A FINITE STATE MACHINE APPROACH TO CLUSTER IDENTIFICATION USING THE HOSHEN-KOPELMAN ALGORITHM

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Matthew Lee Aldridge
May 2008

Acknowledgments

I would like to thank all those who have supported and believed in me over these past twenty-two years of my education: family, friends, teachers and professors. In particular, special thanks go to my parents David and Debbie Aldridge; I surely wouldn't be here without you, in so many ways! I am very grateful to have had Dr. Michael Berry as my advisor throughout graduate school, for the opportunities he has pointed me toward, and for his help in keeping me focused on my goals. I also kindly thank Dr. Brad Vander Zanden, Dr. Lynne Parker and Dr. Shih-Lung Shaw for serving on my committee, and Dr. Gregory Peterson for his assistance with the Palm PDA testbed.

Lastly, I would like to graciously acknowledge that I have received much of my graduate school funding through a subcontract from Oak Ridge National Laboratory as part of the RSim Project (SERPD SI-1259) of the Strategic Environmental Research and Development Program (SERDP) supported by the United States Departments of Defense and Energy and the Environmental Protection Agency, and the Yates Dissertation Fellowship at the University of Tennessee.

Abstract

The purpose of this study was to develop an efficient finite state machine implementation of the eponymous Hoshen-Kopelman cluster identification algorithm using the nearest-eight neighborhood rule suitable to applications such as computer modeling for landscape ecology. The implementation presented in this study was tested using both actual land cover maps, as well as randomly generated data similar to those in the original presentation of the Hoshen-Kopelman algorithm for percolation analysis. The finite state machine implementation clearly outperformed a straightforward adaptation of the original Hoshen-Kopelman algorithm on either data type. Research was also conducted to explore the finite state machine's performance on a Palm mobile computing device, and while it was competitive, it did not exceed the performance of the straightforward Hoshen-Kopelman implementation. However, a discussion of why this was the case is provided along with a possible remedy for future hardware designs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

It is often desirable in the analysis of various types of datasets to identify distinct subsets based upon some common traits, a process known as *cluster identification* or simply *clustering*. Often cluster identification is performed using some type of distance metric, which defines the similarity between two data elements. With spatial data, the distance metric used is often simple Euclidean distance, but this is not universal. A subclass of cluster identification techniques considers pure connectivity among cluster components rather than some measure of similarity. In the application of these techniques, the data are typically represented as a lattice, with each data point connected to some number of neighbors according to the lattice structure being used. As cluster formation is driven by adjacency, each point in a cluster can be reached from any other point in the cluster by traversing the lattice connections in an uninterrupted fashion. Methods for identifying such cluster structures can generally be classified as *recursive* or *iterative*.

The recursive, or depth-first, approach has its roots in the works of [16] and [10]. This method makes a single pass over the lattice. When a cluster site—meaning a site that needs to be assigned a cluster label—is encountered, the algorithm then examines each connected, or neighbor, site for other cluster sites that have not yet been assigned a label. This process is repeated recursively for all

such neighbor cluster sites until every site in the entire cluster structure has been assigned the appropriate cluster label. At this point the algorithm continues its pass along the lattice until it encounters the next unlabeled cluster site. While this method can be used to identify all clusters within a lattice, it has the added benefit of allowing for the identification of a single cluster without examining the entire lattice, so long as at least one site of the cluster is known. This selective cluster labeling process is useful in such domains as image processing, where it is better known as *connected component extraction* [7]. Another advantage of the recursive technique is that it is not necessary to maintain an additional data structure for managing the cluster labels. However, in practical terms this method can be limited by the need to store the entire lattice in memory to overcome its poor locality of reference and, even more importantly, the amount of stack space required for the number of recursive function calls for large clusters. The stack space requirements for a purely recursive method have been shown to grow unreasonably large as the lattice size grows [12].

The second general class of adjacency-based cluster identification algorithms is the iterative type. The work of [14] is generally regarded as the first instance of an iterative adjacency-based cluster identification algorithm. This method labels all clusters in a lattice through forward propagation. The lattice is traversed row-wise, and a cluster site is assigned the label belonging to any previously labeled neighboring cluster site. Because this method does not label an entire cluster at

once like the recursive method, an additional data structure must be maintained to keep track of the cases where multiple previously disjoint clusters are discovered to be the same cluster as the lattice continues to be traversed. In these cases, a single cluster will have multiple labels associated with it, and the additional data structure essentially maintains a table of equivalence classes that can be used to assign each cluster a single unified label in a second pass of the lattice.

The Hoshen-Kopelman (HK) algorithm (presented in further detail in Part 2) can be viewed as a variation on this iterative method [9]. The main advantage of HK over the method in [14] is its use of the efficient UNION-FIND algorithm to maintain the set of cluster label equivalence classes [17]. Additionally, HK (or any iterative method) exhibits much better spatial and sequential locality than recursive methods, no longer forcing the entire lattice to be present in memory at once. Originally developed for use in percolation analysis, HK remains a standard in that field [13], [15].

Given the adjacency-based nature of cluster membership in the HK algorithm, it is quite suitable for identifying homogeneous regions in landscape raster maps within the field of landscape ecology [2]. The HK algorithm has previously been implemented using a finite state machine (FSM) to improve upon its performance, but that implementation is limited to a neighborhood rule in which only the four cardinal neighbors are considered to be connected to a point in the map [4]. Due to artifacts introduced by the process of rasterizing a

landscape map, it is often desirable to consider a point in a map to be connected to its nearest eight neighbors [11]. Thus forms the motivation for developing an efficient finite state machine HK implementation using the nearest-eight neighborhood rule, as described in Part 3. Additionally, Part 3 contains a discussion of the relation between HK and the UNION-FIND algorithm, as well as how the UNION operation can be adapted to consider cluster size in order to boost the overall performance of an implementation of the HK algorithm.

Part 4 presents performance evaluations of this FSM HK implementation on a workstation. The performance of the FSM is compared to that of a straight-forward implementation of the HK algorithm, and the effects of implementing the UNION operation based on cluster size operation are explored. Two basic data types are used in these performance evaluations: randomly generated lattices, similar to what one might expect in Monte Carlo simulations for percolation analysis [15]; and landscape maps of varying sizes. In fact, early versions of this implementation have already proved useful in two applications involving landscape map analysis [6], [8].

Another application proposed in this study is cluster analysis using limited capability computational devices like Palm personal digital assistants. One such hypothetical application may involve a field researcher who wishes to perform cluster analysis on a map of the surrounding area, without access to a more powerful computer. While the Palm testbed architecture used in this study (Part 5)

proves unsuitable for this implementation, the underlying cause for the problem is explored and a potential solution is proposed.

2 Hoshen-Kopelman Algorithm Background

This part includes a discussion of background information on the Hoshen-Kopelman (HK) algorithm and previous research involving the algorithm. HK as it was originally defined is presented first, followed by an overview of independent research involving a finite state machine implementation.

2.1 Original HK Algorithm

The Hoshen-Kopelman algorithm is a single-pass, aggregate-type cluster identification algorithm [9]. Unlike many other cluster identification algorithms, HK does not utilize a distance or similarity metric in determining cluster assignments. Rather, as HK was originally developed for use in percolation analysis, cluster membership is defined by adjacency.

The clustering space considered by HK is defined using a lattice structure. While the example presented in the next section uses a square lattice, the HK algorithm can be applied to other lattice structures such as triangular lattices, double-triangular lattices, and so forth. The sites within the lattice are binary valued, referred to here as either type *A* or *B*, with type *A* sites being targeted for clustering. Each *A* site is assigned a cluster label from a set of natural numbers, while *B* sites are always labeled with zeros. A single cluster of *A* sites may be identified by multiple labels. Such multiple label instances occur upon the

discovery of a pathway linking two previously disjoint clusters. When a cluster is identified by multiple labels, the smallest of those labels is regarded as the *proper* label. While the proper labels for coalesced or merged clusters may change, the labels already assigned to sites within previously disjoint clusters remain unchanged. To track labels and cluster merges a separate list N is maintained, which records the links among all clusters and their proper labels. Using this technique allows for a single-pass approach, however it is possible, if desired, to perform a second pass through the lattice to reassign each site its proper label. This is useful if one wishes to save the cluster identification results without also saving N. Also note that this algorithm can be performed in-place, overwriting the source data in the lattice with the resulting cluster labels.

The pseudo-code for assigning a label to a site s_i using HK is given in Figure 2-1 (all figures are located in the appendix). HK can traverse the lattice either row-wise or column-wise (for a two-dimensional lattice structure). When a site of type B is encountered, it is assigned a value of 0. When a site of type A is encountered, previously labeled sites within the *neighborhood* of s_i are searched. The neighborhood of s_i consists of all its *neighbors*, or adjacent sites as defined by the lattice structure. If there are no A neighbors found, s_i is assigned the smallest unused label, determined by the label counter k and increment Δk (the increment is usually defined as 1). If A neighbors are found, N is referenced to determine those neighbors' proper labels. In the case where all A neighbors have the same

proper label, indicating that those sites are already known to belong to the same cluster, site *i* is assigned the same proper label. However, when multiple proper labels are found among the *A* neighbors, a cluster merge operation must be performed. The smallest proper label becomes the new label for s_i and the newly coalesced cluster. The list *N* is also updated to reflect the new size of the coalesced cluster and to link the larger—and no longer proper—labels among the *A* neighbors to the new unifying proper label.

The pseudo-code for determining the proper label of a neighbor site s_n is given in Figure 2-2. The temporary variables *r* and *t* are used to follow the label reference chain in list *N*. When *t* is first assigned the value of $-N(r=s_n)$, it is checked for a negative value. If *t* is negative, then the value of $N(r)$ is positive, indicating that s_n is already assigned a proper label value. However, as long as *t* remains positive, the reference chain must be followed. Once the end of the chain is reached and the proper label is found, the entry in list *N* for s_n is updated to refer directly to the proper label.

2.2 HK Example

Consider the 8×8 square lattice in Figure 2-3. Each site in the lattice located at row *i* and column *j* has neighboring sites at *(i-1, j)*, *(i, j-1)*, *(i+1, j)*, and *(i, j+1)*. In the notation used here, row and column indexing begins at 0, and the origin is located in the top-left corner of the lattice. A value of -1 is denotes an *A* site,

while 0 denotes a *B* site. After being processed by the HK algorithm, traversing row-wise across the lattice, the resulting label assignments are shown on the left side of Figure 2-4. The right side of Figure 2-4 shows the contents of the first nine elements of list *N*.

After processing just the first row of the lattice, cluster 1 contains two sites, cluster 2 one site, and cluster 3 two sites. The third row exhibits the first instance of cluster coalescence. When the site at (2, 1) is first encountered, neither of the two previously examined sites at *(i-1, j)* and *(i, j-1)* belong to *A*, and the site is therefore assigned a label of 5. However, the subsequent site at (2, 2) has two *A* neighbors: (1, 2) with label 4, and (2, 1) with label 5. When clusters 4 and 5 coalesce, the smaller of the two labels becomes the new proper label, which is reflected in *N*. After the row has been processed, *N*(5) has value -4, and *N*(4) shows that cluster 4 has three member sites. Figure 2-5 shows the label-assigned lattice following an optional second pass to assign each *A* site its proper cluster label.

2.3 Nearest-Four FSM

The implementation of HK presented in [4] utilizes a finite state machine to achieve improved performance and is the basis for the original research presented in this paper. However, one fundamental difference between this FSM implementation and that presented in Part 3 is the use of the *nearest-four*

neighborhood rule. This neighborhood rule is analogous to the square lattice structure presented in Section 2.2, but henceforth we shall use a new terminology. Whereas the original HK specification in Section 2.1 referred to lattices and sites, the following text refers to *matrices* and *cells*. The matrix contains the data on which cluster identification is performed, and each data element within the matrix is a cell. For convenience, we shall refer to the nearest-four neighboring cells using the cardinal directions north, south, east and west, as shown in Figure 2-6.

This nearest-four FSM implementation uses two data structures: `matrix` and `csize`. `Matrix` is a two-dimensional integer array of size $n \times m$, where n is the number of rows and m is the number of columns of the matrix being analyzed. As with the original HK algorithm, this FSM implementation expects binary-valued input data. Prior to initiating the FSM HK algorithm, `matrix` is preprocessed to filter out irrelevant data. Each matrix cell belonging to the target class, or cell value of interest, are reassigned a value of -1. All non-target class cells are reassigned a value of 0. As the algorithm proceeds to analyze the matrix, those cells with a -1 value are given cluster labels as appropriate.

The second data structure, `csize`, is a one-dimensional integer array analagous to the list N in the original HK definition. This array serves a dual purpose—to track the size of clusters, along with cluster mergers as interconnecting pathways between previously independent clusters are

discovered. The `csize` array is indexed from 1, with each value initialized to 0. The value at index *i* corresponds to the cluster with label *i*. A positive value at index *i* indicates the size (number of members) of that cluster. A negative value is the result of cluster merging, and its absolute value is an index to the true cluster label. Note that such redirection is not limited to a single level; negative values may follow any non-circular, finite chain. Unlike in the original HK specification, however, this implementation performs *path compression* on the `csize` array. During the process of merging multiple clusters, every point in the label reference chain of those clusters is updated to refer directly to the new proper label. This effectively reduces the amount of indirection between a temporary label and its proper label to a single level and is key in improving the overall performance of the implementation.

The finite state machine in this implementation is used to encapsulate the cluster membership status of the west neighbor (previous cell) and the northwest neighbor (previous cell's north neighbor). The encapsulated information for each state in the FSM is shown in Table 2-1 (all tables are located in the appendix). When the FSM is in state s_0, the west neighbor is known to have a value of 0. In this case, a current cell value of -1 indicates that a new cluster is formed if the north neighbor is 0, or added to the north neighbor's cluster if that neighbor is nonzero. When in state s_1, the west neighbor is known to have a nonzero value, denoting cluster membership, while the northwest neighbor's value is 0. Here a

current cell with value -1 will be added to the west neighbor's cluster if the north neighbor is 0, but when the north neighbor is nonzero, a possible merge could take place between the north and west neighbor clusters. The reason for encapsulating the northwest neighbor's value—even though the northwest neighbor is not directly connected to the current cell according to the nearest-four neighborhood rule—becomes evident with state s_2. When the FSM is in this state both the west and northwest neighbors are known to have nonzero values and belong to the same cluster. If the current cell is -1 and the north neighbor is also nonzero, it is now known that no cluster merge must take place, as the north and west neighbors are connected by the northwest neighbor. Thus the current cell is simply given the proper label for the west-northwest-north cluster, and no time is spent determining whether a merge operation is necessary. The final two states, s_3 and s_4, indicate that the end of a row or the end of the matrix has been reached, respectively.

This implementation is reported to achieve a speedup factor in the range of 1.39 to 2.00 over the implementation of HK used in [2], based upon the original HK specifications. This is due to a combination of path compression in the `csize` array, encapsulating neighbor information in a state variable, and avoiding unnecessarily checking for potential cluster merge situations due to such encapsulation.

3 Nearest-Eight FSM

The original motivation for an efficient nearest-eight neighborhood finite state machine implementation of HK was provided by research involving a tool to analyze landscape maps and detect potential dispersal corridors, or pathways, among suitable habitat regions for various wildlife [8]. Habitat fragmentation is a major focus of landscape ecology [18], and the identification of distinct habitat clusters (or *patches*, in landscape ecology nomenclature) is a necessary step for analyzing the effects of fragmentation. However, the common practice of representing a landscape map using a raster model can introduce a certain error or ambiguity in terms of connectivity [11]. For instance, a landscape feature that moves diagonally across the frame of reference may have its continuity disrupted if one considers only connectivity along cell edges in a matrix (i.e., using the nearest-four neighborhood rule). See Figure 3-1 for an example of this rasterization artifact. While the landscape feature in this example—say, a road or stream—should be considered contiguous, cluster identification using the nearest-four neighborhood rule will report six unique clusters. To overcome this problem, we use the nearest-eight neighborhood rule, as shown in Figure 3-2.

The nearest-eight finite state machine implementation works similarly to the nearest-four method described in Part 2, with the following exceptions. If the current cell needs to be added to a cluster, the north and west neighbors' values

must be checked, along with the northeast and northwest. If any one of these neighbors belongs to a cluster, the current cell is added to that cluster. If the north neighbor does not belong to a cluster, but both the northeast and either the west or northwest neighbors do, there may be a need to perform a cluster merge (similar to west-north neighbor merges using the nearest-four neighborhood rule).

Given that the number of neighbor comparisons is effectively twice that of the original HK method, this seems to be a prime candidate for performance gains by using a FSM. When the cell currently under examination needs to be clustered, four neighbors need to be checked. Note that the north and northeast neighbors of the current cell are the northwest and north neighbors of the next cell, respectively. If these values are known, they may be encapsulated in the next state, reducing the number of neighbor comparisons by two if the next cell also belongs to a cluster. Additionally, the current cell's value is always checked and thus may be encapsulated in the next state.

3.1 State Definitions

Figure 3-3 presents the seven possible states based on known and unknown neighbor values. State s_0 represents the case when only the west neighbor's value is known. States s_1 through s_4 represent the cases when the previous cell belongs to a cluster, and therefore the values of all three of the west, northwest, and north neighbors have been checked and are known. States s_5 and s_6 represent the cases

when the previous cell is not a part of a cluster, but the cell prior to that is. In these cases, only the west and northwest neighbors have been checked and are known. One such case is illustrated in Figure 3-4. Part A of Figure 3-4 shows a matrix segment with all cell values revealed, while parts B through D show the state and encapsulated neighbor information as the FSM processes the bottom row. In part B, the FSM is in state s_2 and both the north and northwest values are known. As the current cell in part B belongs to the class being clustered, the northeast neighbor's value must be checked, and this information is encapsulated in the state for the next cell, s_4. The current cell in part C does not belong to a cluster, so the northeast neighbor's value is not checked and thus will not be available in the state for the next cell. However, the value of the north neighbor, also the subsequent cell's northwest neighbor, is known at this point and can encapsulated in the next state, s_5.

Figure 3-5 shows the formal definition of this FSM, but there are several deviations from this definition in the actual implementation. Note that the final state, s_7, is not shown in Figure 3-3. While the formal definition requires an end-of-input marker at the end of the input matrix, the implementation does not utilize such a marker. Rather, simple bounds checking determines when the end of input is reached. The first row of the matrix is handled as a special case, because there are no north neighbors for that row's cells. Additionally, bounds checking is used to determine when the end of a row is reached, and the final cell is handled as a

special case, due to the lack of a northeast neighbor for that cell. When the FSM proceeds to the beginning of the next line, the state is reset to s_6, which forces the FSM to assume that the west and northwest neighbors are zeros, thus avoiding references to those nonexistent neighbors.

It is possible, however, to implement the FSM strictly following the formal definition. The insertion of a buffer row of zeros at the beginning of the matrix would eliminate the necessity of treating the first row of data elements as a special case. Also, a buffer column of zeros on both sides of the matrix would allow the FSM to proceed from one row to the next without explicit bounds checking, as the FSM would reset to s_6 by its natural progression. However, using either approach is ill-advised, due both to increased storage space requirements and to the additional overhead associated with retrieving buffer elements from memory.

Maintaining the state can be done explicitly by using an integer state variable, as is the case with the nearest-four FSM implementation from Part 2. Alternatively, the state may be implicitly maintained by segmenting the code by state and jumping to the appropriate code segment as the state changes, which is the method used in this nearest-eight FSM implementation. The program is written in C, and `goto` statements are used to transition from one state to another. While such a practice can harm program locality and impair predictive branch execution, the `goto` method has shown improved performance, as it eliminates the necessity of checking and possibly setting a state variable at each

cell in the matrix.

3.2 Relation to UNION-FIND Algorithm

As defined in [5], a *disjoint-set data structure* maintains a collection of non-overlapping sets of objects, and each set is identified by a single *representative* object contained within the set. The representative may change as the set is altered, but the representative must remain the same as long as the set is unaltered. A *disjoint-set forest* is an implementation of the disjoint-set data structure that represents sets by rooted trees. Each node in the tree contains one member and points only to its parent node, and the root of the tree is the representative for the set.

Three functions provide useful manipulations of a disjoint-set data structure: MAKE-SET(x), UNION(x, y), FIND-SET(x). The MAKE-SET function creates a new set whose only member object is x. The UNION function combines the two sets containing objects x and y. Finally, FIND-SET returns the representative of the set containing x. An algorithm that performs these operations on a disjoint-set data structure is known as a UNION-FIND algorithm.

The `csize` array (or list N in the original HK specifications) may be viewed as a disjoint-set forest. Each set is a cluster, and the objects contained in the sets are cluster labels. While a cluster may contain multiple labels, only the proper label is considered the representative. Figure 3-6 illustrates a sample `csize` array

and its graphical representation as trees. The root node of each tree is the proper label of the represented cluster.

HK is an example of a UNION-FIND algorithm. When a cell being examined belongs to the class being targeted for clustering and none of its previously examined neighbors belong to a cluster, then that cell is assigned the next unused cluster label and `csize` is updated to reflect the new cluster. This satisfies the MAKE-SET requirement of a UNION-FIND algorithm. When a linking pathway is discovered between two previously disjoint clusters, the two must be merged. This implementation of HK performs this task in a function called MERGE. The MERGE function first finds the proper cluster label for each of the two clusters by following the label reference chain in `csize`. Then, if the two proper labels are different—indicating that the two clusters are, in fact, disjoint—MERGE updates `csize` to redirect the proper label of one cluster to the proper label of the other. This satisfies both the UNION and FIND-SET requirements of a UNION-FIND algorithm.

Two heuristics may be used to improve the performance of the UNION-FIND operations on a disjoint-set forest: *path compression* and *union by rank*. Path compression is used during the FIND-SET operation to set each node's parent pointer directly to the root node or representative. To accomplish this, the FIND-SET becomes a two-pass method. The first pass follows the path of parent pointers from parameter node *x* to the root node, while the second pass traverses

back down the path to set each node's parent pointer directly to the root node. The nearest-four FSM HK implementation from Part 2 uses path compression, as does this nearest-eight FSM HK implementation within the MERGE function, which includes the FIND-SET operation.

The second heuristic, union by rank, affects the UNION operation, making the root node of the smaller of the two trees point to the root node of the larger tree. Rather than explicitly tracking the size of each tree in a disjoint-set forest, a separate *rank* value is maintained for each node. The rank is an upper bound on the height of the node. When a singleton tree is created with MAKE-SET, the single node begins with a rank of 0. When two trees are passed to the UNION function, the root node with higher rank becomes the parent of the root node with lower rank, but the rank of both root nodes remains unchanged. If the rank of both root nodes is equal, one root node is chosen arbitrarily to become the parent node, and its rank is incremented by 1. These tie-breaker situations are the only times when a node's rank is changed.

None of the HK implementations considered in this study apply union by rank. The original HK specification merges the cluster with the larger proper label into the cluster with the smaller proper label. Similarly, the nearest-four FSM implementation makes arbitrary decisions when merging two clusters. However, the MERGE function in this nearest-eight FSM implementation does make informed decisions when merging two clusters, which we shall call *union by*

cluster size. Recall that the `csize` array is viewed as a disjoint-set forest, and each node of a tree is a cluster label. Each tree represents a cluster, which may have multiple labels associated with it. While union by rank would make the tree with more nodes—or the cluster with more labels associated with it—the parent of the tree with fewer nodes, union by cluster size sets the new parent as the tree whose represented cluster size is larger than the tree whose represented cluster size is smaller. While this method may cause the runtime for a UNION operation to perform worse than it would using union by rank, it is important to consider the cost of relabeling cells in the matrix during the second pass relabeling phase. Furthermore, since the `csize` array tracks the size of clusters as they are formed, the need for an additional array to track node ranks is avoided. See Part 4 for runtime comparisons between an implementation with union by cluster size and one without.

For an example of how union by rank and union by cluster size can produce different results, refer again to Figure 3-5. If clusters 2 and 6 were to be merged using union by cluster size, the new proper label and root node would be 6, even though the tree for cluster 2 is larger than that for cluster 6. Thus, only eight cells previously belonging to cluster 2 would have to be relabeled during the second pass. On the other hand, union by rank would merge cluster 6 into cluster 2, requiring eleven cells previously belonging to cluster 6 to be relabeled.

3.3 Alternative Implementations

While all discussion of the HK algorithm to this point has involved the identification of a single cluster type in a given pass, it is possible to implement the algorithm in such a way as to identify multiple cluster types in a single pass. Such an implementation comes with a few caveats, however. Regardless of whether a finite state machine is used, identifying multiple clusters in a single pass precludes the option of performing the cluster label assignments in-place. When the cluster label assignments are stored in the original matrix, it becomes impossible to determine whether a nonzero neighbor value indicates that the neighbor belongs to the same cluster type as the current cell, because the neighbor's original value has been overwritten. Therefore a second matrix must be allocated to store the results so that the original cells' values may be retained.

Multiple concurrent finite state machines may used to identify clusters of multiple types in a single pass, with each FSM maintaining a separate state. At each cell in the matrix, every FSM must examine that cell and make its own state transitions and action individually. The drawback to this method is that state variables must be maintained instead of having separate blocks of execution like the FSM implementation presented in this study. One possible method for overcoming this hurdle is to spawn each individual FSM as a separate process or system thread.

The HK algorithm may parallelized for execution in a parallel computing environment by segmenting the data into distinct regions [1]. Each data segment can then be clustered by the algorithm on separate processors without memory sharing. It is important that each data segment has a unique region of available cluster labels to ensure that no two individual data segments contain overlapping labels. The results for these individual segments can then be sent to one master processor, which performs cluster merging along the borders of adjacent data segments. The overhead for such communication among processors can be great, however, and tests have shown that the benefits of such parallelization are very slim, if even existent. However, the benefit may be increased as the data space becomes larger, much larger than the test data used to evaluate the FSM implementation's performance in the next part.

4 Workstation Performance

An appropriate application of the FSM HK implementation may involve very large datasets or many repetitions of cluster analysis of constantly changing data, as is often the case in ecological or other models utilizing landscape maps. This part presents performance tests conducted in an environment suitable to such applications and demonstrates the clear advantage of using the FSM HK implementation.

4.1 Methodology

The tests described here were performed on a Linux workstation with a 2.4 GHz Intel Xeon processor. The system contains 8 KB L1 data cache, 512 KB L2 cache, and no L3 cache.

The *target class density* metric is simply the percentage of cells within a matrix that belong to the class being targeted for cluster identification. Performance is evaluated as *cells processed per millisecond*, where *cells* is a count of target and non-target class cells alike (i.e., the total size of the matrix). The time measurements are taken as wall-clock time (as opposed to CPU time) in order to capture the effect of delays caused by data element accesses from memory. The processing time is measured over what is considered a complete cluster analysis, which includes the first pass of temporary cluster ID assignments

and the second pass of final cluster ID relabeling. However, time taken for file I/O and data structure initialization is not included. For the plots in this section that display the number of cells processed per millisecond, each point is the mean of forty observations of a particular implementation and parameter set. The error bars in these plots represent the standard deviation.

These tests compare the FSM and non-FSM implementations of the HK algorithm. Additionally, two separate FSM implementations are examined: one utilizing the MERGE method discussed in Section 3.2 that makes the larger of two merged clusters the parent, referred to below as *proper merge*; and a version that arbitrarily decides which of two merged clusters becomes the parent, referred to as *lazy merge*. When comparing the FSM and non-FSM implementations, both use proper merge.

In addition to the processing rate comparison between the FSM and non-FSM implementation, figures are provided for the number of merges and relabel operations at each target class density level. Note that the term “merge check” in in these figures refers to any possible merge situation, such as when the cell currently under examination and both its northeast and west neighbors all belong to the target class. However, only when the northeast and west neighbors in this situation have different cluster IDs does an “actual merge” take place. Also note that the number of merge checks and actual merges is constant among all implementations: FSM or non-FSM, lazy merge or proper merge. The only

difference arises between the lazy and proper merge methods in terms of which of the two clusters becomes the parent cluster. This in turn affects the number of relabeling operations necessitated in the second-pass.

4.2 Test Data

Four datasets are used in the following tests. The first dataset consists of nineteen randomly generated 5000×5000 binary-valued matrices, one for each of d={0.05, 0.1, 0.15, ..., 0.95}, where d is the target class density. Thus, each cell in the matrix is assigned a value of 1 with d probability and a value of 0 with (1-d) probability. While this simple stochastic method is not guaranteed to produce a matrix with a target class density of precisely d, all matrices used in this test fell within 0.02% of d. Figure 4-1 shows two of the generated matrices, for d=0.25 and d=0.7.

The second dataset is a land cover raster map of five counties surrounding Fort Benning in Georgia [6]. This is represented as a 2771×2814 matrix with fifteen target classes. These fifteen classes, along with their densities and the number of identifiable clusters for each are given in Table 4-1. The land cover map also contains a sixteenth "no data" class, which defines the border surround the five counties in question. However, as this class is simply used as padding in the minimum bounding rectangle for the five counties area, it is not considered in the tests below. Thus, the total of all target class densities given in Table 4-1 is

somewhat less than 1. This landscape map, with all fifteen target classes, is shown in Figure 4-2.

The third dataset is a land cover raster map centered on the Tennessee Valley and southern Appalachian Mountains, covering portions of Tennessee, Alabama, Georgia, North Carolina, and South Carolina [8]. This 4300×9891 map contains twenty-one classes. Unlike the Fort Benning landscape map, this map is fully populated, without a "no data" class. Table 4-2 gives the target classes, densities, and number of clusters. This landscape map is shown in Figure 4-3.

The fourth and final dataset is another land cover map, covering a portion of Yellowstone National Park [8]. This 400×500 map is much smaller than all the previous maps and contains six classes. As with the Tennessee Valley map, this map is fully populated with target class values. Table 4-3 gives the target classes, their densities and number of clusters, and the landscape map is shown in Figure 4-4.

4.3 FSM and non-FSM Performance Comparisons

Figure 4-5 shows the performance of both FSM and non-FSM implementations on the randomly generated 5000×5000 matrices. The FSM implementation exhibits a clear performance boost over the non-FSM version across all target class density levels. The least improvement is observed for d=0.05, with the FSM implementation performing at 128,600 cells per

millisecond and the non-FSM at 125,000. This is explained by the infrequency and wide dispersion of the target class cells across the matrix. When target class cells are fewer and farther apart, the FSM is unable to retain knowledge of previously examined neighbors less often, and the FSM advantage is diminished. Conversely, the FSM advantage increases as the target class density approaches maximum. The most separation between the two implementations is observed at d=0.95, with the FSM performing at 100,040 cells per millisecond and the non-FSM at 69,137. With such a high target class density, the FSM is able to retain neighbor value knowledge much of the time, thereby significantly reducing the number of memory references.

The performance for both FSM and non-FSM implementations is worst near 0.45 target class density. This corresponds to the number of merge and relabel operations shown in Figures 4-6 and 4-7, respectively, reaching their maximum levels in approximately the same density range.

The FSM and non-FSM performance comparison on the 2771×2814 Fort Benning landscape map is given in Figure 4-8. As with the randomly generated matrices, the smallest margin of improvement is observed at the lower target class density levels, while the largest margin is observed at the highest target class density. At d=0.00005 (class 7), the FSM processes 135,375 cells per millisecond, while the non-FSM is competitive at 130,395. On the other end of the spectrum, when d=0.20648 (class 42), the FSM outperforms the non-FSM at 85,782 cells

per millisecond to only 73,148.

Figure 4-9 shows the performance comparison for the 4300×9891 Tennessee Valley map. These results follow the same pattern as the with the Fort Benning map. The smallest margin of improvement for the FSM over the non-FSM implementation is observed at the low density levels. At d=0.00168 (class 10), the FSM processes 135,225 cells per millisecond while the non-FSM processes 130,752. The largest margin of improvement occurs at d=0.22324 (class 1), with the FSM implementation processing 117,196 cells per millisecond versus the non-FSM implementation at 98,153 cells per millisecond. This target class is also interesting in that it breaks the trend of generally monotonically decreasing processing rates as the target class densities increase. To see why this is the case, refer to Table 4-2 and Figure 4-10. Class 1 has only 549 clusters—three orders of magnitude lower than either of the two target classes—19 and 17—with similar density levels. This corresponds to the significantly lower number of merge operations (both merge checks and actual merges) for class 1. Figure 4-11 shows the Tennessee Valley map with only classes 19, 1, and 17 visible in succession.

The final workstation performance comparison between the FSM and non-FSM implementations is shown in Figure 4-12, using the Yellowstone dataset. This test case follows the same general pattern as before. The smallest margin of improvement is once again at a small density level—d=0.00038—with the FSM processing 132,669 cells per millisecond versus the non-FSM at 124,804.

Likewise, the largest margin of improvement is at d=0.36626. Here the FSM processes 92,059 cells per millisecond while the non-FSM processes only 66,833. As it happens, the number of merge and relabel operations on this map is constant across all target classes.

One notable facet of the Yellowstone is that the map is very small compared to those in the previous tests. The implementations examined here store the matrix using 4-byte integers, resulting in just under 2 KB per row for this 500 column matrix. Recall that the HK algorithm, using the nearest-eight neighborhood rule, will examine at most the northwest, north, northeast and west neighbor values for any given cell. When the matrix is stored in a row-major format, as it is here, all possible neighbor references fall within the previous *ncols*+1 matrix elements, where *ncols* is the number of columns in the matrix. Thus, for the Yellowstone map it is possible, though not guaranteed, that all four relevant neighbors at any given time are present in the 8 KB L1 cache in the testing environment used here. Failing that, it is likely those data elements are present in the 512 KB L2 cache. Though the cost of accessing the four relevant neighbor values is therefore decreased substantially, the FSM nonetheless exhibits a performance boost by encapsulating neighbor values in the state.

4.4 Lazy and Proper Merge Performance Comparisons

To demonstrate the significantly detrimental effect that a lazy implementation of the MERGE method can have on the performance of HK, some comparisons between FSM HK implementations with lazy and proper merging are presented here, using the randomly generated matrices and the Tennessee Valley landscape map. The relative performances of the non-HK implementations with lazy and proper merging are quite similar and thus are not presented here.

Figure 4-13 shows the processing rates of two implementations using lazy and proper merge. At the lowest density, d=0.05, the implementation with lazy merging slightly outperforms the proper merging method. This is to be expected, as the low density of the target class means that there are relatively few second-pass relabeling operations taking place for either implementation, as can be seen in the previously referenced Figure 4-7. The proper merge implementation involves a slight overhead incurred by checking the size of each cluster being merged, and this overhead is not fully and consistently mitigated for the smallest density target classes in any of the tests discussed here. However, as the target class density increases the separation between two merging methods becomes very great, corresponding to the reduced number of second-pass relabeling operations.

The relative performances of lazy and proper merge on the Tennessee Valley landscape map can be seen in Figure 4-14. As with the randomly generated matrices, the lazy method is competitive with the proper method for very small target class densities. In this case the lazy method never actually outperforms the proper method, though the advantage of proper merge is often negligible. Figure 4-15 shows the number of second-pass relabeling operations necessitated by each of the two merging methods. As the target class density increases, the separation between the two becomes significant.

5 Palm Device Performance

In addition to applications that would naturally be suited to a fixed-location, workstation computing environment, it is posited that the FSM HK implementation could be useful in low-powered, embedded, or mobile computing environments. One can imagine, for instance, researchers wishing to perform cluster analysis on landscape data collected in-field via manual observations or GPS on a lightweight computing device. While the FSM implementation does not outperform the non-FSM implementation in the computing environment used in the following tests, the underlying causes of its lackluster performance are explored, and potentially suitable hardware specifications are discussed.

5.1 Methodology

The tests described in this part were performed on a Palm IIIxe personal digital assistant. This mobile device, running Palm OS 3.5.3, contains a 16 MHz Motorola Dragonball 68328EZ processor with 8 MB RAM and no cache.

Due to the memory structure imposed by the Palm operating system, no single allocated data region, or *chunk*, may exceed approximately 64 KB. This severely limits the size of the test data used here. While it would be possible to segment a matrix across multiple chunks, perform cluster analysis on each segment and then merge the results, it was decided to forgo this option to avoid

the overhead incurred by additional merge and relabel operations. Even with the small datasets used here, it sometimes took as long as twenty seconds to perform a single cluster analysis.

While the results in Part 4 were presented in terms of cells processed per millisecond, the Palm device processing rates are reported as cells processed per second. While Palm OS does not allow programmer access to the underlying system clock, and the current time is only accessible at finest granularity of one second, the OS generates "system ticks" one hundred times per second. As this is the most precise timing method available, the time measurements used here are based on these system ticks. While it is not guaranteed that the number of system ticks per second is constant, extensive testing has not exposed any variation in this rate.

While the workstation tests in Part 4 were performed forty times each to compute the mean running time for each implementation and target class density, each implementation on the Palm is executed exactly once for each target class in each dataset. Due to the single-threaded nature of the Palm OS and the lack of system or I/O interrupt handling in these HK implementations, the run time is observed to be constant within one system tick for each implementation on any given dataset and target class.

5.2 Test Data

Two datasets are used in the following tests. The first is a series of randomly generated binary matrices, similar to those described in Section 4.2. These matrices are 150×150, with one matrix for each of d={0.05, 0.1, 0.15, ..., 0.95}.

The second dataset is a segment from the Fort Benning landscape map described in Section 4.2 and is shown in Figure 5-1. This 175×175 matrix has thirteen target classes, and the densities and number of clusters of each class is shown in Table A-4.

5.3 FSM and non-FSM Performance Comparisons

Figure 5-2 shows the performance of both FSM and non-FSM implementations on the randomly generated 150×150 matrices. The non-FSM implementation clearly outperforms the FSM implementation for small target class density values. At d=0.05, the non-FSM version processes 13,975 cells per second, while the FSM processes only 12,640 cells per second. However, as the density level increases the non-FSM advantage is diminished. When d=0.3, the non-FSM outperforms the FSM at 2,577 cells per second to a very close 2,508 cells per second. While this gap is similarly narrow to the tail end of the graph, the FSM never actually reaches a point where it outperforms the non-FSM by any amount. As an aside, the tail end of the graph never increases, as was the case with the randomly generated matrices on the workstation in Part 4. This is an

artifact of the Palm OS requirement that a system call, which performs bounds-checking, be used to write to any area of memory except for the stack or heap. Because the memory architecture requires that our matrix be stored in a more permanent area of memory (analogous to a hard drive or secondary storage), this slow method must be used when assigning class membership and relabeling.

The results of the second test on the Fort Benning landscape map segment are shown in Figure 5-3. As in the previous test, the non-FSM implementation has a clear advantage over the FSM for small target class density values, but that advantage wanes as the target class densities increase. At d=0.00235 (class 20), the non-FSM processes 40,833 cells per second while the FSM processes only 31,572 cells per second. As before, there is never a point where the FSM implementation outperforms the non-FSM.

5.4 Further FSM Performance Analysis

The reason for the lackluster performance of the FSM can be explained by the Palm IIIxe system specifications. While the FSM might be expected to perform better than the non-FSM by significantly reducing the number of data element memory accesses, thereby mitigating the negative effects of a lack of cache memory between the main RAM and CPU registers, it is ironically this very lack of any cache that hinders the FSM. The non-FSM does indeed retrieve data elements from memory more often, but its program code follows a much more

predictable path, with far fewer possible branches. This allows the CPU to pre-fetch sequential instructions much more successfully than with the FSM code. With no cache to store program code, the penalty for incorrectly predicting code branches and pre-fetching the wrong instruction is much greater, because this stalls the CPU pipeline while waiting for the correct instruction from main memory.

This effect can be demonstrated with the example C code segment in Figure 5-4. This is a simple doubly-nested loop that accesses the one hundred element *array* in the inner loop and performs a test-and-set on another variable *k* in the outer loop. When this code is compiled with global compiler optimizations, the number of code branches is significantly reduced when compared to an unoptimized compiled version. The variables *i*, *j*, and *k* are specified as register variables in order to avoid retrieving those values from the stack every time they are referenced when not using any compiler optimizations. Furthermore, the inner loop stops on condition `j < (i/100)`, which prevents the compiler-optimized version from gaining an advantage from loop unrolling.

The relevant assembly code segment resulting from compiling without global compiler optimizations is shown in Figure 5-5, and the version produced with global compiler optimizations is shown in Figure 5-6. Though it is certainly unnecessary to dwell on the details of each of these figures, note that the instructions that cause branching are in bold typeface. In the unoptimized version

there are eight branch points, compared to only five in the optimized version. The optimized version executes in 2.51 seconds, while the unoptimized, branch-heavy version executes in a whopping 9.71 seconds. The lack of locality in the unoptimized code clearly has a negative effect on performance.

The assembly code resulting from the FSM and non-FSM implementations is too unwieldy to present here, so refer instead to Figure 5-7. This figure shows the code branches for each cell processed in the first pass of the HK algorithm. The top tree represents the non-FSM implementation. The root of the tree corresponds to the beginning of the process of examining a single cell, and each branch corresponds to a test condition. Upon reaching a leaf node in the tree, the current cell has been either added to a cluster or ignored (in the case of a non-target class cell), and the next cell is examined. At this point the root node of the tree is logically reentered. The first, single-leaf subtree represents a non-target class value for the current cell. The remaining branches represent the various neighbor-cell value comparisons.

The bottom tree in Figure 5-7 represents the logical branches in the FSM implementation. The dashed branches represent state checks, but otherwise this tree works much the same as the previous. The three-leaf subtree near the root again represents the case when the currently examined cell belongs to a non-target class. Upon reaching any leaf node, a state transition occurs and the next cell is examined, logically reentering the root node of the tree.

While the FSM logical structure is more complex than that of the non-FSM, the benefit of this complexity arises when the current state encapsulates more than just the previously examined cell's value. This situation is represented graphically in Figure 5-7 as the four two-leaf subtrees below the level of state-check branches, corresponding to four of the seven states defined in the FSM. At low target class density levels the added complexity is obviously detrimental, but the overhead of the greater number of branches is mitigated when the target class density is higher and the FSM spends increased time in the four more knowledgeable states.

Given these facts, it is reasonable to assume that the FSM could well outperform the non-FSM implementation in limited capacity devices, provided that the device has an instruction cache with enough capacity—perhaps just a few kilobytes—to overcome the ill effects of the additional code branching. Research presented in [3] shows that a simulated Palm m515—similar to the Palm IIIxe, but with a 33MHz Motorola Dragonball MC68VZ328 CPU—can have effective memory access times reduced by 50% with as little as 1-2 KB of cache. However, to date no Palm devices have been manufactured with cache memory, and no other equivalent architecture was available at the time of this study.

6 Conclusion

This study has provided an efficient finite state machine implementation of the Hoshen-Kopelman algorithm using the nearest-eight neighborhood rule. By using states to encapsulate neighbor cell information and reduce redundant memory accesses, along with informed UNION-FIND operations, this implementation clearly outperforms an implementation based upon the original HK specifications on a workstation testbed for both randomly generated and actual landscape maps. While this FSM implementation does not outperform the classic implementation on a Palm device, it is nonetheless competitive for all but very sparse target class densities. Tests indicate that a potential Palm-like device with a minimum amount of cache memory would allow this FSM implementation to perform much more efficiently.

Possible future work in this vein remains. Just as the original HK algorithm can be applied to three dimensional lattices, so could a FSM implementation. However, as the number of neighbor relationships increases, so does the number of states in a FSM adaptation. While the FSM in this study was defined with extensive manual examination and optimization, it may be possible to create a system that can define the FSM states for an arbitrarily defined neighborhood rule automatically. This would open up the possibility for researchers without programming knowledge to define custom neighborhood rules suitable to their

particular applications. Though this study had a particular focus on the analysis of landscape maps, the methods presented here should be applicable for many types of data representable in a lattice format.

Bibliography

Bibliography

[1] M.L. Aldridge. A Parallel Finite State Machine Implementation of a Nearest-Eight Hoshen-Kopelman Adaptation for Landscape Analysis. *Proceedings of the 45th Annual ACM Southeast Regional Coneference*, 391-394, 2007.

[2] M. Berry, E. Comiskey, and K. Minser. Parallel analysis of clusters in landscape ecology. *IEEE Computational Science and Engineering I*, 2:24-38, 1994.

[3] H. Carroll, J.K. Flanagan, and S. Baniya. A Trace-Driven Simulator for Palm OS Devices. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 157-166, 2005.

[4] J.M. Constantin, M.W. Berry, and B.T. Vander Zanden. Parallelization of the Hoshen-Kopelman Algorithm Using a Finite State Machine. *The International Journal of Supercomputing Applications and High Performance Computing*, 11(1):34-48, 1997.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, Second Edition. MIT Press, Cambridge, 2001.

[6] V.H. Dale, M.L. Aldridge, T. Arthur, L.M. Baskaran, M.W. Berry, M.Chang, R.A. Efroymson, C. Garten, C. Stewart, and Washington-Allen. Bioregional Planning in Central Georgia. *Futures*, 38:471-489, 2006.

[7] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*, Third Edition. Pearson Prentice Hall, Upper Saddle River, 2008.

[8] W.W. Hargrove, F.M. Hoffman, and R.A. Efroymson. A practical map analysis tool for detecting potential dispersal corridors. *Landscape Ecology*, 20(4), 2005.

[9] J. Hoshen and R. Kopelman. Percolation and cluster distribution: I. Cluster multiple labeling technique and critical concentration algorithm. *Phys. Rev. B.1*, 14 (October):3438-3445, 1976.

[10] P.L. Leath. Cluster size and boundary distribution near percolation threshold. *Phys. Rev. B*, 14(11):5046-5055, 1976.

[11] A.B. Leitão, J. Miller, J. Ahern, and K. McGarigal. *Measuring Landscapes: A Planner's Handbook*. Island Press, Washington, 2006.

[12] J. Martín-Herrero. Hybrid Cluster Identification. *J. Phys. A: Math. Gen.*, 37:9377-9386, 2004.

[13] N.R. Moloney and G. Pruessner. Asynchronously parallelized percolation on distributed machines. *Physical Review E*, 67:037701, 2003.

[14] A. Rosenfeld and J.L. Pfalz. Sequential operations in digital picture processing. *J. ACM*, 13:471-494, 1966.

[15] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*, Second Edition. Taylor and Francis, London, 1992.

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

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

[18] M.A. Withers and V. Meentemeyer. Concepts of Scale in Landscape Ecology. In J.M. Klopatek and R.H. Gardner, eds., *Landscape Ecological Analysis: Issues and Applications*. Springer-Verlag, New York, 1999.

Appendix

```
if s_i is of type B
    s_i = 0
else
    search previously labeled neighbor sites

    if no A neighbors found
        s_i = k+Δk
        N(s_i) = 1
        k = k+Δk
    else
        find proper labels K of neighboring A sites

        s_i = min(K)
        N(s_i) = 1

        foreach k in K
            N(s_i) = N(s_i) + N(k)
            N(k) = s_i
        endfor
    endif
endif
```

Figure 2-1. HK method for assigning a label to a site s_i.

```
r = sₙ
t = -N(r)
if t < 0
    return r
endif

while t > 0
    r = t
    t = -N(r)
endwhile

N(sₙ) = -r

return r
```

Figure 2-2. HK method for determining the proper label of a site s_n.

```
-1 -1  0  0 -1  0 -1 -1
 0  0 -1  0 -1  0 -1 -1
 0 -1 -1  0 -1  0 -1 -1
 0 -1  0  0 -1 -1 -1  0
 0 -1 -1  0  0  0  0 -1
-1  0 -1  0 -1  0  0 -1
-1  0  0  0  0  0 -1 -1
-1  0  0  0  0  0 -1  0
```

Figure 2-3. An 8×8 binary, square lattice input for HK. Here -1 denotes an *A* site, and 0 a *B* site.

								1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	1	0	0	2	0	3	3	2	1	2	0	0	0	0	0	0
0	0	4	0	2	0	3	3	2	2	4	1	0	0	0	0	0
0	5	4	0	2	0	3	3	2	3	6	3	-4	0	0	0	0
0	4	0	0	2	2	2	0	2	12	-2	4	-4	0	0	0	0
0	4	4	0	0	0	0	6	2	12	-2	6	-4	1	0	0	0
7	0	4	0	8	0	0	6	2	12	-2	7	-4	2	1	1	0
7	0	0	0	0	0	9	6	2	12	-2	7	-4	4	2	1	-6
7	0	0	0	0	0	6	0	2	12	-2	7	-4	5	3	1	-6

Figure 2-4. The lattice from Figure 2-3 after HK cluster identification (left) and the contents of array *N* after processing each row of the lattice (right). The integers 1 through 9 above *N* are indices for the list and also denote cluster labels.

```
1  1  0  0  2  0  2  2
0  0  4  0  2  0  2  2
0  4  4  0  2  0  2  2
0  4  0  0  2  2  2  0
0  4  4  0  0  0  0  6
7  0  4  0  8  0  0  6
7  0  0  0  0  0  6  6
7  0  0  0  0  0  6  0
```

Figure 2-5. The lattice from Figure 2-3 after the HK algorithm and the optional second pass relabeling operation.

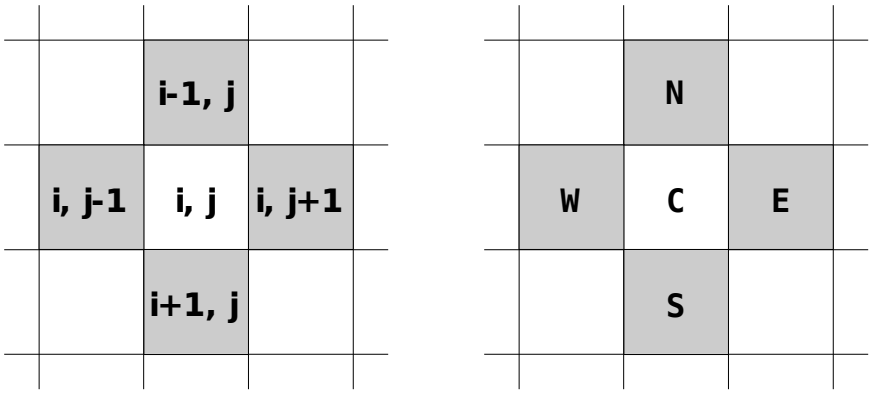

Figure 2-6. North, east, west and south relationship to cell *(i, j)*.

Table 2-1: States Defined by Nearest-Four FSM

State	Encapsulated Information
s_0	Not currently in a cluster (W == 0)
s_1	Currently in a cluster on current line (W != 0, NW == 0)
s_2	Currently in a cluster on previous line (W != 0, NW != 0)
s_3	At a map boundary
s_4	Finished

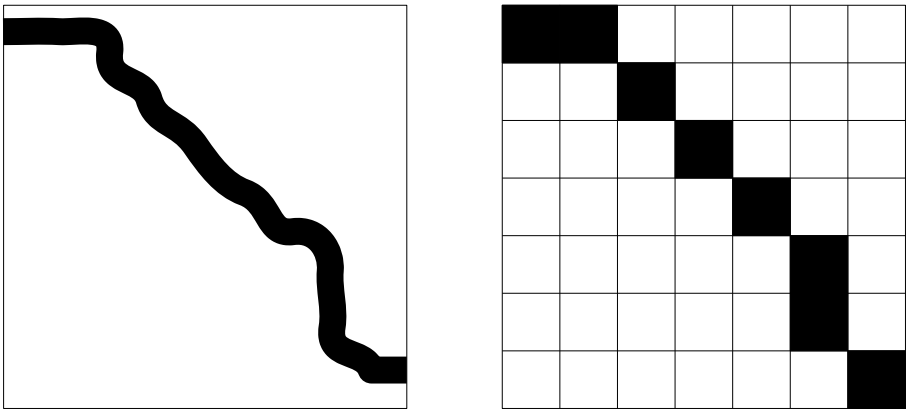

Figure 3-1. A hypothetical landscape feature (left) and its representation in a raster format (right).

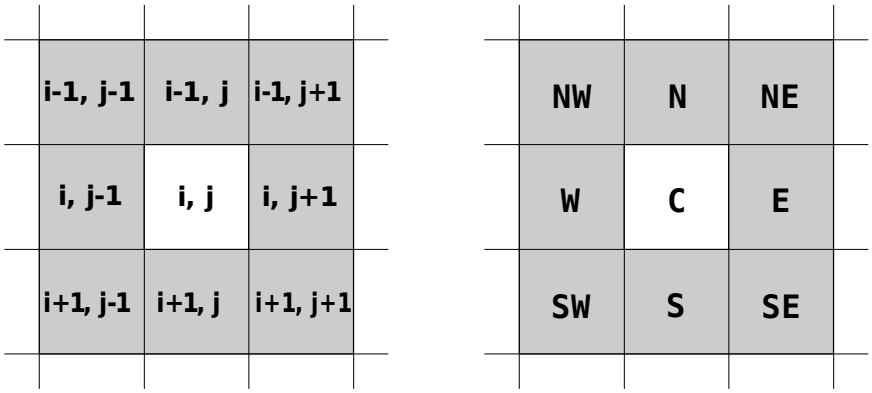

Figure 3-2. Cardinal and ordinal neighbor relationships to cell *(i, j)*.

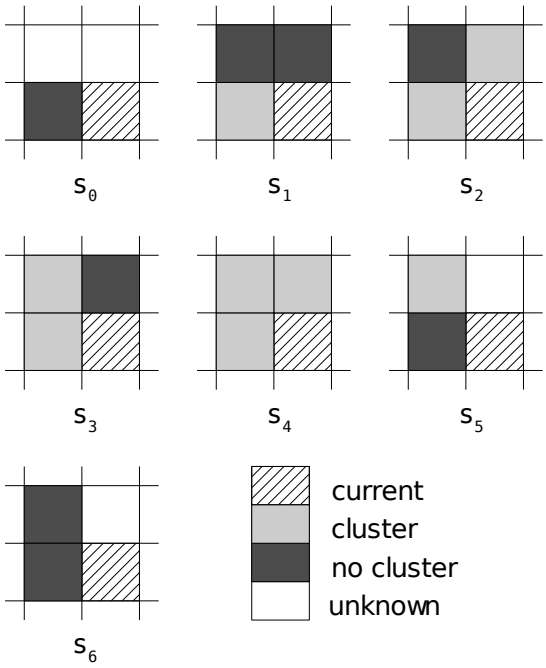

Figure 3-3. Seven states in the nearest-eight FSM.

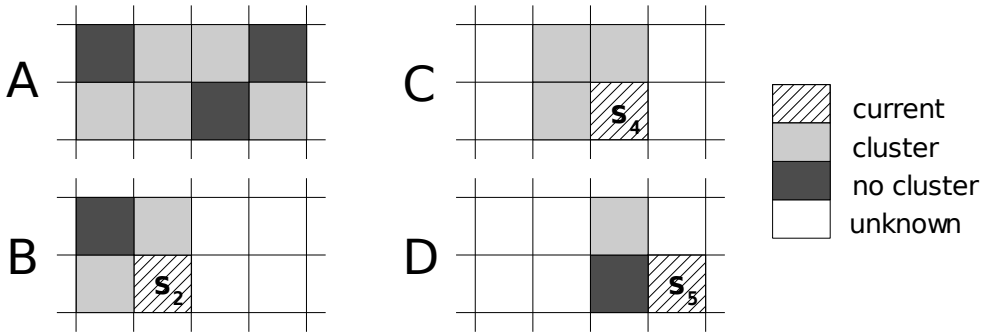

Figure 3-4. An example situation where partial neighbor information can be retained using state s_5.

$FSM = (Q, \Sigma, \delta, q_0, F)$

$Q = \{ s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7 \}$

$\Sigma = \{ C, T \} \cup \{ 0, \ldots, max\_label \}$

$\delta = \{ (s_0, 0/x_n/x_{ne}, s_0), (s_0, 0/!0/!0, s_4), (s_0, C/0/!0, s_2),$
$(s_0, C/!0/0, s_3), (s_0, C/0/0, s_1), (s_1, 0/x_n/x_{ne}, s_6),$
$(s_1, C/x_n/0, s_1), (s_1, C/x_n/!0, s_2), (s_2, 0/x_n/x_{ne}, s_5),$
$(s_2, C/x_n/0, s_3), (s_2, C/x_n/!0, s_4), (s_3, 0/x_n/x_{ne}, s_6),$
$(s_3, C/x_n/0, s_1), (s_3, C/x_n/!0, s_2), (s_4, 0/x_n/x_{ne}, s_5),$
$(s_4, C/x_n/0, s_3), (s_4, C/x_n/!0, s_4), (s_5, 0/x_n/x_{ne}, s_0),$
$(s_5, C/0/!0, s_2), (s_5, C/!0/!0, s_4), (s_5, C/!0/0, s_3),$
$(s_5, C/0/0, s_1), (s_6, 0/x_n/x_{ne}, s_0), (s_6, C/0/0, s_1),$
$(s_6, C/0/!0, s_2), (s_6, C/!0/0, s_3), (s_6, C/!0/!0, s_4),$
$(s_0, T/x_n/x_{ne}, s_7), (s_1, T/x_n/x_{ne}, s_7), (s_2, T/x_n/x_{ne}, s_7),$
$(s_3, T/x_n/x_{ne}, s_7), (s_4, T/x_n/x_{ne}, s_7), (s_5, T/x_n/x_{ne}, s_7),$
$(s_6, T/x_n/x_{ne}, s_7) \}$

$q_0 = \{ s_6 \}$

$F = \{ s_7 \}$

Assumptions:

Each input symbol is composed of the current/north/northeast cell values

C is the cell value denoting an element that should be clustered

T is the cell value indicating the map terminal

! is the unary negator

x_n, x_{ne} are the north, northeast cell values (variable)

x_n, x_{ne} $\in \{ 0, \ldots, max\_label \}$

Figure 3-5. Formal FSM definition of nearest-eight FSM.

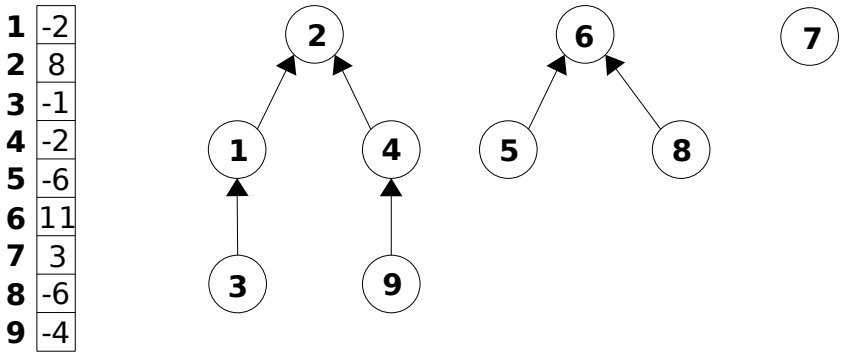

Figure 3-6. A csize array and its representation as a disjoint-set forest.

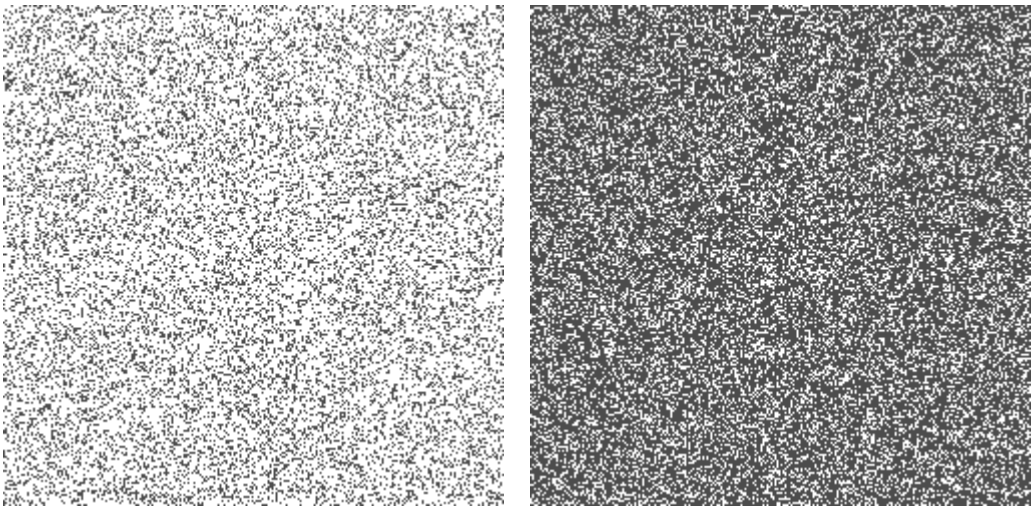

Figure 4-1. Two 5000×5000 binary matrices, with densities *d*=0.25 (left) and *d*=0.7 (right).

Table 4-1: Target Classes, Densities, and Number of Clusters in Fort Benning Landscape Map

Class	Density	Clusters
7	0.00005	133
73	0.00080	252
33	0.00114	105
20	0.00175	961
24	0.00617	7481
11	0.00982	4444
22	0.01128	15623
83	0.01867	7783
80	0.03261	18590
91	0.03980	28149
18	0.04047	518
43	0.04581	59850
31	0.06375	49220
41	0.15167	77996
42	0.20648	58757

Table 4-2: Target Classes, Densities, and Number of Clusters in Tennessee Valley Landscape Map

Class	Density	Clusters
2	0.00160	3248
10	0.00168	13284
8	0.00232	6462
6	0.00233	12553
4	0.00313	10358
15	0.00335	18373
18	0.00546	27199
3	0.00600	4214
16	0.00994	27164
5	0.01137	8253
7	0.01415	13960
9	0.01851	23534
12	0.02054	47455
20	0.02573	121709
13	0.03160	88377
11	0.03631	38711
14	0.05615	175768
21	0.09886	79882
19	0.19180	314362
1	0.22324	549
17	0.23531	210952

Table 4-3: Target Classes, Densities, and Number of Clusters in Yellowstone Landscape Map

Class	Density	Clusters
6	0.00038	5
1	0.00118	5
5	0.06643	104
4	0.24212	80
2	0.32365	83
3	0.36626	124

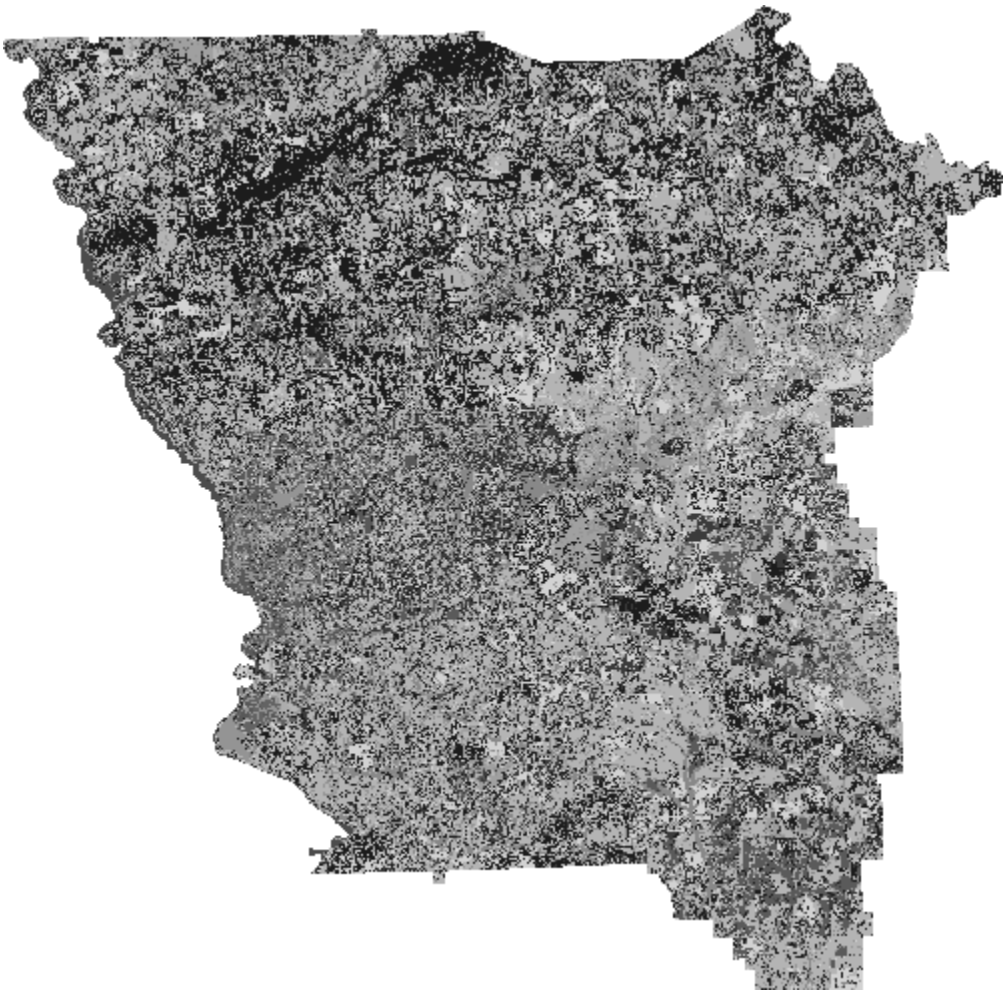

Figure 4-2. A 2771×2814 landscape map of a five county region surrounding Fort Benning, GA. The map contains fifteen target classes.

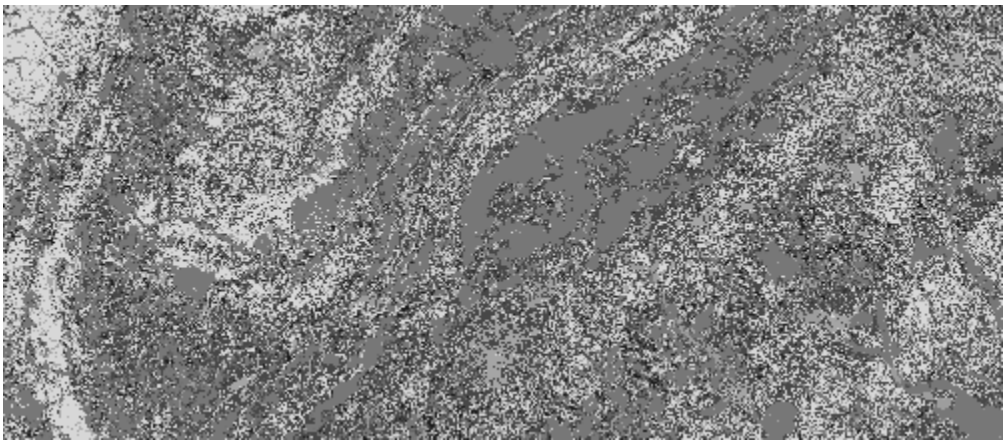

Figure 4-3. A 4300×9891 landscape map centered on the Tennessee Valley and southern Appalachian Mountains. The map contains twenty-one classes.

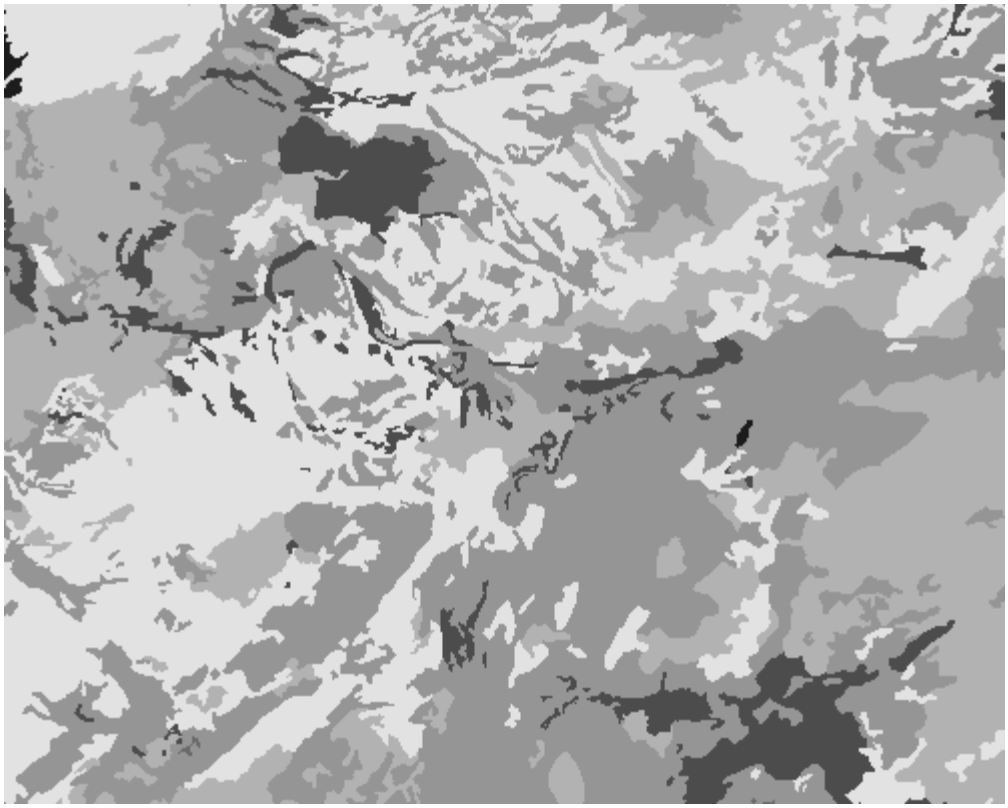

Figure 4-4. A 400×500 landscape map covering a portion of Yellowstone National Park. The map contains six classes.

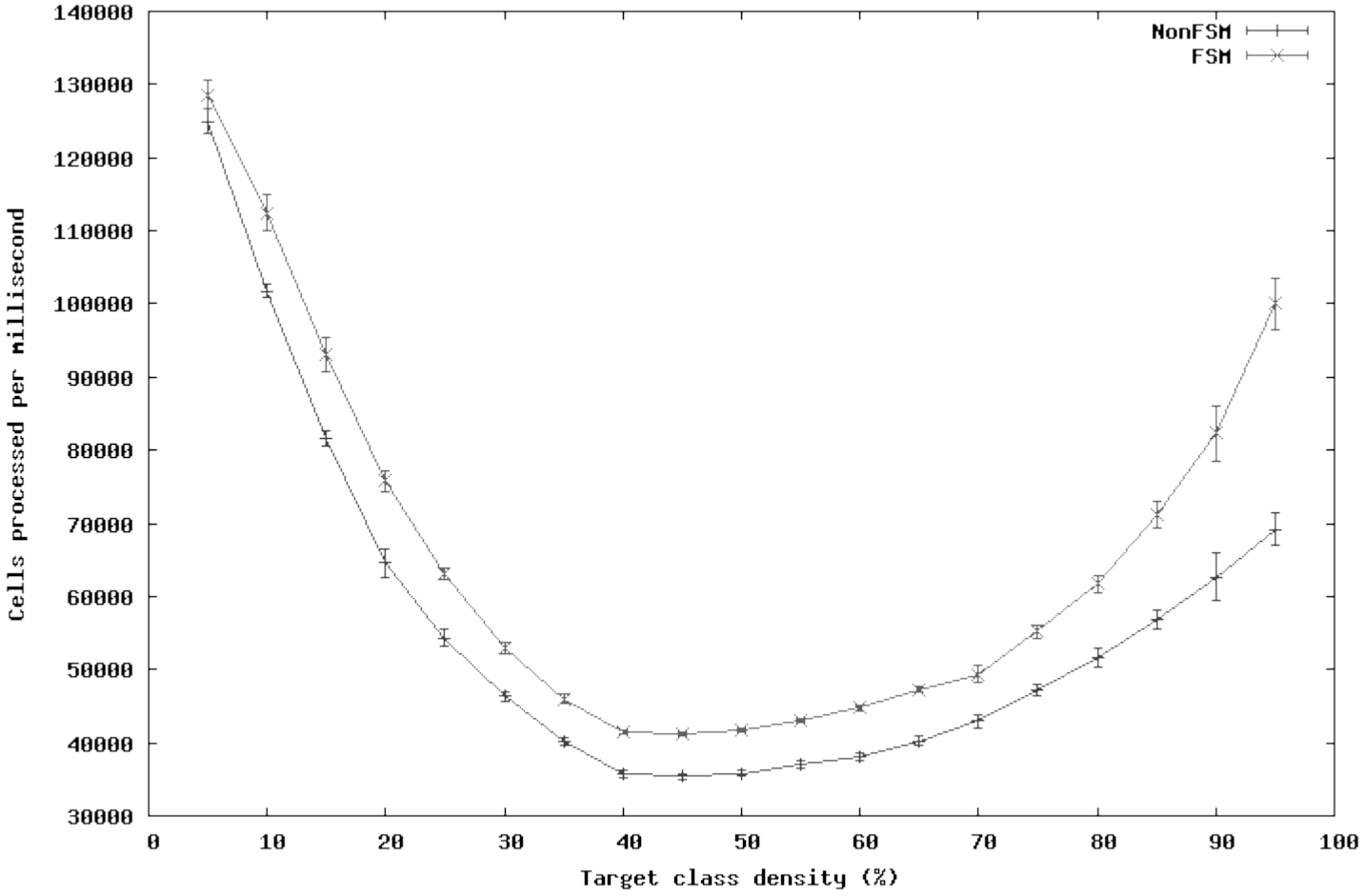

Figure 4-5. Performance of FSM and non-FSM implementations on randomly generated 5000×5000 matrices.

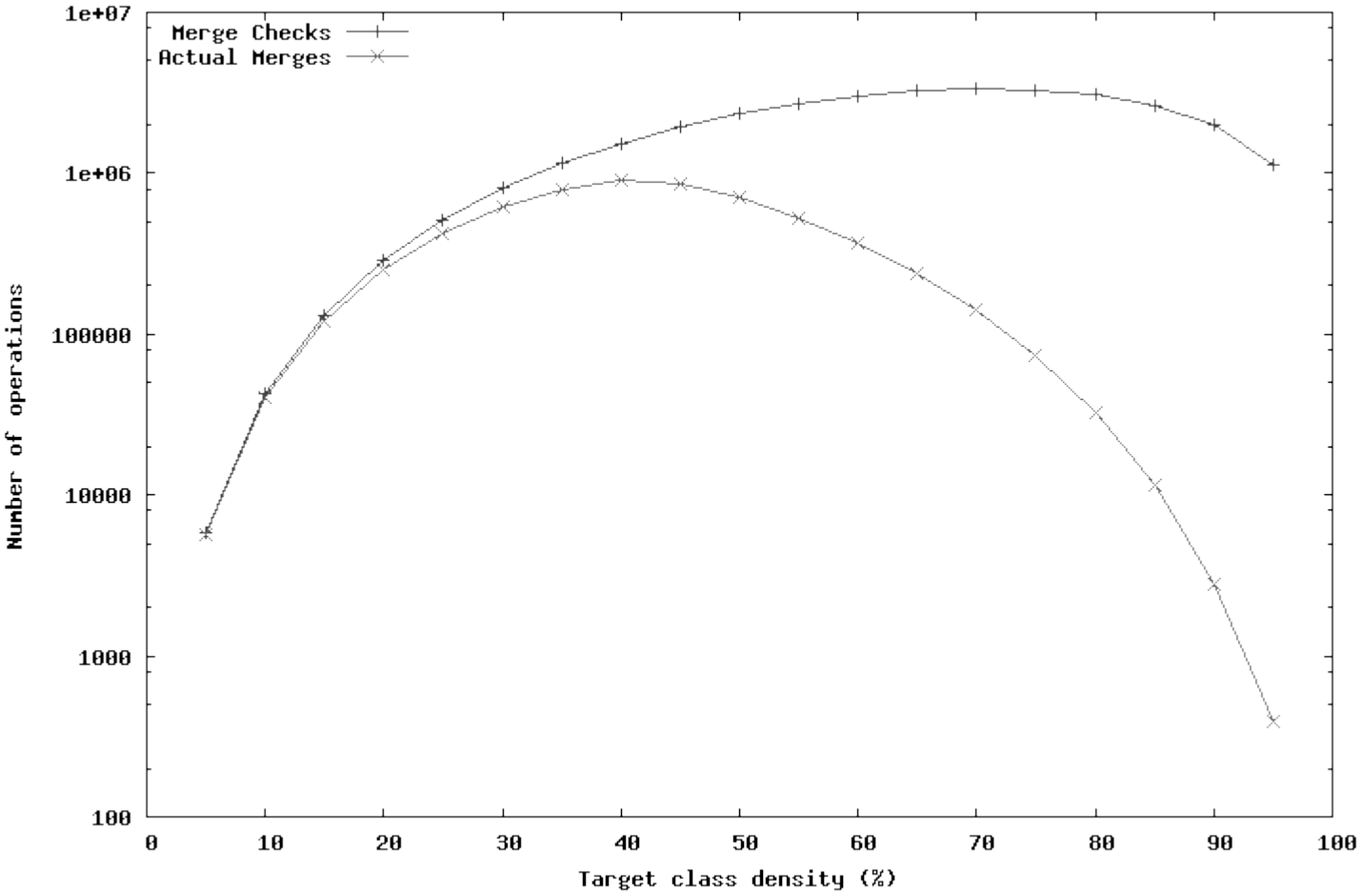

Figure 4-6. Merge checks and actual merge operations on randomly generated 5000×5000 matrices. Note the logarithmic scale of the y-axis.

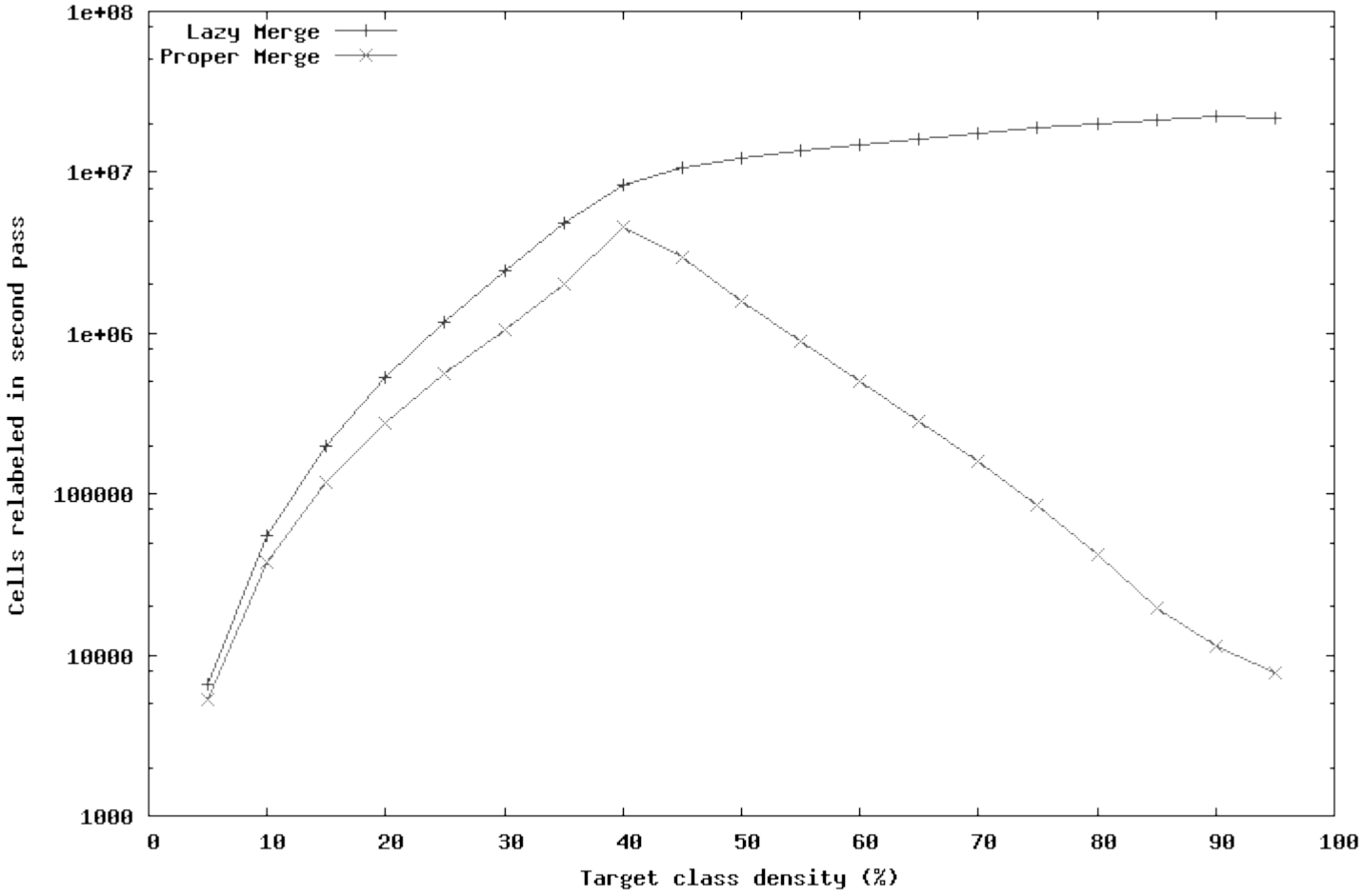

Figure 4-7. Second-pass relabeling operations necessitated by lazy and proper MERGE implementations on randomly generated 5000×5000 matrices. Note the logarithmic scale of the y-axis.

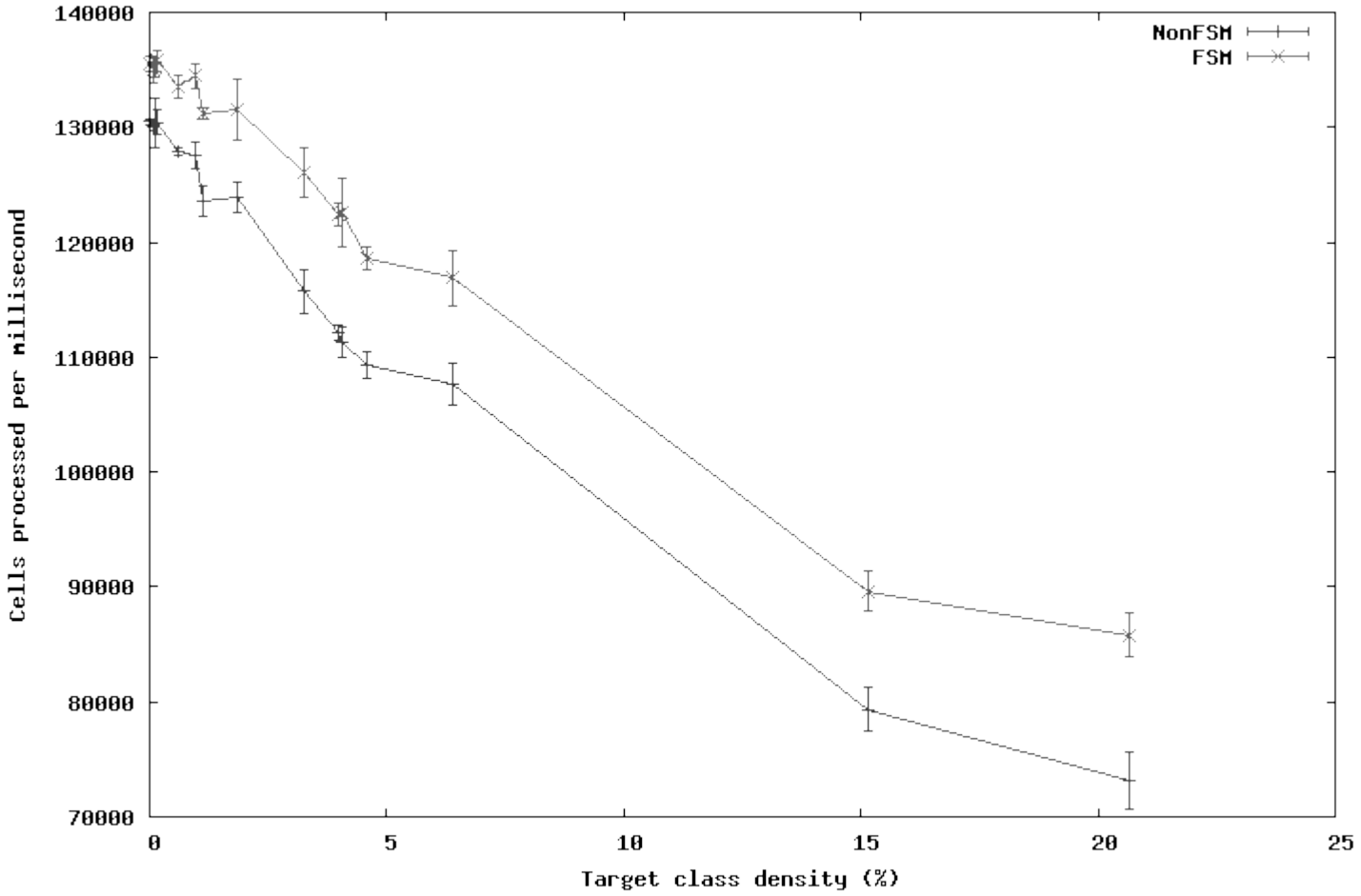

Figure 4-8. Performance of FSM and non-FSM implementations on target classes in 2771×2814 Fort Benning map.

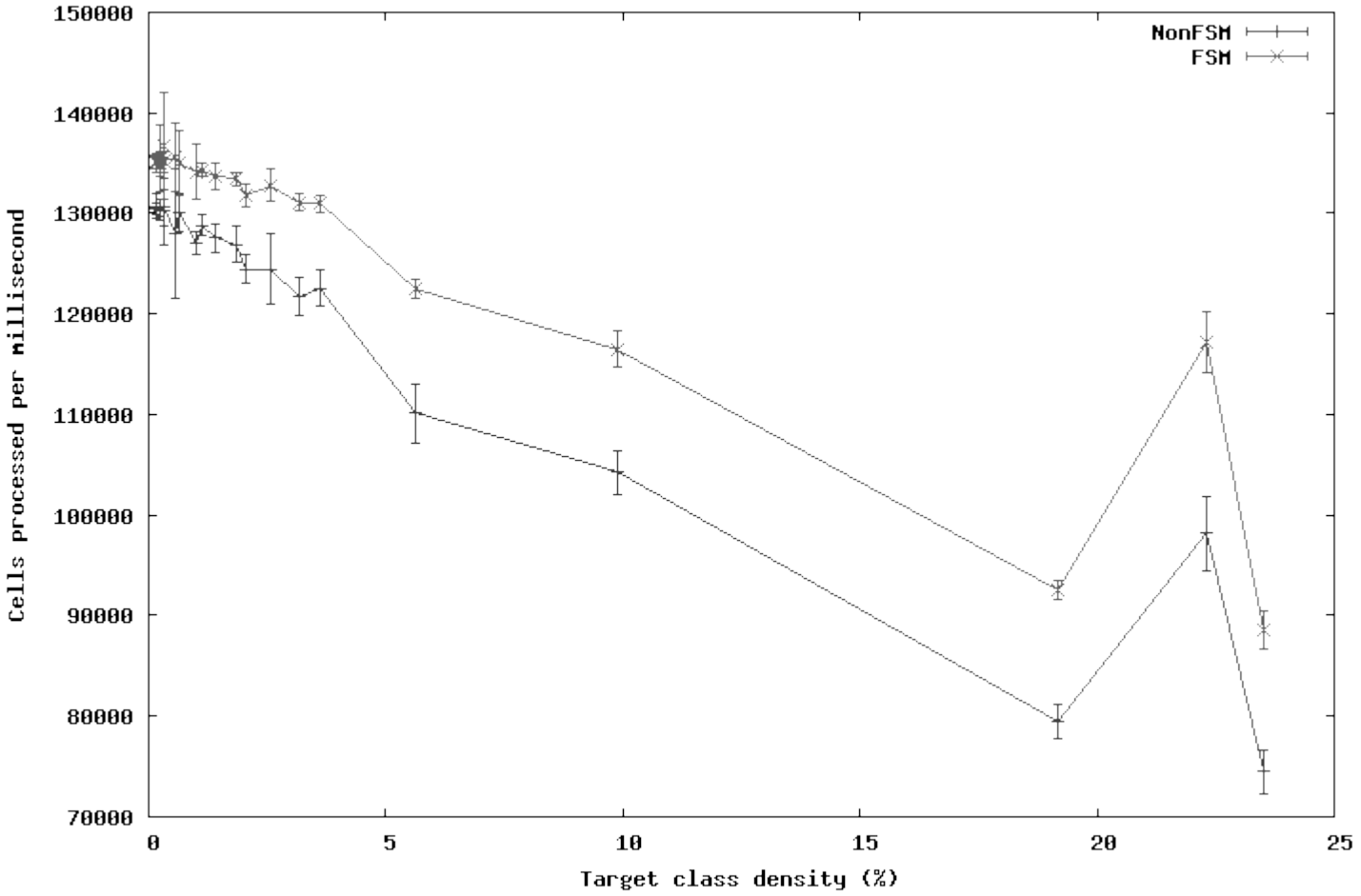

Figure 4-9. Performance of FSM and non-FSM implementations on target classes in 4300×9891 Tennessee Valley map.

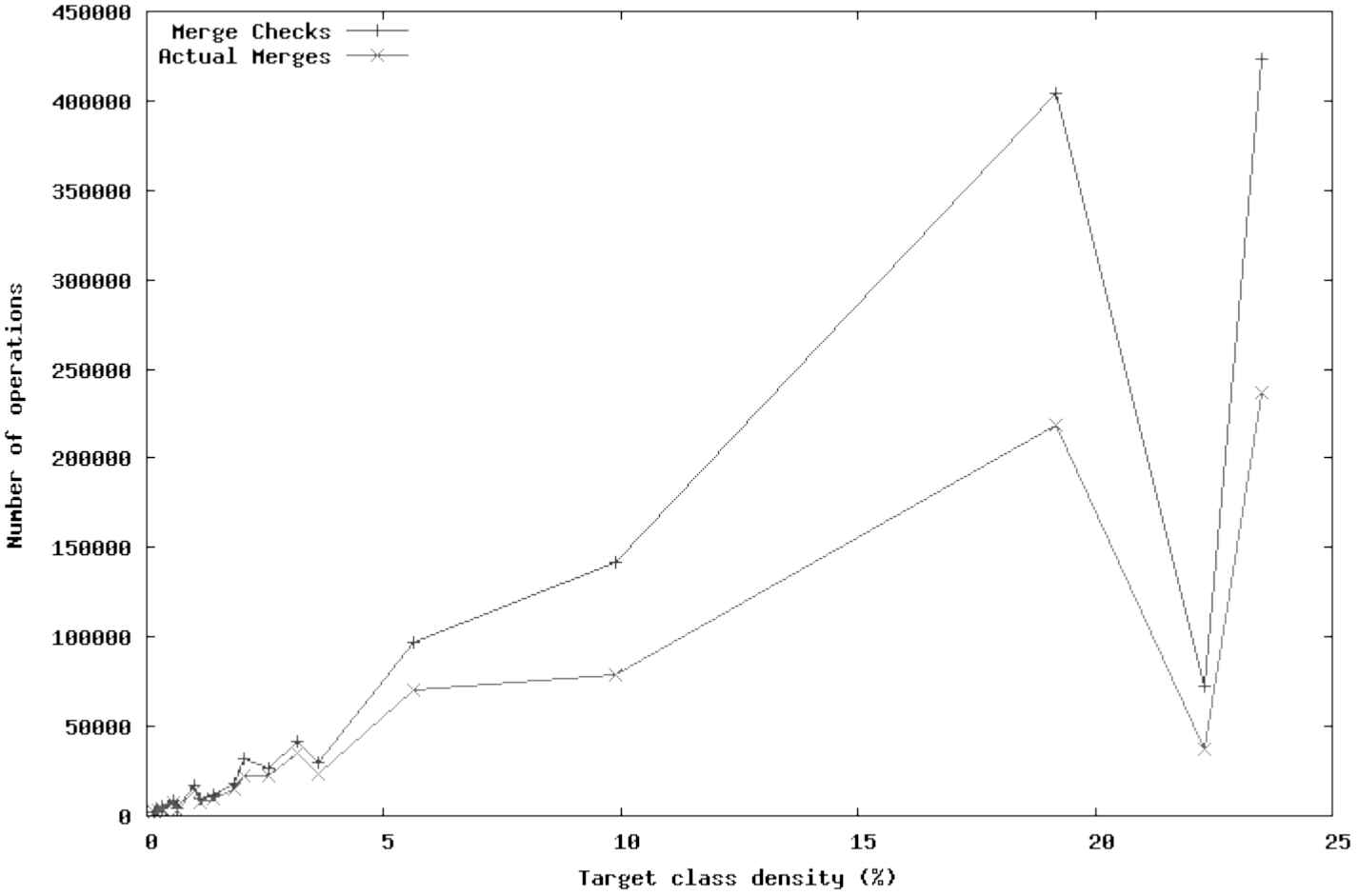

Figure 4-10. Merge checks and actual merge operations on target classes in 4300×9891 Tennessee Valley map.

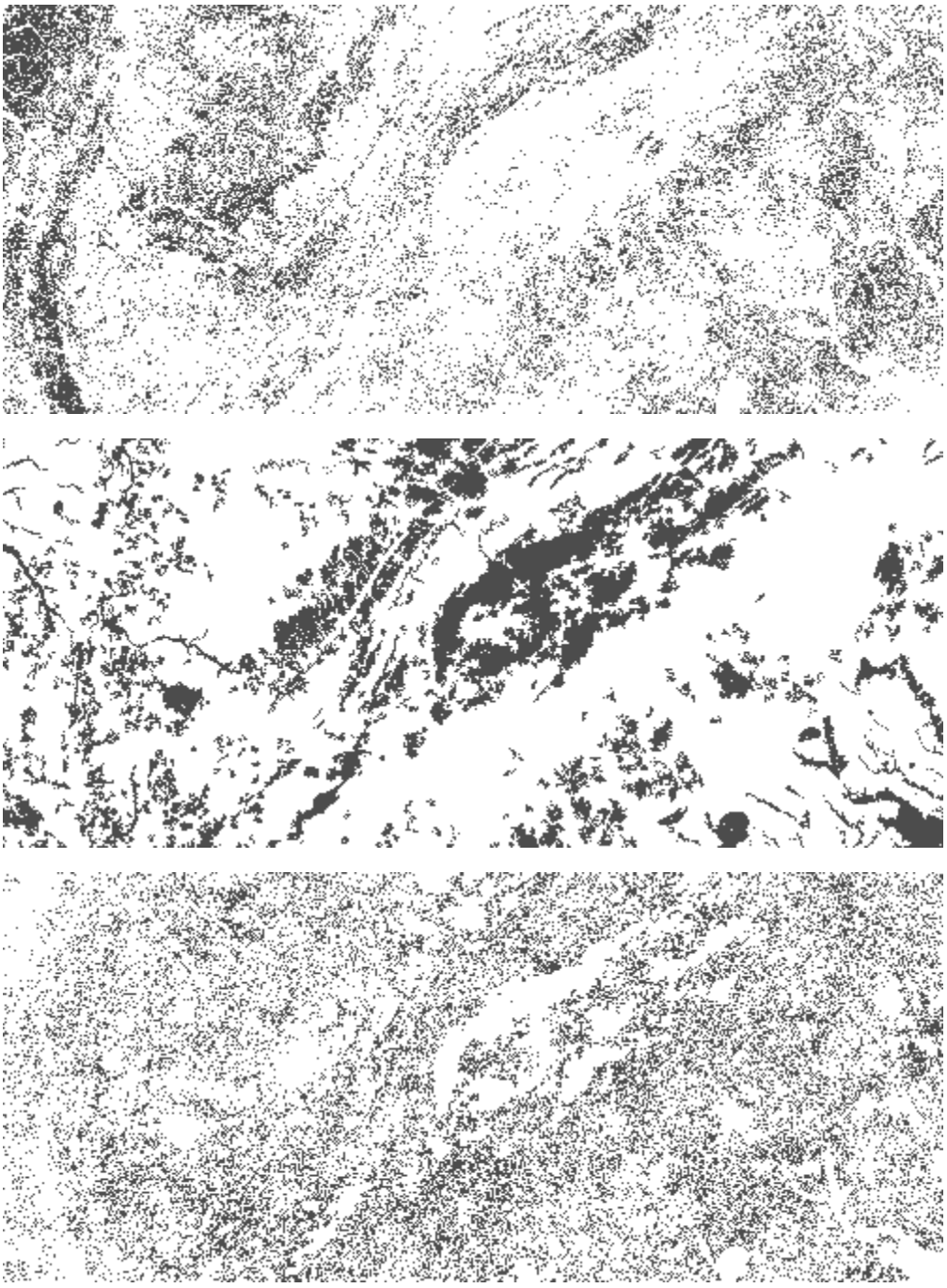

Figure 4-11. The 4300×9891 Tennessee Valley landscape map with class 19 (*d*=0.19180), class 1 (*d*=0.22324), and class 17 (*d*=0.23531) visible from top to bottom.

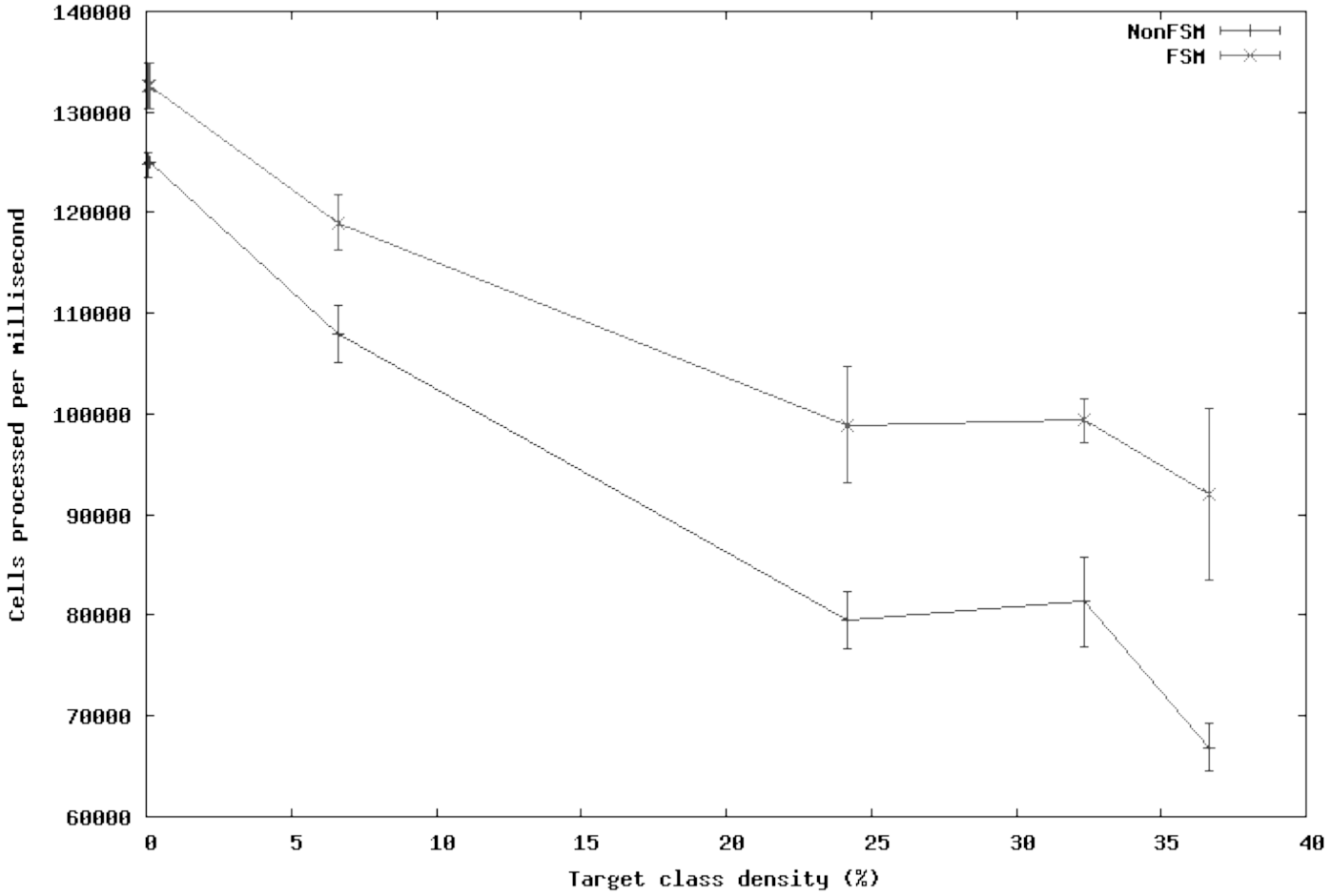

Figure 4-12. Performance of FSM and non-FSM implementations on target classes in 400×500 Yellowstone map.

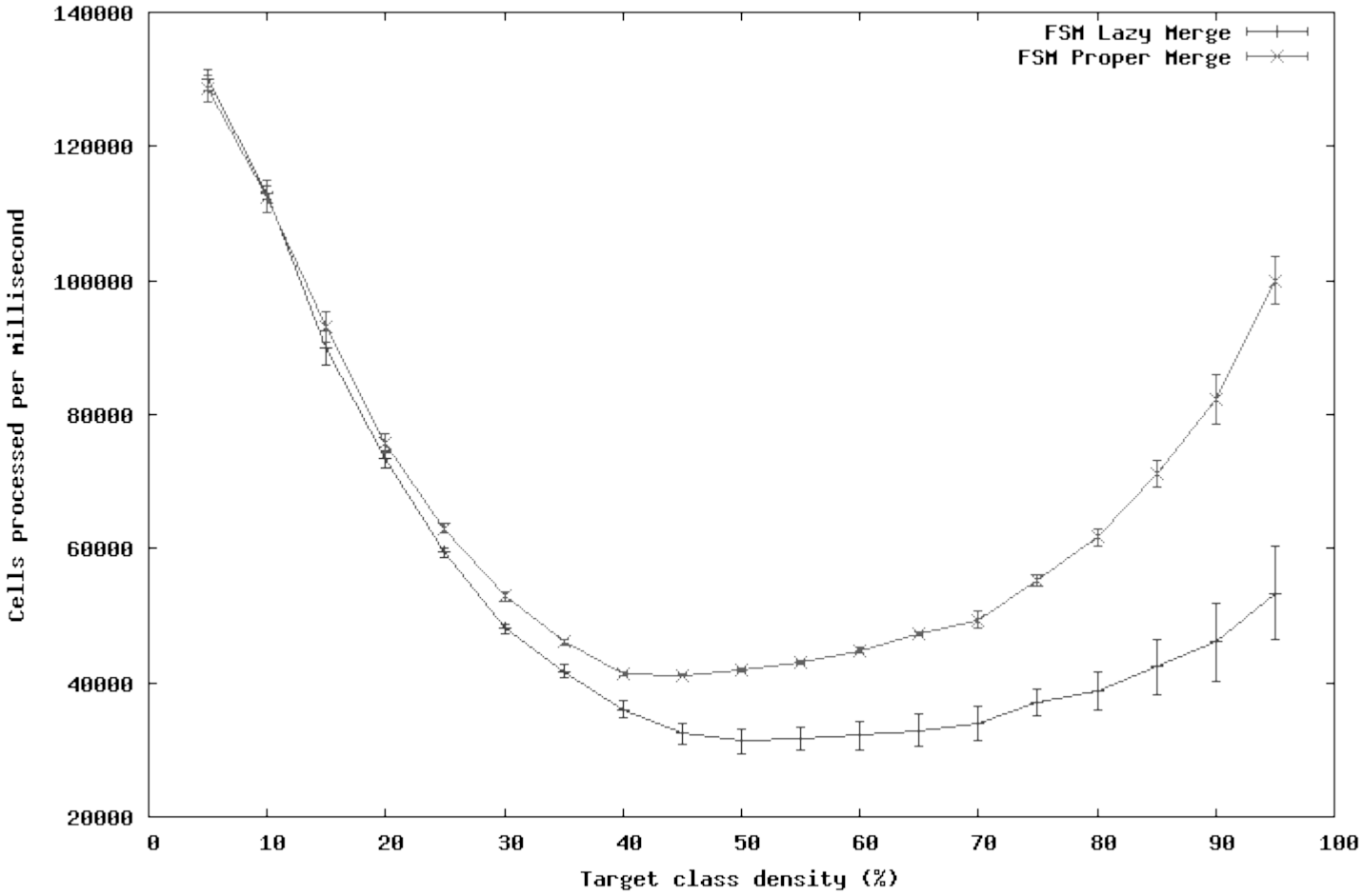

Figure 4-13. Performance of lazy and proper MERGE implementations using a FSM on randomly generated 5000×5000 matrices.

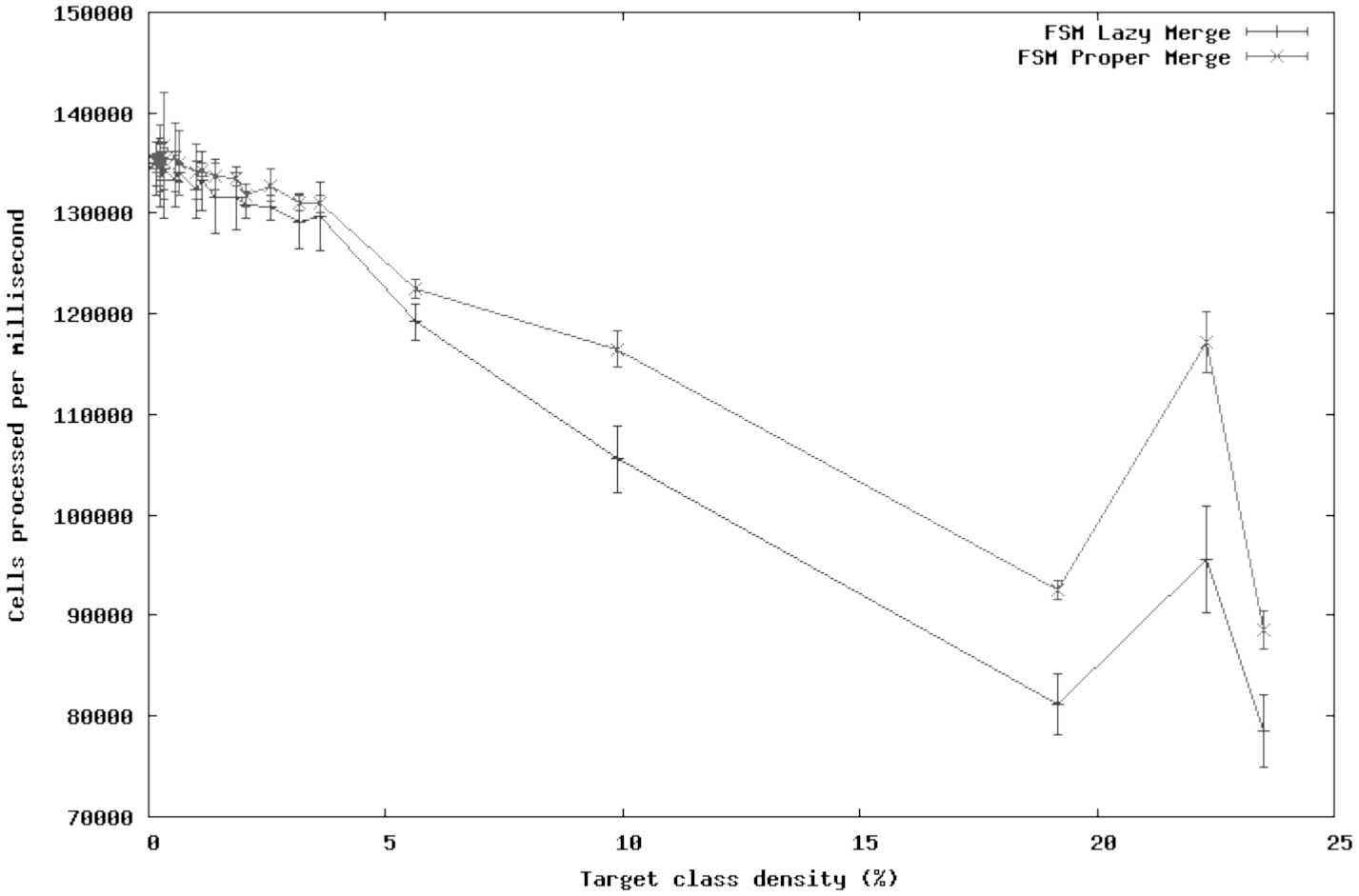

Figure 4-14. Performance of lazy and proper MERGE implementations using a FSM on target classes in 4300×9891 Tennessee Valley map.

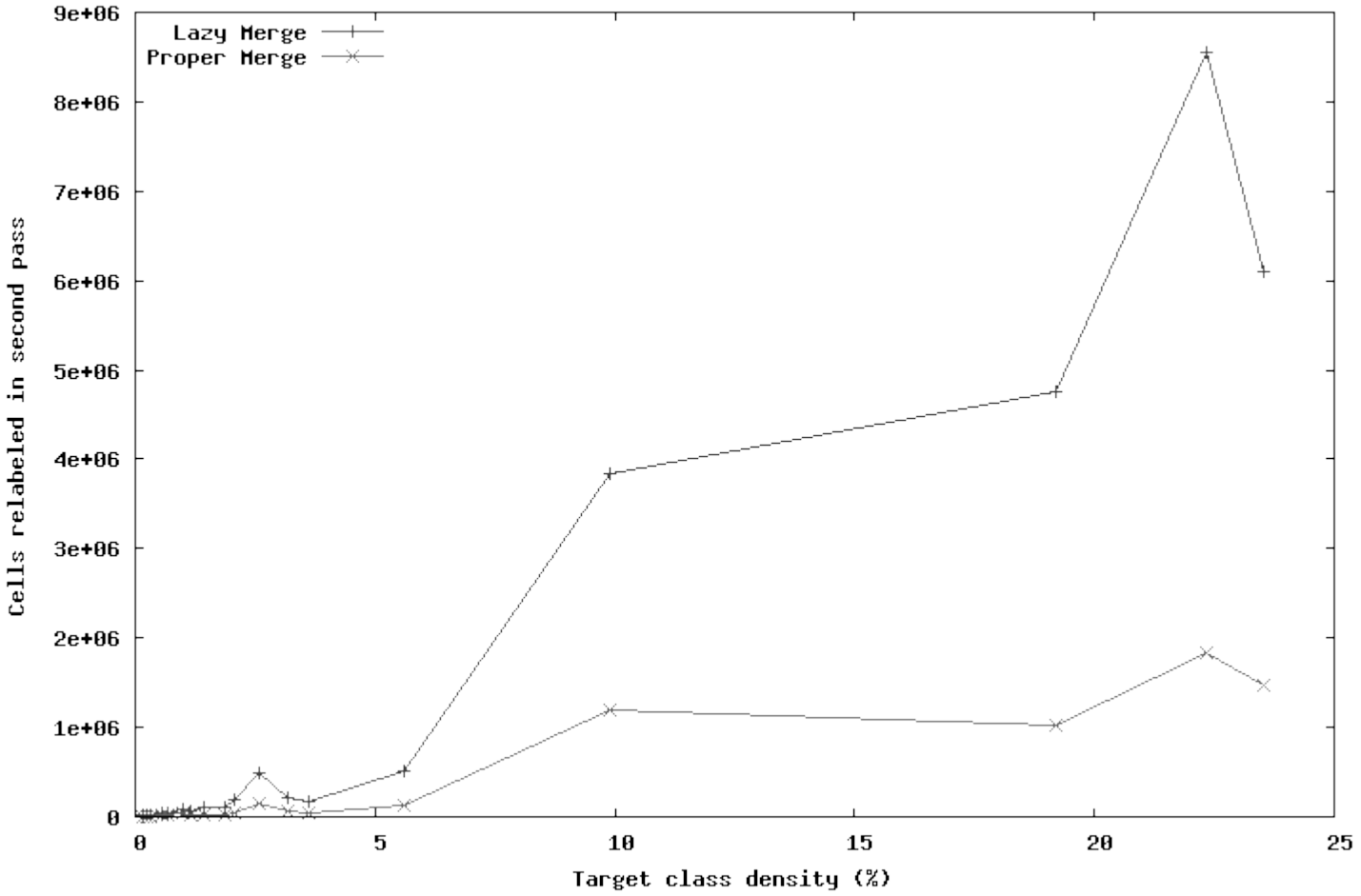

Figure 4-15. Second-pass relabeling operations necessitated by lazy and proper MERGE implementations on target classes in 4300×9891 Tennessee Valley map.

Table 5-1: Target Classes, Densities, and Number of Clusters in Fort Benning Landscape Map Segment

Class	Density	Clusters
20	0.00235	48
83	0.01218	157
73	0.01398	41
91	0.01554	124
11	0.02015	53
80	0.04118	443
31	0.06015	790
41	0.08447	762
43	0.09656	725
24	0.12584	508
22	0.13812	825
42	0.18439	610
18	0.20509	21

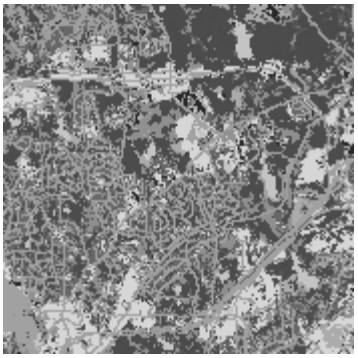

Figure 5-1. A 175×175 segment of Fort Benning landscape map. This map contains thirteen target classes.

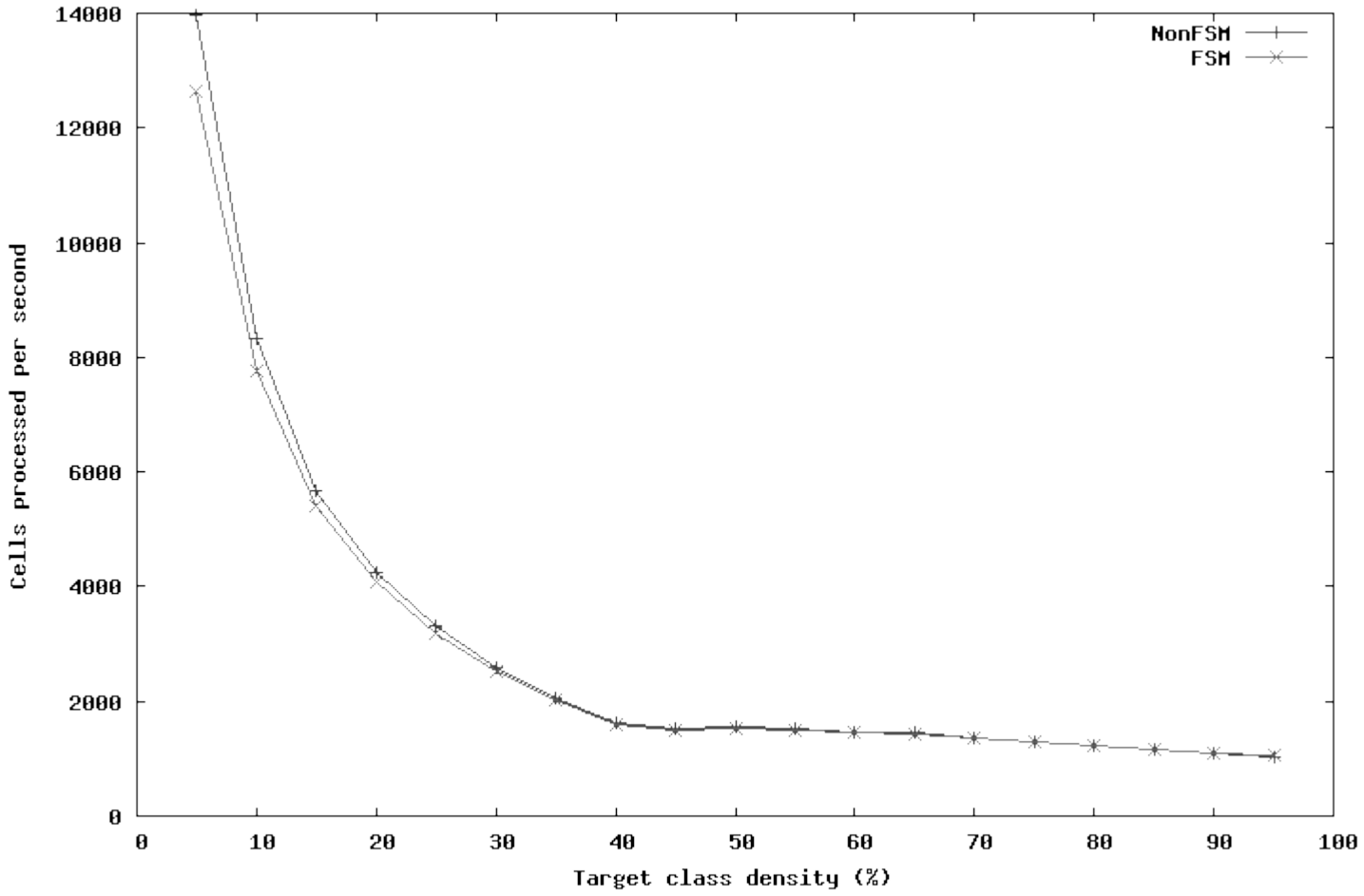

Figure 5-2. Performance of FSM and non-FSM implementations on randomly generated 150×150 matrices.

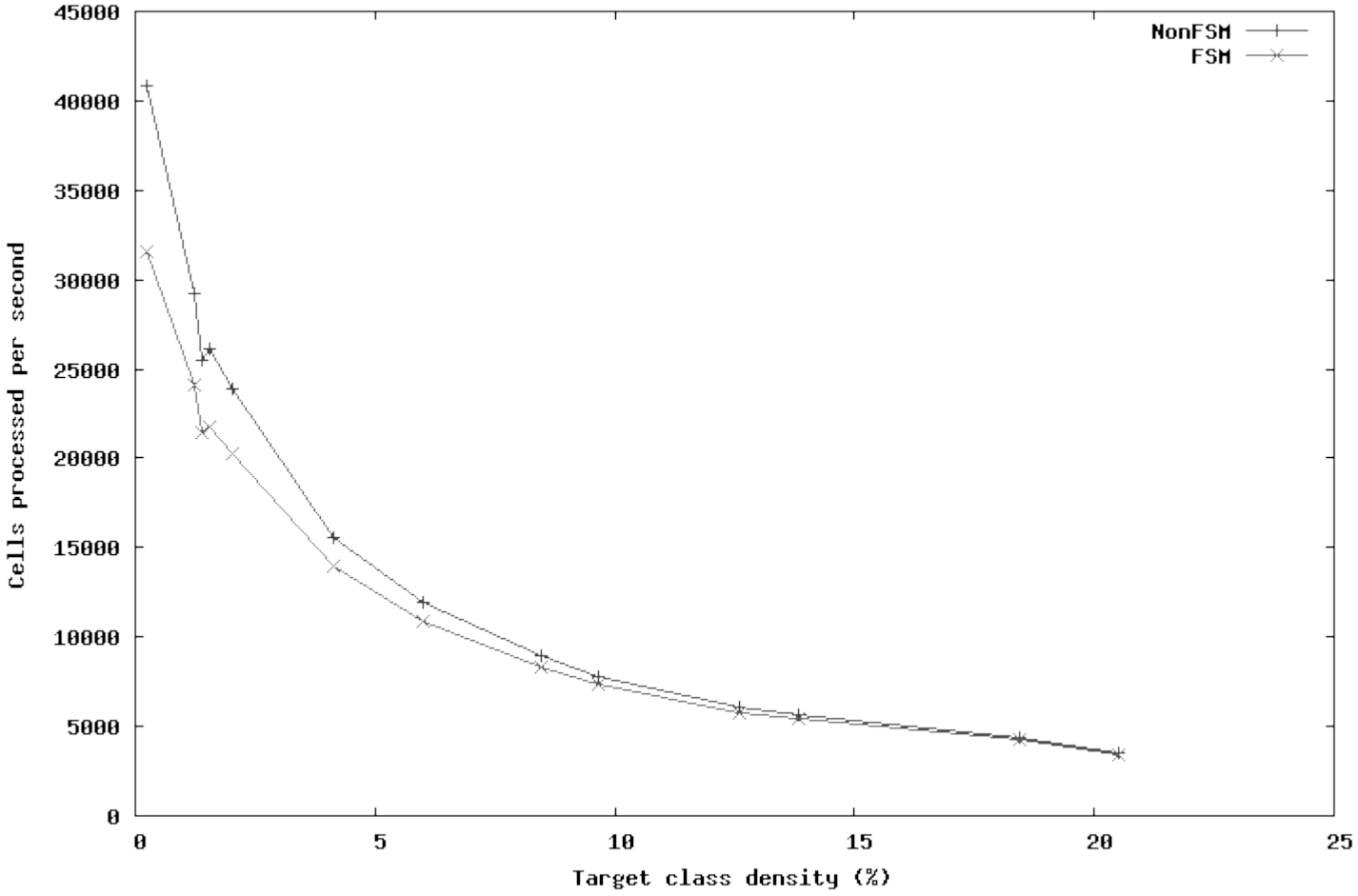

Figure 5-3. Performance of FSM and non-FSM implementations on 175×175 Fort Benning map segment.

```
int16 array[100];
register int16 i, j, k;

k = 1;
for (i = 0; i < 10000; i++) {
    for (j = 0; j < i/100; j++) {
        array[j] = i*2-j;
    }
    if (k == 0) k = 1;
    else k = 0;
}
```

Figure 5-4. C code segment to demonstrate effect of performance degradation caused by code branching.

```
    move.l %d0,-22(%a6)
    moveq.l #1,%d5
    clr.w %d3
    .even
.L5:
    cmp.w #9999,%d3
    jble .L8
    bra .L6
    .even
.L8:
    clr.w %d4
    .even
.L9:
    move.w #5243,%d0
    move.w %d3,%d1
    muls.w %d0,%d1
    move.l %d1,%d0
    clr.w %d0
    swap %d0
    move.w %d0,%d1
    asr.w #3,%d1
    move.w %d3,%d2
    moveq.l #15,%d0
    asr.w %d0,%d2
    move.w %d1,%d0
    sub.w %d2,%d0
    cmp.w %d4,%d0
    jbgt .L12
    bra .L10
    .even
.L12:
    move.w %d4,%a0
    move.l %a0,%d1
    move.l %d1,%d0
    add.l %a0,%d0
    lea (-226,%a6),%a0
    move.w %d3,%d1
    move.w %d1,%d2
    add.w %d3,%d2
    move.w %d2,%d1
    sub.w %d4,%d1
    move.w %d1,(%a0,%d0.l)
.L11:
    addq.w #1,%d4
    bra .L9
    .even
.L10:
    tst.w %d5
    jbne .L13
    moveq.l #1,%d5
    bra .L7
    .even
.L13:
    clr.w %d5
.L14:
.L7:
    addq.w #1,%d3
    bra .L5
    .even
```

Figure 5-5. Loop from Figure 5-3 compiled without optimizations. Branching instructions are in bold typeface.

```
    move.l %d0,%d7
    moveq.l #1,%d6
    clr.w %d3
    lea (10,%sp),%sp
    .even
.L8:
    clr.w %d2
    move.w %d3,%d0
    muls.w #5243,%d0
    clr.w %d0
    swap %d0
    move.w %d0,%d1
    asr.w #3,%d1
    move.w %d3,%d0
    moveq.l #15,%d4
    asr.w %d4,%d0
    sub.w %d0,%d1
    move.w %d3,%d4
    addq.w #1,%d4
    cmp.w %d2,%d1
    jble .L10
    lea (-200,%a6),%a1
    add.w %d3,%d3
    move.w %d1,%d0
    .even
.L12:
    move.w %d2,%a0
    add.l %a0,%a0
    move.w %d3,%d1
    sub.w %d2,%d1
    move.w %d1,(%a0,%a1.l)
    addq.w #1,%d2
    cmp.w %d2,%d0
    jbgt .L12
.L10:
    tst.w %d6
    jbne .L14
    moveq.l #1,%d6
    bra .L7
    .even
.L14:
    clr.w %d6
.L7:
    move.w %d4,%d3
    cmp.w #9999,%d3
    jble .L8
```

Figure 5-6. Loop from Figure 5-3 compiled with optimizations. Branching instructions are in bold typeface.

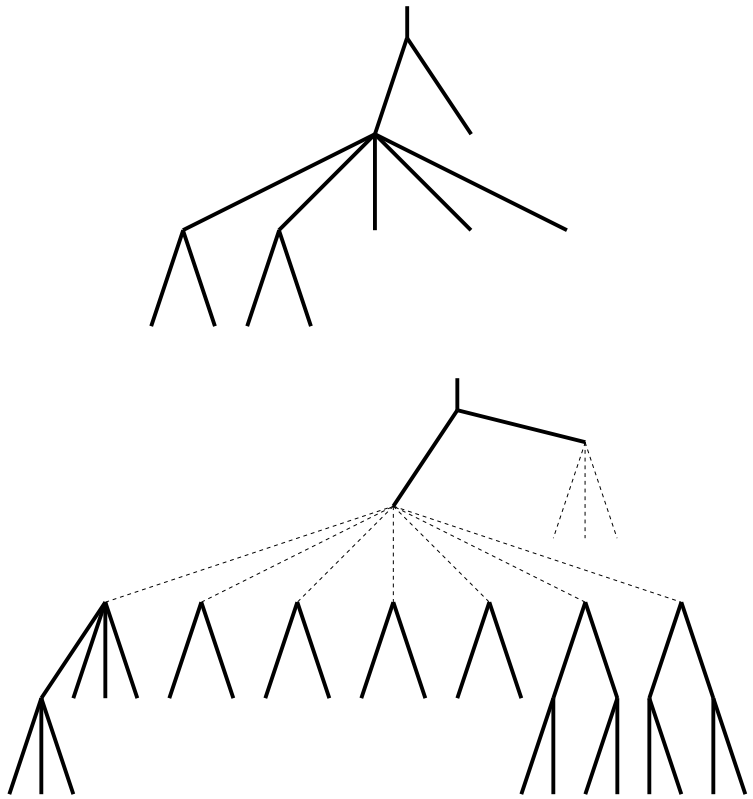

Figure 5-7. Branches in non-FSM (top) and FSM (bottom) code for each cell processed.

Vita

Matthew Lee Aldridge was born in Boone, NC, on September 9, 1980, to David and Debbie Aldridge. He was raised in Salisbury, NC, and later in Norris, TN. He graduated from Anderson County High School in 1998 and received a B.S. in computer science from the University of Tennessee, Knoxville, in 2002. There he has stayed and is currently pursuing his doctorate in computer science.