FAST 3-D EUCLIDEAN CONNECTED COMPONENTS

W Randolph Franklin
Electrical, Computer, and Systems Engineering Dept
Rensselaer Polytechnic Institute
Troy, NY USA 12180
mail@wrfranklin.org

Salles Viana Gomes de Magalhães
Universidade Federal de Viçosa
Viçosa – MG
Brasil 36570-900
sallesviana@gmail.com

Eric N. Landis
University of Maine
Orono, ME, USA
landis@maine.edu

ABSTRACT

We present an efficient algorithm and implementation for computing the connected components within a 3-D cube of voxels, also known as the Euclidean union-find problem. There may be over 10^9 voxels. The components may be 8-connected or 26-connected. Computing connected components has applications ranging from material failure in concrete under increasing stress to electrical conductivity in complex metal objects to elasticity in 3D printed parts. One key to efficiency is representing voxels by 1-D *runs* of adjacent voxels. We also compute each component's surface area. As a special case, 2-D connected components of images may easily be computed.

***K*eywords** connected components · union-find · voxel array

1 Introduction

The general *connected component* problem takes a graph G=(V,E) as input. V is a set of vertices and E is a set of undirected edges. Each edge connects a pair of vertices, thereby making them *adjacent*. A connected component contains vertices joined by the transitive completion of the adjacency relation.

The *3-D Euclidean* case is a useful specialization that has been studied less. Here, the universe is a 3-D grid of binary voxels. A *0* value for a voxel represents a solid voxel, i.e., a vertex. An edge is implicit whenever two adjacent voxels are both solid. The adjacency relation may be either 6-adjacency or 26-adjacency. With 6-adjacency, two voxels are adjacent if exactly one of their coordinates differs by one, e.g., (10,20,15) and (10,19,15). With 26-adjacency, one or more coordinates differs by one, e.g., (10,20,15) and (11,19,16).

Since 3D illustrations are difficult, Figure 1 shows 2D connectivity. The original image was a scanned 2D map from the USGS. The greyscale image was thresholded to produce a B&W image of size 18573×19110 bits or pixels. Then the black pixels were grouped into components using 8-connectivity. Each component was then randomly colored. The figure is a 2000×2000 detail of the full image, for which 32858 components were computed.

Note the complexity of some of the components, such as the dark green one in the lower right that contains several letters and sections of roads. Its zigzag shape is assembled gradually as smaller subcomponents are combined. This 2D example was computed by treating it as a $18573 \times 19110 \times 1$ 3D problem. The components in a full 3D example can be much more complicated. Imagine a ball of yarn composed of several long strings of yarn all tangled together, with each string one component.

Secondarily to computing the components' voxels, we compute each component's surface area. Also, we wish to process datasets with well over 10^9 vertices and edges. This paper presents various improvements to implement this efficiently.

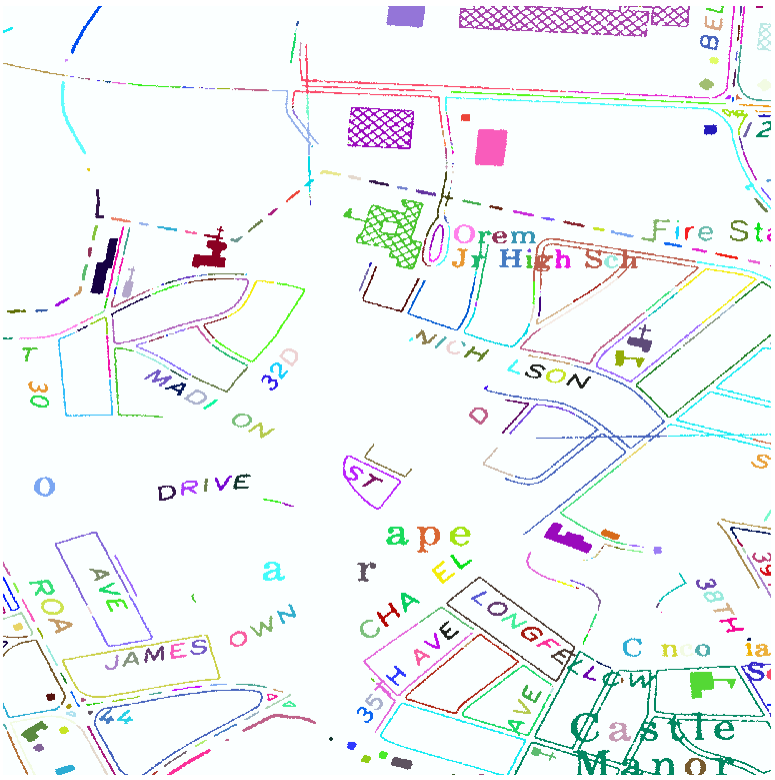

Figure 1: 2D connected components

This algorithm builds on the classic *union-find* or *merge-find* algorithm, which computes the connected components for a general undirected graph. Initially, each vertex is its own component. Then, each edge is processed in turn. A *find* operation finds which components contain the edge's endpoints. If those two components are different, then a *union* operation combines them into one component. The time is very slightly over linear [6]. Specifically, the time to execute n *union* and m *find* operations on n elements is $O(n + m\alpha(m, n))$, where $\alpha(m, n)$ is the functional inverse of Ackermann's function.

It uses the the *disjoint-set* data structure, [1] to arrange the vertices in a forest of trees, with each vertex having a pointer to another vertex. Each tree is one connected component. The union-find algorithm reads the data and rewrites vertex pointers several times. Various implementations exist in addition to [2, 3, 4], but they often build upon general *union* and *find* algorithms. Since they do not exploit the geometric coherency in the Euclidean case, they can be very slow, and take much more space, compared to what is described in this note.

Applications include material failure in concrete under increasing stress, geological porosity and percolation, electrical conductivity in metal objects, elasticity in 3D printed objects, solving mazes, image processing, equivalence of finite state automata [5], and compiling sets of equivalance statements in Fortran.

Parallel connected component algorithms, although not necessary for the Euclidean case, include Zhang et al[8, 7].

2 A better data structure for the Euclidean case

The key to our efficiency in time and space is the careful choice of data structure that is specialized for the Euclidean nature of the data. The efficiency is enhanced by several small optimizations. The details follow.

1. The universe is a 3-D voxel grid or array of $g \times g \times g$ bits, indexed with x, y, and z coordinates.
2. Let n be the number of solid voxels. $n \leq g^3$, and typically, $n \sim g^3/2$. Each solid voxel is considered a vertex of a graph.
3. Each pair of adjacent voxels defines an edge of the graph. The edges are not stored explicitly since they can be enumerated by scanning the voxel grid for adjacent pairs of solid voxels.

4. The major new data structure is a *run*. A run is a sequence of consecutive solid voxels with constant x and y coordinates, and sequentially increasing z coordinates. The runs in a connected component are formed into a *tree*; and each run contains the index r of its father. Runs are indexed from 1 up.

   The root run is marked by making its parent's index 0 or negative. A negative number is used to store some useful information about the whole component like its surface area. It is convenient to assemble the runs into trees, to facilitate the operations. This assembly is done in such a way that each run's father has a larger index than the run, and the root run has the highest index of any run in the tree.

   The runs may be considered to be a *forest* of *trees*, with each tree representing one connected component. Initially, each run is in a separate tree, of which it is the root and only element. Trees will be merged and grow, as described later.

   Possible alternatives to the run include storing the voxels separately or using 2-D slabs. The run format is much more compact than the former, while simpler than the latter.

5. Abstractly, a run is the tuple (x, y, z_l, z_h, r). However, for large datasets, there may be several runs with the same (x, y), so all such runs are grouped together in a ragged array indexed on (x, y). Only (z_l, z_h, r) is stored explicitly for each separate run.

   Ragged arrays:

   (a) This solves the problem of implementing a read-only array whose elements have widely varying size, with many elements being empty, and a few very large. Equivalently, there may be from 0 to many elements for each key.

   (b) It is defined abstractly as follows.
   There are m keys, assumed for this description to be integers from 0 to $m - 1$. The more complex case used in this note, where the key is an ordered pair, follows easily. There are l (key, value) pairs. The value retrieved with key i is an integer l_i and a list of l_i items of some fixed length type, say integer. Note that $l = \sum_i l_i$.

   (c) In our case, the explicit data stored for each run is the tuple (z_l, z_h, r).

   (d) Our implementation of the ragged array goes as follows.
   All the values for the whole ragged array are stored consecutively in one data array a, of size l, ordered by key. There is a *dope vector* d of size $m + 1$. $l_i = d_{i+1} - d_i$. The j-th item with key i is a_{d_i+j}. Random items can be read in constant time, which compares favorably to using a linked list for the items with the same key. If we assume that one integer has size 1 word, and one item has size s words, then the total space for the ragged array is $m + 1 + ls$. This again compares favorably to using a linked list or C++ vector to store the items with the same key. The space used when a key has no values is only one word (in the dope vector). This again compares favorably to C++ vectors.
   Our algorithm inserts elements into the ragged array in order of increasing (x, y). So, new elements are always appended to the end, and the construction is efficient.
   However, if that were not the case, we could construct the ragged array with a read twice, write once, algorithm. Using the dope vector temporarily to store counts, we would read the data once, and count how many times each key is used. Then, we would partially sum those counts to create the actual dope vector, read the data again and populate the data array. However our algorithm is designed so that this is unnecessary.

6. Finally, we can compute the total surface area of each component because we already compute the adjacencies of each new run to compute or update the adjacent run pointers. The component's surface area is the sum of the surface area of each run minus double the area of overlap between every pairs of adjacent runs. That adjacent area is computed as each run is formed and inserted into the ragged array. This ability to compute the areas is a unique advantage of this algorithm.

3 Connected component algorithm

This section describes how many small components are gradually united into a few large ones.

1. The input data is a sequential file of g^3 bits, ordered with x varying slowest and z fastest. Each bit is considered to be a voxel to emphasize its geometric nature.
2. The first part of the algorithm builds the ragged array of 1-D runs and forms a preliminary forest of trees of connected runs.

   (a) While reading the input data sequentially, each empty voxel or the end of a row delimits a new run (z_l, z_h) with known x and y.

(b) We search the ragged array for any runs that overlap in z and that are adjacent below or to the left, that is, with exactly one of $x' = x - 1$ or $y' = y - 1$.

The process involves first checking the runs with $(x, y - 1)$ in the ragged array. Note that we can find the list of those runs in constant time. Searching down the list for runs with overlapping z takes time linear in the number of runs in the list. A binary search would be faster but is unnecessary.

Then we check the runs with $(x - 1, y)$ for overlaps in z. For 26-adjacency, the two runs may also be diagonally adjacent, so the $(x - 1, y - 1)$ list must be checked. Here, two runs also overlap if their ending z-s are offset by one.

Various uninteresting but necessary boundary conditions must be handled.

(c) If the two runs are adjacent, then they are part of the same component (tree), so we record that fact as follows.

We trace the pointers of each run to the root of its tree. Remember that the id of the root is that largest id of any run in that tree, and that the root run will point to fictitious run 0. Of the two root runs of the two trees, pick the one with smaller id. Change its run pointer from 0 to the id of the higher numbered root run. Then repeat the trace from the two runs to the root, making all runs that we touch to point to the new root. This flattens the tree.

(d) Finally, we optimize the trees and compute mass components, as follows. After all the runs have been processed, we pass through the run array from the maximum to minimum numbered run. We link each run directly to its component's root. Simultaneously, we compute the volume of each component. It may be stored in any convenient data structure.

3. We also thoroughly instrument the code with counters and timers in order to understand its performance in practice. It might be that a beautiful optimization offers no significant improvement, and so ought to be removed.

4 Implementation

The research program written to demonstrate this technique is 600 lines of C++ code. It is freely available for non-profit research and education. The data files are very compressible, so we use a file system (zfs) that does that automatically and quickly.

If we wish to minimize the I/O time because the computation time is more interesting, we can put the current directory into an in-core filesystem that exists in the main memory (DRAM). In linux, that location is */dev/shm*. It can use up to $1/2$ of the physical main memory. The cost of reading and writing reduces to the cost of converting between the internal binary number format and the external character string, which is insignificant here.

Earlier implementations of our algorithm, with application to analysis of tomographic images and cracking, damage and fracture in concrete under stress, are described in [2, 3, 4]. That earlier code is available for nonprofit research and education at https://wrf.ecse.rpi.edu/pmwiki/pmwiki.php/Research/ConnectedComponentsImplementation . The current code is being cleaned up and will be posted soon.

4.1 Implementation validation

Permuting the grid's x, y, and z indexes or rotating it around any diagonal or even the grand diagonal completely changes the number, lengths, and connectivity of the runs. However it does not change the number of components, or their sizes, or the total surface area. We made such changes to some large datasets, and compared the outputs. There was a complete match. This may not be completely definitive but is a good check.

5 Examples

The testbed was a Lenovo p73 laptop with dual 6-core Intel Xeon E-2276M CPU @ 2.80GHz and 128GB of DRAM. The OS was Ubuntu linux 21.04. The only important requirement there is that there be enough real memory, because paging is too expensive. Because of the efficient data structures, much larger datasets are possible than if each voxel were stored separately. For extremely large datasets, we see how the algorithm could be designed to page nicely.

We measured CPU time, which was within a few percent of wall clock elapsed time. When the program was rerun, the time changed by less than 5%.

Table 1: Statistics for random dataset with 26-connectivity

Property	**Value**
Universe size	$1000 \times 1000 \times 1000$
Number of voxels	1,000,000,000
Fraction of solid voxels	0.5
Number of runs	250,243,747
Number of components	35
Average number of runs per component	$7.2 \cdot 10^6$
Time	47 sec

Table 2: Statistics for random dataset with 6-connectivity

Property	**Value**
Universe size	$1000 \times 1000 \times 1000$
Number of voxels	1,000,000,000
Fraction of solid voxels	0.5
Number of runs	250,243,747
Number of components	8,963,541
Average number of runs per component	28
Time	33 sec

5.1 Random voxels, 26-connectivity

The first test dataset is a $1000 \times 1000 \times 1000$ universe of random uncorrelated voxels, with the probability of any particular voxel being solid as 0.5. We used 26-connectivity. Some output statistics are in Table 1.

With 26-connectivity when half of the voxels being solid, the components are geometrically very complicated, zigzagging up and down. This algorithm handles that well. 90% of the time was spent reading the data and building the forest of trees. That is why it would be difficult to design a parallel version of this algorithm. In contrast, all the subsequent steps were very fast.

5.2 Random voxels, 6-connectivity

The next test used the same dataset but processed it with 6-connectivity. That caused the components to be much smaller, simpler, and more numerous. That made the execution faster. Some output statistics are in Table 2.

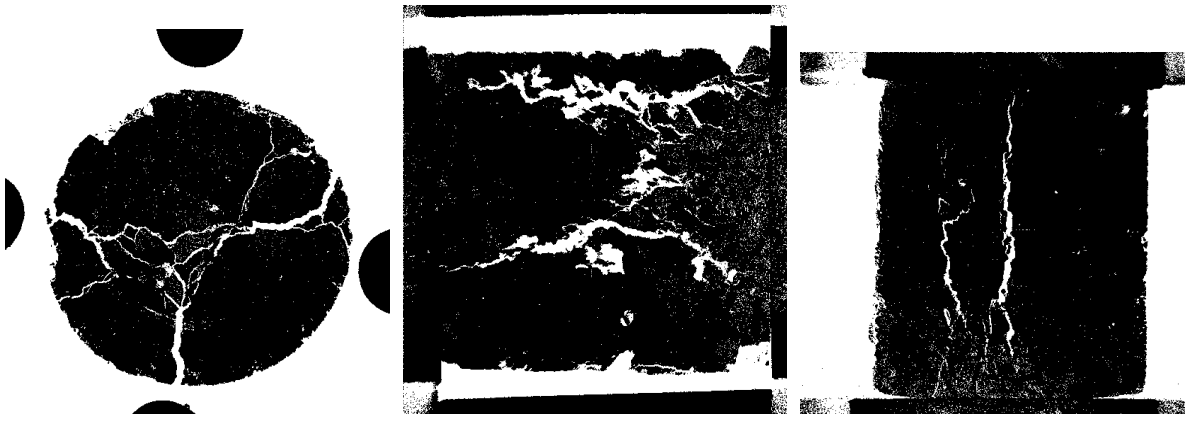

Figure 2: 2D cross-sections of 3D data from scanned concrete cylinder

Table 3: Statistics for concrete dataset

Property	Value
Universe size	$1024 \times 1088 \times 1088$
Number of voxels	1,212,153,856
Fraction of solid voxels	0.5
Number of runs	20,216,828
Ave run size	30
Number of components	4,539,562
Average number of runs per component	4.5
Maximum number of runs per component	2993
Time	11.4 sec

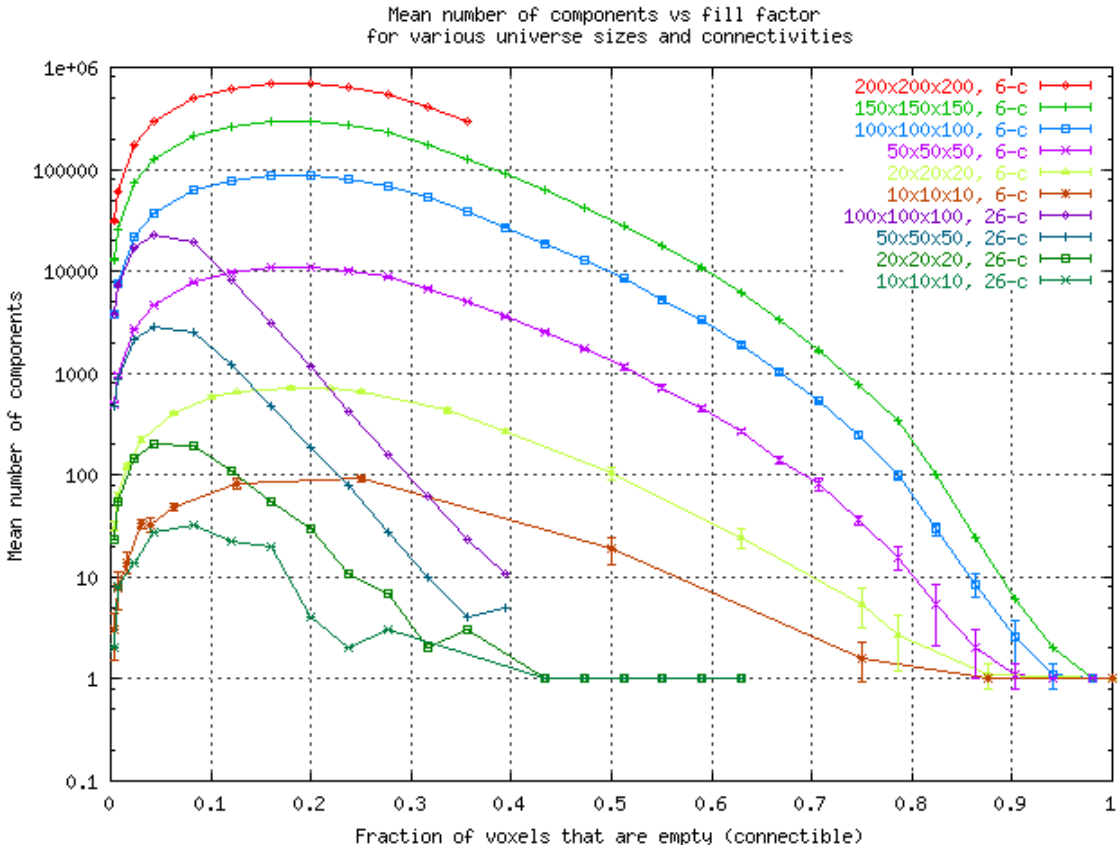

Figure 3: Number of components vs fill fraction

5.3 Concrete under compression

The motivating application for this work was Landis's study of how concrete fails under compression, [4, 3, 2]. Figure 2 shows three cross sections in different directions from a smaller example of a concrete cylinder, perhaps a few cm long, being compressed to partial failure and scanned in the Brookhaven National synchrotron.

The problem is that 2-D sections do not capture the 3D nature of the fractures. Therefore it was proposed that computing the connected components in 3D might be useful. Some output statistics are presented in Table 3.

5.4 Random dataset properties

The program is efficient enough to perform repeated experiments to determine the properties of random datasets. So, we asked this question: How many components are generated for various universe sizes and fill factors or fractions

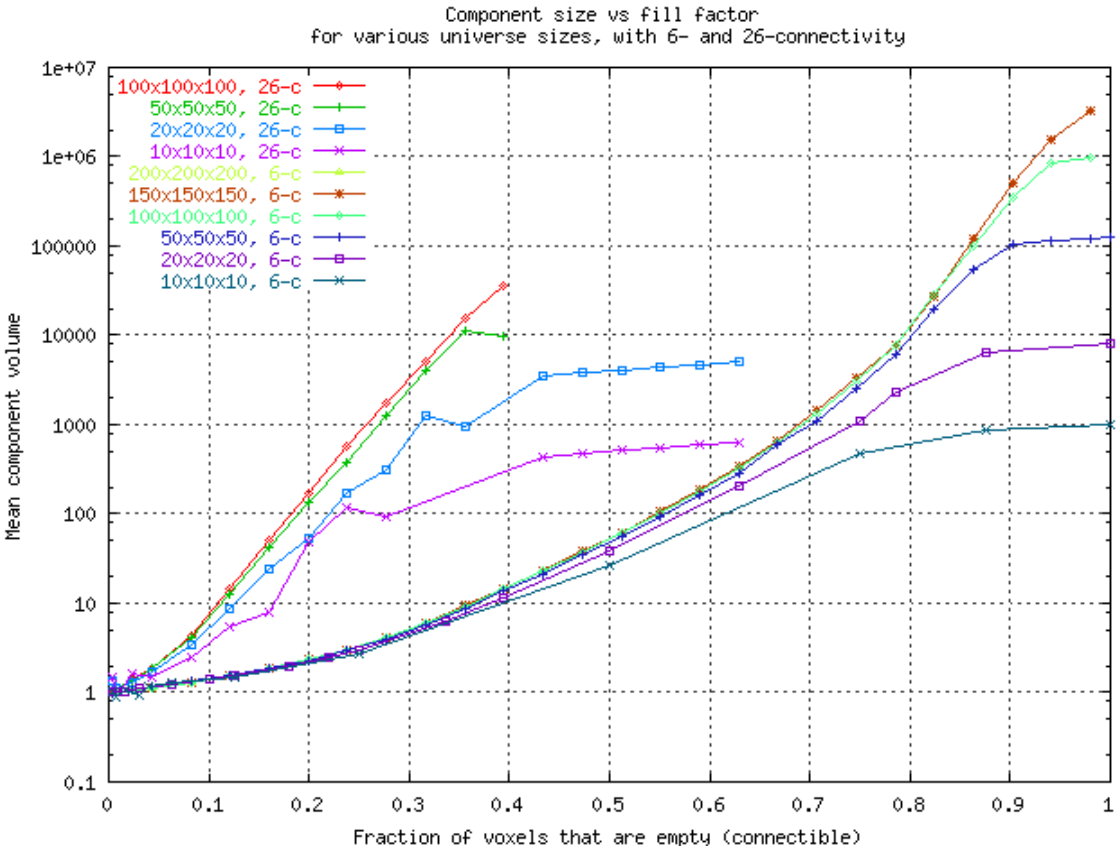

Figure 4: Component size vs fill fraction

(probability that a voxel is empty)? We did 10 runs each for many combos of universe sizes and fill fractions for 6-connectivity. Then we did one run each for some combos of sizes and fill factors for 26-connectivity. Figures 3 and 4 show the results. The error bars for the 6-connectivity are 1 sigma to each side of the mean. The 26-connectivity cases are the curves scrunched to the left.

We observe that the fill fraction giving the maximum number of components is independent of the universe size. For 6-connectivity, p=.2 (approx), gives the most components. For 26-connectivity, p=0.05 does. This independence is reasonable since connectivity is a local property.

We then re-analyzed the above data to show how component size (volume) depended on universe size and fill factor. The lower group of lines is 6-connectivity, while the upper group, which does not extend all the way to the right, is 26-connectivity. The 26-connectivity lines are more irregular since we ran fewer cases.

For component sizes much smaller than the universe size, the universe size was irrelevant, which is reasonable. The limiting case for a fill factor approaching 1 is one component whose size is the universe's volume.

It's not clear what the functional relationship is. In one dimension, component sizes (lengths) would be exponentially randomly distributed, with mean length=$\frac{1}{1-p}$. However, here in 3D, neither the component volumes nor their lengths appear to follow this. This suggests an area for theoretical work.

6 Summary and future

This note presented an algorithm that is efficient in space and time, with implementation, for computing the connected components of a 3D grid of over 10^9 bits or voxels. Although 2D datasets can also be processed as a special case, this algorithm operates in 3D.

Future possibilities include parallelizing it, or modifying it to page efficiently in virtual memory.

References

[1] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, May 1964.

[2] E. N. Landis, T. Zhang, E. N. Nagy, G. Nagy, and W. R. Franklin. Cracking, damage and fracture in four dimensions. *Materials and Structures*, online date: 13 July 2006.

[3] E. Nagy, T. Zhang, W. R. Franklin, G. Nagy, and E. Landis. 3D analysis of tomographic images. In *16th ASCE Engineering Mechanics Conference*, U Washington, Seattle, 16-18 July 2003. electronic proceedings.

[4] G. Nagy, T. Zhang, W. Franklin, E. Landis, E. Nagy, and D. Keane. Volume and surface area distributions of cracks in concrete. In C. Arcelli, L. Cordella, and G. S. di Baja, editors, *Visual Form 2001: 4th International Workshop on Visual Form IWVF4*, volume 2051/2001 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, Capri, Italy, 28-30 May 2001.

[5] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley Professional, 2011.

[6] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[7] Y. Zhang, A. Azad, and A. Buluç. Parallel algorithms for finding connected components using linear algebra. *Journal of Parallel and Distributed Computing*, 144:14–27, 2020.

[8] Y. Zhang, A. Azad, and Z. Hu. *FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence*, pages 46–57. SIAM, 2020.