Scanning Order	UL-row	UL-column	UR-row	UR-column
Unconnected direction				
	LL-row	**LL-column**	**LR-row**	**LR-column**
Unconnected direction				

derived the complexity of SSO to be $O(N^{1.5})$ and MSO to be O(bNlogN), where $b < 1$, based on that observation. We realize that their derivation is specific to hydrological process in which there usually exists a main channel, where this channel is usually the longest and the most complex. In this study, we try to generalize the notion of complexity of raster-based algorithms for a *global directional operation*. By *global directional operation*, we mean a raster operation that is based on local direction measurement, but also requires information from an area beyond the regularly defined local neighborhood. Calculations of flow accumulation and LCS belong to this type of operation.

For the purposes of discussing algorithm complexity, we first define a raster cell to be an *unconnected cell* if it is on the path and the moving direction from its upstream cell to this cell does not match the general moving direction of the current scanning order. We call such a direction *unconnected direction* and note that different scanning orders have different *unconnected directions*. Table 1 illustrates the *unconnected directions* of the eight basic scanning orders.

We then define a *moving step*: first of all, the first cell of the path is the natural start of the *first moving step*; the last cell of the path is the natural end of the last *moving step*; then, between the first and last cells of the path, each *unconnected cell* and its immediate upstream cell on the path form a *moving step*; and finally, the section of the path between the end cell of the previous *moving step* and the beginning cell of the next *moving step* forms a *moving step*. Under the conventional UL-row scan, the top path (A→B) has only one *moving step* (Figure 4), from the very first to the very last cell, because it does not have any *unconnected cell* in the middle under UL-row. The bottom path (C→M), however, has nine *moving steps* because it has eight *unconnected cells* (D, E, F, G, J, K, L, and M) and a section between two *moving steps* (G→I). Each *moving step* requires one scan. Thus, with UL-row, the bottom path takes nine scans to complete, and the overall time cost of the entire raster is determined by the path that requires most scans, which in the case of Figure 4 is the bottom path.

If we denote the path that has the most *moving steps* under the adopted scanning order(s) among all paths to be *P*, the complexity of both SSO and MSO can be written as:

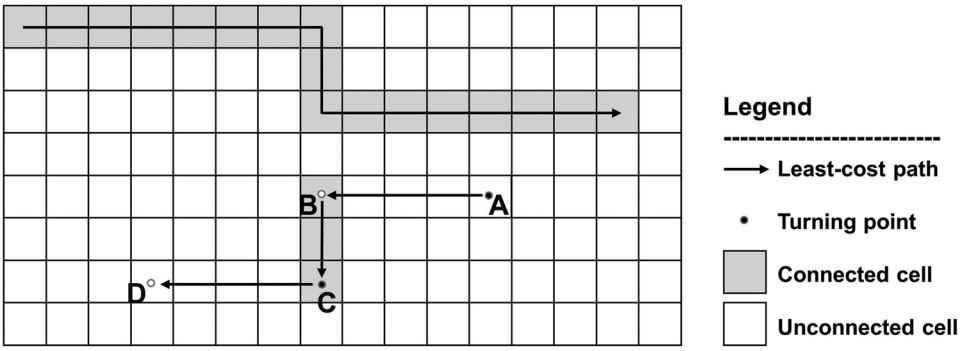


Figure 4. *Unconnected cells* (shaded) under the upper-left by row scanning order (UL-row).

$$N \times c \tag{1}$$

where N is the total number of cells in the raster and c is the total number of *moving steps* on P. According to the definition of *moving step* above, every *unconnected cell* defines one *moving step* and some define a second one. For example, cell G in Figure 4 not only defines *moving step* F→G, but also works together with H to define I→G. Therefore, c will not exceed two times the number of *unconnected cells*. For SSO, it is obvious that the number of *unconnected cells* is in the order of N, so c is also in the order of N, and the worst-case complexity of SSO is $O(N^2)$. MSO, however, is much more complicated. With MSO, an *unconnected cell* under one scanning order may be not *unconnected* under another scanning order. In other words, different scanning orders may mutually disable each other's *unconnected cells*, and the eventual number of *moving steps* is reduced as a result. However, this mutual-disabling process is complicated and it is hard to precisely derive c under MSO. Yao and Shi (2015) derived that MSO's c is in the order of log(N) for the flow accumulation problem, based on certain hydrological laws. For LCS, we have not identified similar laws to derive a good estimate of c, but based on the above discussion of the mutual-disabling process, it is most likely that c is not linearly related to N, and this notion is likely to be generalizable. Most importantly, c is key to analyzing the difference between SSO and MSO. In this research, while we have not theoretically untangled the complexity of MSO, we would like to give it an empirical exploration. In the next section, we present experiments conducted with real data which not only empirically demonstrate the difference between SSO and MSO, but also indicate the complexity of MSO.

A future research topic is to estimate c in different analytical procedures, similar to what Yao and Shi (2015) did for the flow accumulation problem. Different analytical procedures may have different factors determining or affecting c. In the flow accumulation problem, the most important factor is the stream system structure. In LCS, the number of source points is a factor: more source points may result in a shorter P (the path with most *moving steps*), and in turn a smaller c. Further, generalization of the representation of c of MSO and in turn the complexity of MSO is a challenging but meaningful topic for future research.

On the other hand, the time complexity of a naive Dijkstra's algorithm is $O(|V|^2)$, where $|V|$ represents the number of vertices in the dataset (Dijkstra 1959). Fredman

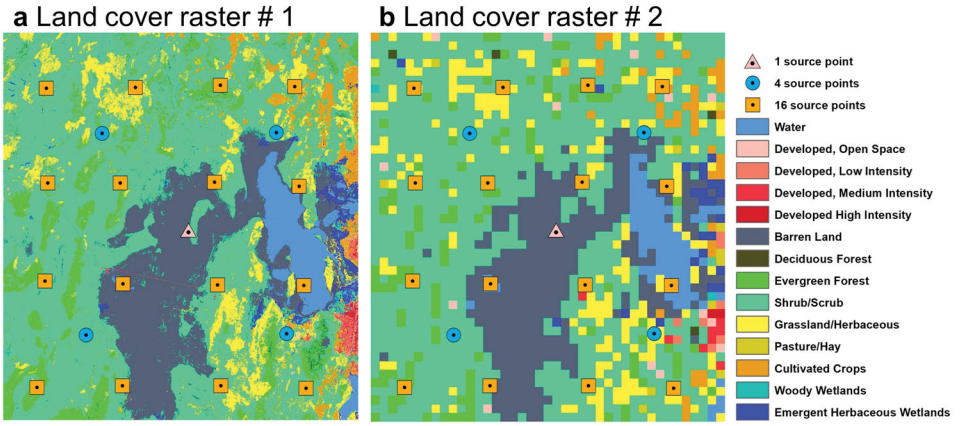


Figure 5. The cost friction input derived from the 2011 national land cover database and the three source point sets.

and Tarjan (1987) propose a standard procedure that employs a binary heap (for keeping the queue sorted) to achieve a time complexity of O(E + VlogV) for Dijkstra's algorithm, where E is the number of edges, and V is the number of vertices. When using raster data, E is linearly related to V (equivalent to the number of cells), i.e. O(E) ~O(V), and thus the overall complexity of the raster process is O(VlogV). The cost of this reduced time complexity is the potentially additional memory used by the raster method. An implication of this is that Dijkstra's algorithm is not sensitive to characteristics of the study areas and particular analytical settings (e.g. number of source points).

4. Experiments and results

4.1. Comparison of LCS algorithms

To empirically evaluate the performance of different algorithms for LCS, we generated two cost (friction) raster layers and three source-point layers (Figure 5). The land cover raster 1 cost raster (Figure 5a) is 8960 × 8223 pixels in size and contains land-use values generated from a 30-m 2011 national land cover database (NLCD) (Wickham *et al.* 2017). The values in land cover raster 1 are continuous and the pattern is complex. To examine the influence of land cover pattern on LCS calculation, we aggregated land cover raster #1 to a 6000-m spatial resolution, and then resampled it back to 30-m resolution (land cover raster #2 in Figure 5b). Compared with land cover raster 1, the land cover pattern in raster 2 is patchier and thus simpler. Further, we applied three sets of source points (1, 4, and 16 points) to both cost data layers, which were regularly distributed in the study area (Figure 5).

For the SSO algorithm, we implemented the conventional UL-row order. For the MSO algorithm, we adopted the eight basic scanning orders. For the MSO-OpenMP algorithm, we examined the running time of the algorithm as the number of computing CPUs was increased from 1 to 30. The experiments were conducted on a

Table 2. The computation time of different least-cost surface calculation algorithms in terms of the numbers source points using slope cost inputs.

	Computation time (seconds) for number of source points		
Algorithms	1 point	4 points	16 points
Landcover raster 1 (8960 × 8223)			
Dijkstra's algorithm	21	23	22
SSO algorithm	3788	1380	1221
MSO algorithm	124	83	79
MSO_OpenMP algorithm (with 20 logical processors)	12	10	8
Landcover raster 2 (9000 × 8200)			
Dijkstra's algorithm	22	23	24
SSO algorithm	1062	1244	785
MSO algorithm	19	15	18
MSO_OpenMP algorithm (with 20 logical processors)	6	5	5

workstation equipped with Intel Xeon (R) E5-2680 16-Cores (32 logical processors) with 2.7 GHz, and 128 GB RAM. The operating system is Windows 7Server 2021 R2 standard 64-bit (6.3, Build 9600).

The execution times of different algorithms in generating LCS based on two different land cover layers with the three sets of source points are given in Table 2. For land cover raster 1 as the cost layer, the MSO algorithm was 15–30 times faster than the SSO algorithm, but it was still approximately four to six times slower than Dijkstra's algorithm. When the number of source points increased from 1 to 16, the running times of the SSO and MSO algorithm decreased by 36-67%, whereas the running times of Dijkstra's algorithm and MSO-Dijkstra algorithm do not show significant change. In all cases, the MSO_OpenMP algorithm (with 20 cores) is the fastest, with a running time only about 36 ∼ 57% of Dijkstra's algorithms.

For Land cover raster 2, all the raster-based algorithms used much less running time than that for the more complex cost layer (Land cover raster 1), whereas Dijkstra's algorithm was not considerably affected by this difference. For Land cover raster 2, of more note is that the MSO method surpassed Dijkstra's algorithm, taking only 65% to 86% of the running time used by the latter. Similar to the situation with Land cover raster 1, Dijkstra's algorithm remains less sensitive to the number of input source points, whereas the other algorithms' running times considerably decreased as the number of source points increased. Again, the MSO_OpenMP algorithm (20 logical processors) is the fastest overall, taking only 27% of the time used by Dijkstra's algorithm, and using 31% of the time used by the serial MSO algorithm.

Figure 6 shows the LCS surface calculated with SSO and MSO. To demonstrate the effect of *unconnected cells*, we display two representative cost paths on the map. One is the path that takes the most SSO scans, and the other is the one that takes most MSO scans. We present these two paths because they are the worst case for SSO and MSO on this same land cover raster, which means that they eventually determine the overall time cost of SSO and MSO, and thus are illustrative when comparing the two methods. It turns out that on both paths, the MSO performance is substantially better than SSO (56 vs. 6414 scans for the first path, and 85 vs. 4675 for the second path). This difference appears not to be due to the size of the raster or local characteristics, as the data used in the experiment is the same. Neither is it due to the length of the paths, as both methods worked on the same paths and the two paths actually have

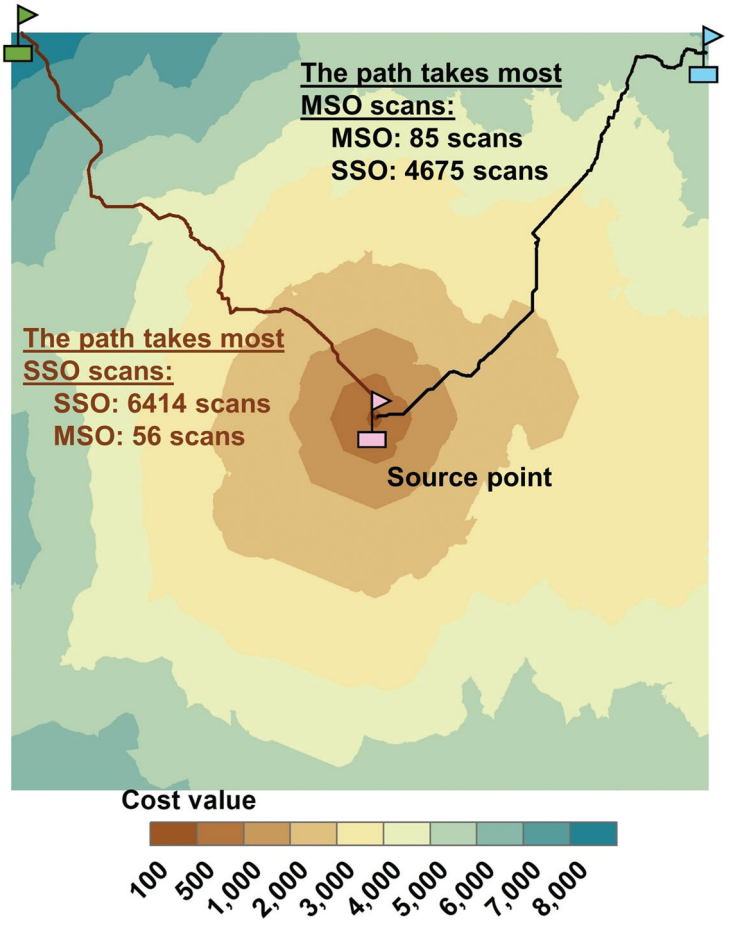


Figure 6. The least cost surface of land cover raster 1 calculated by SSO and MSO. Two representative least cost paths on the surface are displayed. They are the paths that have taken the most scans to complete for SSO and MSO, respectively.

similar lengths. This experiment demonstrates how MSO can substantially reduce the number of effective *moving steps*.

4.2. Empirical comparison of the time complexity

To verify the complexity analysis in Section 3, we conducted a series of experiments to empirically measure and compare the running time of different algorithms. The raster data we used are land cover data. We designated different cost values to different land cover types. We used different spatial extents to clip the original land cover data, generating rasters of different sizes. We set the source point to be at the center of the area (Figure 7a). For each raster, we calculated the LCS using SSO and MSO, and recorded the running time. To better visualize and compare the running time, we plotted the T/N ratio (running time over number of cells) of all rasters, and fit linear and logarithmic functions to the T/N ratio data (Figure 7b). The SSO T/N ratio presents a near perfect linear relation with N ($R^2 = 0.998$), which matches the $\Theta(N^2)$ time

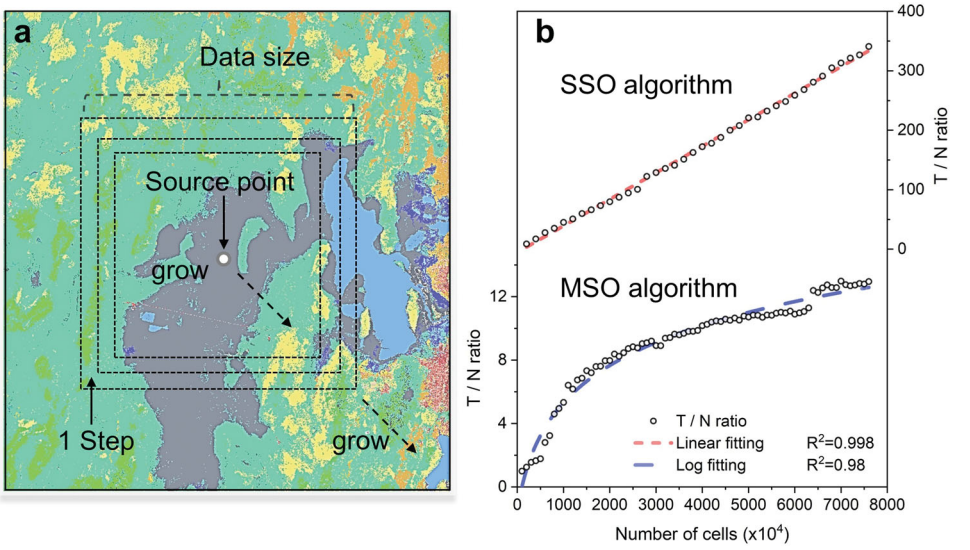


Figure 7. Empirical experiments to compare the time cost of SSO and MSO. a. Land cover raster data of different sizes and the source point for the experiments. We created rasters based on the land cover raster 1, starting with a size of 100 × 100 cells, increasing to 8000 × 8000, with an interval of 100–200 cells on each side. b. The relationship of T/N ratio (running time over number of cells) vs. N (number of cells) of SSO and MSO.

complexity. On the other hand, the logarithmic function fits the MSO T/N ratios ($R^2 = 0.98$), indicating a $\Theta(\mathrm{NlogN})$ time complexity for MSO.

4.3. Parallel computing of MSO algorithm

We also found that the OpenMP-based shared-memory parallelization can substantially decrease the running time of the MSO operation (Figure 8). For both land cover rasters, the most notable decreases in running time were observed when the processors increased from 1 to 5 (Figure 8), for which the average running time decreased by approximately 3–4 times. The efficiency peaked with an average running time of 10 s for land cover raster 1 and 5 s for land cover raster 2 (Figure 8).

5. Concluding remarks

We proposed and implemented improvements to the raster-based method for calculating the least cost surface (LCS). While Yao and Shi (2015) work proposes the multiple scanning order (MSO) method and presents its application in the flow accumulation problem, the LCS problem we tackle in this study is quite different from the flow accumulation problem, in terms of both the analytical process and the practical purpose. In this study, we have adapted MSO to LCS and evaluated the performance of the new algorithm.

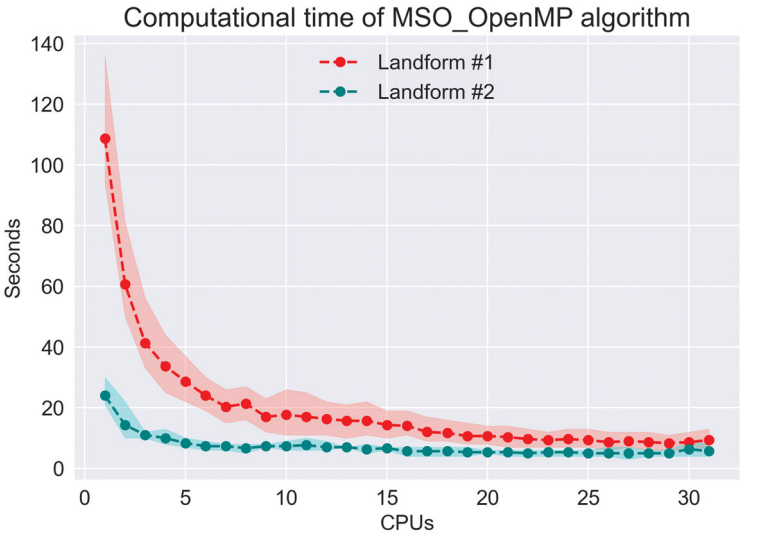


Figure 8. The computation time of MSO_OpenMP algorithm for the least cost surface calculation as the number of logical processors increased from 1 to 30. The dotted line and shaded area represent the mean, and ranges (min. to max.) of running time with different point source sets.

Our experiments lead to three empirical findings. First, on a more complicated landscape, MSO is substantially more efficient than SSO, but that it is not necessarily better than the conventional Dijkstra's algorithm and that it can be sensitive to the number of source points. On a relatively simple landscape, MSO can surpass the Dijkstra's algorithm and become insensitive to the number of source points. Third, a parallelized MSO can be generally and considerably more efficient than the Dijkstra's algorithm. This last finding invites special attention, because the high parallelizability is a distinct advantage of MSO when compared with Dijkstra's algorithm.

Besides those empirical findings, the novel methodological contributions of this work can be summarized as follows:

1. We noted an unnecessary restriction in the conventional raster scanning operations in the LCS calculation where current iteration only updates those cells that were processed in previous iteration. The removal of this restriction considerably simplifies the process and improves its efficiency.
2. We tried to generalize the analysis of the time complexity of SSO and MSO initiated by Yao and Shi (2015) with the special case of the flow accumulation problem. We determined that the theoretical worst-case complexity of SSO is $O(N^2)$, and this is confirmed by our empirical experiments. For MSO, we have not untangled the complicated relationship between different scanning orders, and hence have not reached a precise theoretical complexity. However, our experiments strongly suggest the complexity of MSO to be O(NlogN). They are consistent with the complexities of SSO and MSO that Yao and Shi derived for the flow accumulation problem. This consistency suggests that $O(N^2)$ and O(NlogN) may have a general meaning for SSO and MSO in geographic problems.

3. This work demonstrates a particular advantage of MSO – the high parallelizability – in those analyses whose current standard algorithms (e.g. Dijkstra's algorithm for LCS) cannot be easily parallelized. This last issue was not elaborated in Yao and Shi (2015), but its importance becomes increasingly obvious as high-resolution and therefore large-size raster data become increasingly available.

In this study, we have only achieved a preliminary implementation of parallel computing for the MSO calculation of LCS, mainly for demonstrating the advantage of the MSO method in parallelizing directional raster analyses. Future research will include in-depth investigation on related topics. For example, what is the connection between the strip width and the average/max path length? Would it help if the data decomposition is by columns instead of rows when the scan is in a vertical direction? In light of the success in adopting MSO raster operation into the parallel computing of LCS, we will apply our algorithm to the processing of massive raster data (larger than the computer memory), which requires smart I/O-efficient strategies (Toma *et al*. 2001, Arge *et al*. 2003). Moreover, The MSO algorithm introduced in this study is specific to grid data (regular mesh) processing. In a future study, we will invest more efforts in applying a similar strategy to data structures such as trees and graphs.

Acknowledgement

We sincerely thank the valuable contributions given by the editor (Professor Shawn Laffan) and the anonymous reviewers.

Data and codes availability statement

The relevant datasets of this study are archived in the *zenodo* site (https://zenodo.org/record/6339694#.Yif6u3rMKUl).

Disclosure statement

No potential conflict of interest was reported by the author(s).

Notes on contributors

Yuanzhi Yao is a postdoctoral research fellow at the College of Forestry and Wildlife Sciences, Auburn University and a research professor at the School of Geographic Sciences, East China Normal University. He develops hydrological, and water quality components of terrestrial ecosystem model and disseminates research results on climatic and anthropogenic influences on the physical and biogeochemical processes within the inland water ecosystem and the associate greenhouse gas emissions.

Xun Shi is a professor at the Department of Geography, Dartmouth College. His specialty is in geographic information systems (GIS), quantitative spatial analysis and modeling, and the geocomputational approach. He has explored the applications of GI technologies in various domains, particularly in health studies, digital soil mapping, urban and regional planning, public participation, and renewable energy studies.

Zekun Wang received his Ph.D degree (2022) at the Department of Mechanical Engineering, Auburn University. He mainly engaged in the application of machine learning methods in image

processing. He also concerns about the energy usage and reliability of electrical device, and whisker growing behavior and suppression under extreme environment.

ORCID

Yuanzhi Yao http://orcid.org/0000-0003-2387-4598
Xun Shi http://orcid.org/0000-0002-7169-5175
Zekun Wang http://orcid.org/0000-0003-1641-1229

References

Arge, L., *et al.*, 2003. Efficient flow computation on massive grid terrain datasets. *GeoInformatica*, 7 (4), 283–313.

Balbi, M., *et al.*, 2019. Title: Ecological relevance of least cost path analysis: an easy implementation method for landscape urban planning. *Journal of Environmental Management*, 244, 61–68.

Brabyn, L., and Skelly, C., 2002. Modeling population access to New Zealand public hospitals. *International Journal of Health Geographics*, 1 (1), 3–9.

Collischonn, W., and Pilar, J.V., 2000. A direction dependent least-cost-path algorithm for roads and canals. *International Journal of Geographical Information Science*, 14 (4), 397–406.

Crauser, A., *et al.*, 1998., A parallelization of Dijkstra's shortest path algorithm. *In: International Symposium on Mathematical Foundations of Computer Science*. Berlin: Springer, 722–731.

Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 (1), 269–271.

Douglas, D.H., 1994. Least-cost path in GIS using an accumulated cost surface and slopelines. *Cartographica*, 31 (3), 37–51.

Elsheikh, R.F.A., and Hassan, W.A.S., 2016. Analysis of least cost path by using geographic information systems network and multi criteria techniques. *International Journal of Multidisciplinary Sciences and Engineering*, 7 (5), 1–6.

Fredman, M.L., and Tarjan, R.E., 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34 (3), 596–615.

Gonçalves, A.B., 2010. An extension of GIS-based least-cost path modelling to the location of wide paths. *International Journal of Geographical Information Science*, 24 (7), 983–996.

Gorelick, N., *et al.*, 2017. Google Earth Engine: planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 202, 18–27.

Jasika, N., *et al.*, 2012., Dijkstra's shortest path algorithm serial and parallel execution performance analysis. *In*: *MIPRO, 2012 proceedings of the 35th international convention*. IEEE, 1811–1815.

LaRue, M.A., and Nielsen, C.K., 2008. Modelling potential dispersal corridors for cougars in midwestern North America using least-cost path methods. *Ecological Modelling*, 212 (3–4), 372–381.

Planchon, O., and Darboux, F., 2002. A fast, simple and versatile algorithm to fill the depressions of digital elevation models. *CATENA*, 46 (2–3), 159–176.

Rosenfeld, A., and Pfaltz, J.L., 1966. Sequential operations in digital picture processing. *Journal of the ACM*, 13 (4), 471–494.

Scott, L.M., and Janikas, M.V., 2010. Spatial statistics in ArcGIS. *In*: *Handbook of applied spatial analysis*. Germany: Springer, 27–41.

Soltani, A.R., *et al.*, 2002. Path planning in construction sites: performance evaluation of the Dijkstra, A*, and GA search algorithms. *Advanced Engineering Informatics*, 16 (4), 291–303.

Teng, M., *et al.*, 2011. Multipurpose greenway planning for changing cities: a framework integrating priorities and a least-cost path model. *Landscape and Urban Planning*, 103 (1), 1–14.

Toma, L., *et al.*, 2001. Flow computation on massive grids. *In*: *Proceedings of the 9th ACM international symposium on Advances in geographic information systems*. 82–87.
Vincent, L., 1993. Morphological grayscale reconstruction in image analysis: applications and efficient algorithms. *IEEE Transactions on Image Processing*, 2 (2), 176–201.
Wickham, J., *et al.*, 2017. Thematic accuracy assessment of the 2011 national land cover database (NLCD). *Remote Sensing of Environment*, 191, 328–341.
Xu, J., and Lathrop, R.G., Jr, 1994. Improving cost-path tracing in a raster data format. *Computers & Geosciences*, 20 (10), 1455–1465.
Xu, J., and Lathrop, R.G., Jr, 1995. Improving simulation accuracy of spread phenomena in a raster-based geographic information system. *International Journal of Geographical Information Systems*, 9 (2), 153–168.
Yao, Y., and Shi, X., 2015. Alternating scanning orders and combining algorithms to improve the efficiency of flow accumulation calculation. *International Journal of Geographical Information Science*, 29 (7), 1214–1239.
Yao, Y., Tao, H., and Shi, X., 2012. Multi-type sweeping for improving the efficiency of flow accumulation calculation. *In*: *2012 20th International Conference on Geoinformatics*. IEEE, 1–4.

Appendix. Pseudo codes of the proposed algorithms

In this section, we provide pseudo codes of the MSO algorithm (Function 1 and 2), the queue-based Dijkstra's algorithm (Function 3), and the OpenMP based parallelization algorithm (Function 4 and 2):

//Meanings of the variables:

//preBuf: a buffer hold results from last iteration.

//outBuf: a buffer hold results from current iteration.

//soureBuf: a data raster include all the source points

//costPassage: a buffer that holds pre-calculated friction value for each cell.

//numPixels: total number of cells in the input land cover raster.

//nrow: number of rows in the input land cover raster.

//*ncol*: number of columns in the input land cover raster.

//pos: the starting cell of a scan

//start_i: the staring row of a scan

//end_i: the ending row of a scan

//*leap_i:* the number of cells the pointer to jump in a loop by row

//start_j: the starting column of a scan

//end_j: the ending column of a scan

//*leap_j:* the number of cells the pointer to jump in a loop by column

//NunmberOfProcessors: number of processors; specified by the user

//The name of each scanning order was defined in Lookup Table.

Function 1:

```
// The main function of the entire procedure. Will enumerate each of the 8 scanning orders.
void MSO_LeastCostSurface (costPassage, outBuf, preBuf)
  Initialize preBuf by assigning 0 to each cell
  while any grid cell still needs to be updated do
      Enumerate eight scanning orders using the look-up table:
          Scanning_of_an_Order (pos, start_i, end_i, leap_i, start_j, end_j, leap_j,
                                costPassage, outBuf, preBuf, nrow, ncol, cellsize)
      end loop
  end loop
```

Function 2

```
// Scans the raster using a particular order. Updates the cost value for each cell.
bool ScanningOfAnOrder (pos, start_i, end_i, leap_i, start_j, end_j, leap_j, costPassage,
outBuf, preBuf, nrow, ncol,cellsize)
    bool flag = false
    for (int i = start_i; i < end_i; i++)
        for (int j = start_j; j < end_j; j++)
            if outBuf[pos] - preBuf[pos] == 0 then // This cell was not updated in last iteration
                pos += leap_j
                continue
            else
                flag = true
            end if
            Enumerate all neighbors of pos do
                if a cardinal-direction neighbor then
                    costTemp = outBuf[i] + (costPassage[i] +
                            costPassage[neighbor of pos]) ×cellsize
                else // This is a diagonal-direction neighbor
                    costTemp = outBuf[i]+(costPassage[i]+
                            costPassage[neighbor of pos]) ×\sqrt2cellsize
                end if
                if (costTemp < outBuf[neighbor of pos]) then
                    outBuf[neighbor of pos] = costTemp
                end if
            end loop
            preBuf[pos] = outBuf[pos]
            pos += leap_i
        end for
        pos += leap_i
    end for
    return flag
```

Function 3

```
// Dijkstra's algorithm for LCS
void Dijkstra_LeastCostSurface (costPassage, outBuf)
  Minimal_priority_queue<float, int> cellList
  for all the cells(i) in soureBuf do
    cellList.push(Node(0,i))
  end for
  do
    float pos = cellList.pop()
    for all the neighbors of pos do
        if a cardinal-direction neighbor then
            costTemp = outBuf[i] + (costPassage[i] +
                costPassage[this neighbor ]) ×cellsize
        else
            costTemp = outBuf[i]+(costPassage[i]+
                costPassage[this neighbor]) ×\sqrt2cellsize
        end if
        if  costTemp < outBuf[this neighbor] then
            outBuf[this neighbor] = costTemp
            cellLi-st.push(Node(costTemp, this neighbor))
        end if
    end for
  while (cellList is not empty)
```

```
//start_row represents the index of the first row of the tile for the OpenMP parallel
//computing, end_row denotes the index of the last row.
//rows_strip represents the number of rows in each tile.
//outBuf and preBuf is initialized with source point input, accumulated cost surfaces are
//created for k^th and k-1^th iterations—each cell is given value –1 and the pixel have source
//value 0 as accumulated cost value in k-th iteration;
```

Function 4

```
// Parallelize LCS calculation with OpenMP
void MSO_LeastCostSurface_OMP (costPassage, outBuf, preBuf, NunmberOfProcessors)
  Initialize preBuf by assigning 0 to each cell
  #pragma omp parallel for
  Define the properties of the tile (start_row, end_row, rows_strip ).
  for NunmberOfProcessors
    while any grid cell still needs to be updated do
      Enumerate the eight scanning orders using the look-up table:
        LeastCostSurface (pos, start_i, end_i, leap_i, start_j, end_j, leap_j,
            costPassage, outBuf, preBuf, nrow, ncol, cellsize)
      end loop
    end loop
  end for
```

Lookup Table: Scanning orders and their parameter values (used by Functions 1 and 4).

Scanning order	pos	start_i	end_i	leap_i	start_j	end_j	leap_j
Upper-left by row	0	0	r	1	0	c	1
Lower-right by row	n – 1	0	r	–1	0	c	–1
Lower-left by column	(r – 1) * c	0	c	(r – 1) * c + 1 + c	0	r	–c
Upper-right by column	c – 1	0	c	(1 – r) * c – 1 – c	0	r	c
Upper-right by row	c – 1	0	r	c * 2	0	c	–1
Lower-left by row	(r – 1) * c	0	r	–c * 2	0	c	1
Upper-left by column	0	0	c	(1 – r) * c + 1 – c	0	r	c
Lower-right by column	n – 1	0	c	(r – 1) * c – 1 + c	0	r	–c

where *n* is total number of cells in the raster; *r* is number of rows of the raster; and *c* is number of columns of the raster.