ı. In: G.W. Gorsline, editor,
do Springs, CO, 1982.

ƽpace Related Data: The Field
ımetry, Swiss Federal Institute

ıdex Structure for Geometric
nce on Data Engineering, Los

ıpatial Searching. In: Proceed-
4.

d Case Studies of Applications
;y, Zurich, Switzerland, 1985.

ıdexed Relations. In: Proceed-
Angeles, CA, February 1989.

ımmetric Multi-Key File Struc-
984.

ct-Oriented Database System.
Data, 15(2), 1986.

A-SIGMOD, International Con-

ı Structures. ACM Computing

lulti-Dimensional Objects. In:
Brighton, England, September

A FULL RESOLUTION ELEVATION REPRESENTATION REQUIRING THREE BITS PER PIXEL

Clifford A. Shaffer
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

ABSTRACT

A quadtree-like representation for storing gridded elevation data is described. The data structure is a pyramid with each node containing two bits of data. The root of the pyramid has associated with it the minimum elevation for the corresponding grid and the maximum variance (the difference between the minimum and maximum values). The elevation value at any pixel is calculated by traversing a path from the root to the pixel, refining the local elevation value during the decent by interpreting the two bit codes stored with each node along the path. Since the total number of nodes in the pyramid is 4/3 the number of pixels required for the bottom level of the pyramid, the amortized storage cost is less than 3 bits per pixel, regardless of vertical resolution. This scheme is most appropriate for efficient secondary storage archival, such as on a CD-ROM. It allows efficient retrieval of complete elevation data from any sub-region, at multiple scales, within the entire elevation database. This is a lossless encoding when the difference between sibling pixels is not "too great".

1. INTRODUCTION

The need for high quality topographic data combined with the difficulties encountered when processing such large volumes of data have resulted in many competing representations. One method is to store a collection of benchmark points with associated elevation values. The value for locations not explicitly stored is derived by interpolating the values of nearby *a priori* data points. This approach is time consuming since the nearby data points must be located, and the interpolation performed. The reliability of

This work was partially supported by General Dynamics and by the Virginia Center for Innovative Technology.

such an interpolation is also questionable (consider for example, the results of fractal geometry [Mand82]). Finally, the storage requirements include space for the coordinates of the data points since they may not be at specified intervals. One popular variation on the benchmark point approach is the Triangulated Irregular Network [Peuk78].

Another approach to storing elevation data is as a grid with an elevation value at every cell. Stereographic gestalt mapping techniques make this an attractive approach since grid data is easily captured from areal photographs [Kell77]. This method typically requires 8 or 16 bits of data at every grid cell (depending on the range and resolution of the elevation values). For large areas of the earth at any reasonable resolution, this approach requires a great deal of storage.

The quadtree [Same84] and its many variants has successfully supported a number of cartographic applications. The quadtree data structure provides a spatial index to locate spatial data objects efficiently. For example, see [Shaf86] for a description of a prototype GIS that supports thematic regions, points, and linear feature data. Many related approaches have also been applied to computer cartography and computer graphics applications requiring spatial indexing. For example, EXCELL [Tamm81], image pyramids [Anto86, Shaf87], and others. For the remainder of this section, I will loosely use the term "quadtree" to refer to all such spatial indexing methods.

The quadtree approach attempts to save both space and time over simple grid approaches by either aggregating homogeneous regions (as does the simple region quadtree), or by spatially organizing a collection of data objects (as does the PM quadtree for lines [Same85b]). Such approaches have not been successful in the past for representing elevation data since such data does not fall into either category. Elevation data may be viewed as a surface whose z value varies over the x and y dimensions. Unless the vertical resolution is extremely coarse in relation to the horizontal resolution, neighboring pixels in the corresponding grid are not homogeneous. Thus the region quadtree approach will not be space efficient since there are no homogeneous blocks to aggregate. [Cebr85] suggests that while the quadtree will not compress homogeneous blocks, the elevation grid should still be arranged in Morton order [Mort66] since it would than be more compatible with quadtree-based GIS.

Another approach to applying a quadtree representation to topographic data is to derive a relatively small collection of spatial data objects from the data to be stored by means of a spatial index. One approach is to generate a collection of surface patches to approximate the topographic surface [Mart82]. [Chen86, Leif87] apply the quadtree representation to this technique.

The benchmark point method of storing elevation data may be more amenable to spatial indexing methods since the data points could be indexed with a PR quadtree [Same84] or a grid file [Niev84]. However, while the spatial index makes it easier to locate data points near to a query point, the interpolation problems remain. In addition, if the number of data points is high in relation to the number of grid cells at the desired resolution, than the additional storage required to support the spatial indexing method may well be greater than that required by simply interpolating the data points over the grid and storing the grid. Note that the grid requires no storage overhead beyond the data value at each grid cell. If the number of data points is greater than about 10% of the grid cells, the grid will probably be more efficient in space and time than the aforementioned quadtree methods.

We present an alternative approach to representing elevation data, termed the ***elevation pyramid***. The elevation pyramid uses an image pyramid structure [Tani75] to allow aggregation of the elevation data at higher levels in the pyramid. This aggregation allows full resolution elevation data to be stored at an amortized cost of less than 3 bits per grid cell. It is a generalization of the DEPTH coding scheme of Dutton [Dutt83], though our approach to coding methods is quite different.

2. DATA AMORTIZATION

Trees are often used to aggregate computation costs when processing the data stored in the tree. For example, if every leaf node in a tree must be visited, it is inefficient to travel from the root to each leaf node of the tree in turn. Instead, a tree traversal is performed where each node of the tree is visited once. Since the total number of nodes in the tree is $O(N)$ where N is the number of leaf nodes, the total cost of a traversal is $O(N)$, not $O(N \log N)$ or worse as would be required if each leaf were processed separately. The cost of traversing the path from the root to any leaf node is amortized over the entire traversal. An example of such amortization of the computation in a quadtree is the neighbor passing traversal algorithm of [Same85a]. Here, the neighbors of the child of a quadtree node Q are recognized to be children of Q and its neighbors. Thus, the neighbor-finding computation can be amortized over the traversal of the tree.

While amortization of computation by tree methods is commonly practiced, amortization of the data to save storage is less common. One well known approach is the ***trie*** [Fred60, Aho83], which can be used to store dictionaries in a space efficient manner. For example, assuming a 26-letter alphabet, the trie can be represented as a 26-way tree with each branch labeled by a corresponding letter. All words starting with 'a' would be placed in the 'a' branch of the tree; those starting with 'b' in the 'b' branch, etc. At the second level of the tree, words are further separated by their second letter, and so on. To locate a word in the trie, the path from the root to the desired word is followed, collecting the letters during the process. The total amount of storage required is related to the number of differences between words, not to the total number of characters in the words.

The trie data structure not only compresses the number of characters that must be stored, but also organizes one-dimensional data for efficient retrieval. When storing elevation data, we must be concerned with three dimensions: the location in 2-D space and its elevation value. Both the grid, and hierarchical spatial data structures such as the quadtree, index space with the location of cells or nodes implicitly contained in their structure. Objects whose data values are correlated to their positions are amenable to data aggregation.

An idealized (though impractical) example of such an aggregation would be the location of quadtree blocks. Morton codes [Mort66, Garg82] have been used extensively as an addressing technique to support the linear quadtree (i.e., a quadtree representation that stores only leaf nodes along with a description of their size and location). One way to derive the Morton code is by interleaving the bits of the x and y coordinates for a

pixel. An alternative characterization is as a description of the path from the root to the pixel. The NW branch has a 2 bit code, say, 00. The NE branch has code 01, and so on. The address for a pixel is the concatenation of the codes for each branch in the path. If it were desirable to explicitly store the Morton code address for each node, one could store the complete address of size $2 * n$ (for a $2^n \times 2^n$ image) with each node. A more space efficient approach is to store with each node the 2 bit code representing the branch from the node's father. The complete address is generated by concatenating the codes during the descent. The addresses of neighboring nodes are perfectly correlated, and perfectly inheritable from the parent. A simple inductive proof on k-ary trees shows that the total number of nodes in a quadtree is $\frac{4}{3}L$ where L is the number of leaf nodes in the tree. Thus, this technique would require only $\frac{8}{3}$ bits per node.

Of course, there is no reason to store Morton code addresses with the nodes in a tree structure since they can be computed from the tree structure during traversal. In fact, the most significant application of Morton codes in quadtree representations is to eliminate the tree structure itself. However, other types of data are such that nearby nodes have correlated data values. The concept of aggregating the related portions of the data in the common ancestor nodes can be used in these cases. One significant example of such data is elevation.

The method developed in this paper is related to the technique of progressive transmission for images [Sloa79, Know80, Hill83, Hard84]. Progressive transmission contains elements of both computation and data amortization. These methods store in a variable some form of "average" value for the image, and utilize a tree structure to store "differences" indicating how the refinement takes place as one proceeds through the tree. These methods allow for transmission of images with no loss of information, no increase in storage requirements, and yet at the same time allowing ever-improving versions of the image to be transmitted. In addition, homogeneous regions of such images need not be redundantly transmitted at finer resolution. Progressive transmission allows the receiver to cut the transmission prematurely if the full resolution image is not desired. Note that these methods do not utilize data aggregation for storage compression. The application of lossless image transmission does not encourage the storage saving method described in this paper since adjacent image pixel values are not sufficiently well correlated.

3. THE ELEVATION PYRAMID

This section describes our representation for elevation, termed the *elevation pyramid*. This representation reduces the storage requirements of the grid method for storing elevation data to less than 3 bits per grid cell. It does this by taking advantage of the correlation of the data values for pixels within any region of the grid.

The pyramid structure [Tani75] is derived from a $2^n \times 2^n$ grid. The pyramid can be viewed as a stack of arrays which stores at the bottom level the entire image of size $2^n \times 2^n$. At the next level, each disjoint 2×2 pixel block is represented by a single cell, with the entire image at this level represented by $2^{n-1} \times 2^{n-1}$ cells. This process continues until the nth level, which contains a single cell. Thus, the pyramid

is equivalent to a complete quadtree, i.e., one in which all internal nodes have four children and all leaf nodes are at the lowest level of resolution. However, the pyramid representation requires no pointers since the memory location of the children, sibling, and parent nodes can be derived from the memory location of the current node (much like traditional storage techniques for the ***heap*** data structure [Aho83]). We define the bottom level to be level 0, and the root to be level n.

The basic elevation pyramid represents an elevation map as follows. Associated with the pyramid is the minimum value of all pixels in the image, and the difference (or variance) between the minimum and maximum values. We define two global variables, ALLMIN and ALLVAR. ALLMIN stores the minimum value within the image. ALLVAR is the greatest power of 2 less than or equal to the difference between the minimum value of in the image and the maximum value. As the pyramid is traversed, local copies of these variables will be refined until the pixel level is reached, at which point the actual value for that pixel is determined.

Each internal node of the pyramid is 8 bits long, divided into 4 fields of 2 bits each. Strictly speaking, these four fields represent the codes for the children of that internal node; however, implementation is more efficient if these values are grouped together to form a single byte value. Leaf nodes (i.e., level 0 in the pyramid) thus do not require any storage. Since the complete pyramid contains $\frac{1}{3}N$ internal nodes where N is the number of pixels, with each internal node requiring 8 bits, the total amount of storage is $\frac{8}{3}$ bits/pixel.

Each 2-bit field represents the required modification to the minimum and variance values for the current node to generate the corresponding values for the child associated with that field. In our initial coding scheme, the first bit indicates modification to the minimum elevation for that subtree, while the second bit indicates modification to the maximum variance for that subtree. Specifically, the two bits can be defined by the following operations on variables M and V, which will then be passed to the child node for future operations:

Code 00 - $M_{new} := M_{old}$; $V_{new} := V_{old}/2$;
Code 01 - $M_{new} := M_{old}$; $V_{new} := V_{old}$;
Code 10 - $M_{new} := M_{old} + V_{old}/2$; $V_{new} := V_{old}/2$;
Code 11 - $M_{new} := M_{old} + V_{old}$; $V_{new} := V_{old}$;

Determining the value of a pixel begins with the root of the tree, where M is initialized to ALLMIN, and V is initialized to ALLVAR. As the pyramid is traversed from the root to the desired pixel, the appropriate two-bit field is processed and the variables updated. When a pixel is reached, the stored value for that pixel is in M. Algorithm 1 shows Pascal-like pseudocode for the decoding operation using this coding method. Figure 1 shows a sample elevation grid and the corresponding elevation pyramid. Pointers are drawn between nodes in these figures purely to aid in viewing the picture. An actual implementation would store the nodes as a list in memory.

Construction of the coded elevation pyramid is easily performed. We assume that the original elevation array is available during the coding process. This elevation data is stored into the base of a temporary pyramid which is capable of storing the minimum value and variance for each node in the pyramid. Two passes are then made over the temporary pyramid. The first pass sets at each internal node of the temporary pyramid

the minimum value and variance for its sub-pyramid. This is performed by a simple post-order traversal of the pyramid. The second phase generates (and outputs) the encoded form of the final pyramid. ALLMIN and ALLVAR are set based on the minimum value and variance of the root. For each internal node of the pyramid, the minimum and variance values of each child are examined. The function MAKECHILDCODE compares these values to the current value of M and V to generate the proper code. As a side effect, MAKECHILDCODE sets the minimum and variance values of the child node in the intermediate pyramid to the values computed from the codes traversed thus far, making these values available when MAKECHILDCODE is executed on the child. One advantage of this approach is that the algorithm we present can easily be converted to operate as a breadth-first traversal. The PASCAL pseudo-code for the second pass in this process is shown in Algorithm 2.

4. VARIANTS ON THE ELEVATION PYRAMID

Ideally, the elevation pyramid would provide a completely faithful representation of the original elevation data. The term "faithful" is used instead of "accurate" considering that the source of elevation data is often suspect. A potentially significant problem with the coding scheme described in the previous section is that rapidly changing elevation values cannot be faithfully represented, as shown in Figure 2. Note in this figure that 5 pixels (whose values are circled) were incorrectly labeled. In each case the computed value is one less than the true value. Figure 3 reverses the pixel elevation values of Figure 2 in order to demonstrate that the minimum elevation representation does not give anomalous affects when the minimum value varies rapidly. Figure 3 yields similar results to Figure 2.

We see that the elevation pyramid tends to smooth out rapid changes in elevation. The significance of this problem would depend on the relationship between horizontal and vertical resolution. If the original elevation data is not extremely accurate, minor degradation in the representation may not be of concern. However, greater faithfulness can be achieved through various modifications to the basic coding algorithm.

Note that the code generation function of Algorithm 2 has been very carefully constructed. Changes in this function can lead to significant changes in the quality of the codes produced. For example, the variance value can be the greatest power of 2 *less than or equal to* the true variance, as opposed to the greatest power of 2 *less than* the true variance. Otherwise, a variance of 1 would always be reduced to 0, yielding incorrect values. Another feature of the function MAKECHILDCODE is that it first checks and, if possible, modifies the minimum value for the child without regard to the child's variance. In the case of code 11, this could lead to rapid increases of the minimum value while maintaining the variance even when the true variance for that area is rather low. However, it is reasonable to expect that in general, if the minimum value is changing rapidly, the variance will remain high. Likewise, code 10 allows a moderate increase in the minimum value, but forces a decrease in the variance regardless of the true state. One might be concerned that an area with a slightly higher minimum value

but a high variance might be incorrectly coded since the variance is artificially reduced at the expense of an increase in the minimum value. However, this is not normally a problem since small increases in the minimum value can only occur when the variance is already low enough to allow them (since the least increase is $V/2$). Modifications to the code generating function, or the definition of the codes themselves, would form one class of variants to our algorithm.

Another class of variants on our basic coding scheme results from storing and modifying values other than the local minimum and variance. For example, the average elevation value for the subtree and the variance could be stored, with a suitable modification being to add or subtract the current variance at each branch in the tree. We do not speculate further on such variants.

A third class of variants results from modifying the definition of the codes based on the current level in the pyramid. Defining the lowest level to be 0, the next as 1, etc., we can modify the codes based on level to take advantage of peculiarities at the bottom two levels. In particular, we note that modification to the variance at the bottom level is not significant, except for its affect when added to the minimum. A better coding for the bottom level would be to increase the range of effect for the variance. Thus, an improved coding for the bottom level would be as follows.

Code 00 - $M_{new} := M_{old}$;
Code 01 - $M_{new} := M_{old} + V_{old}/2$;
Code 10 - $M_{new} := M_{old} + V_{old}$;
Code 11 - $M_{new} := M_{old} + 3V_{old}/2$;

Further, we expect that the bottom level will require lower variance than higher levels. Thus, we prefer to modify the code for level 1 as follows.

Code 00 - $M_{new} := M_{old}$; $V_{new} := V_{old}/2$;
Code 01 - $M_{new} := M_{old}$; $V_{new} := V_{old}$;
Code 10 - $M_{new} := M_{old} + V_{old}/2$; $V_{new} := V_{old}/2$;
Code 11 - $M_{new} := M_{old} + V_{old}$; $V_{new} := V_{old}/2$;

Note that this version changes the meaning of only code value 11 from the original, decreasing the variance. This works especially well in conjunction with the described modification for level 0 which allows more flexibility for smaller variances. Figure 4 shows the elevation grid for Figure 2 as it would be coded using these modified functions. In this example, only the lower right pixel varied too quickly for the coding scheme to faithfully reproduce. Algorithm 3 presents the modified MAKECHILDCODE function.

A fourth class of variants has to do with the number of bits stored at each node of the pyramid. Storing more bits would allow for a greater number of codes. This in turn would allow for more flexibility in representing images with high local variance. In fact, we can view the 2 bit per pixel elevation pyramid as being at one extreme of a continuum of representations whose other end would store complete elevation data at every node, as in the traditional image pyramid [Tani75]. A space-efficient modification that is worthy of future examination is to store an extra 2 bits for internal nodes of the pyramid. Since $\frac{3}{4}$ of the nodes are at the bottom level, this would raise the amortized storage requirements only from $\frac{8}{3}$ to $\frac{10}{3}$ bits/pixel.

A fifth class of variants is based on use of a bintree pyramid instead of a quadtree pyramid [Tamm84]. A bintree pyramid would store at level 1 a $2^n \times 2^{n-1}$ array with

each cell corresponding to 2 cells at level 0; level 1 would be a $2^{n-1} \times 2^{n-1}$ array; and so on. Since the bintree pyramid has as many internal nodes as leaf nodes, the storage requirements will be 4 bits/pixel. However, the bintree provides additional control over the elevation and variance values, allowing more accurate representation. This is because each internal node of the quadtree is equivalent to three internal nodes in the bintree, arranged in a triangle. The result is that for each internal node along a path in the quadtree, there will be two equivalent nodes in the bintree. This allows for more opportunity to correctly modify the values. Another alternative is to use a bintree, but store only 1 bit/node instead of two. Odd levels would store, e.g., V bits, even levels would store M bits (thereby making the final modification be to the pixel's minimum value). This would require only 2 bits/pixel total storage. The relative abilities of these variants to faithfully represent elevation data as compared to the 2 bit/node quadtree method await further investigation.

A final consideration is the order of storage for nodes in the pyramid. Heaps [Aho83] are traditionally stored in breadth-first order. It may be the case that for large, disk-based images, that a depth-first storage order is preferable to minimize disk accesses during traversal.

5. ARCHIVAL STORAGE, RETRIEVAL, AND TRANSMISSION

The primary motivation for the elevation pyramid technique is not so much to store elevation data for direct manipulation, but rather for large scale archival with relatively fast retrieval. Using the elevation pyramid, large portions of the earth's surface could be stored on disk or CD-ROM. In this application, the user would like to retrieve portions of the database for further processing. Such retrieval must be done quickly, so long term archival methods are not desirable if they slow the retrieval process. The region to be represented may not be a simple square region of land, but may include significant portions of ocean or lake. We suggest that a quadtree be used to represent the entire map, perhaps using a scheme to account for the spherical shape of the earth as reported in [Mark87]. This upper-level quadtree would be used primarily to distinguish ocean from land. At a certain level in the tree, those nodes that do not represent entirely water would store a pointer to an elevation pyramid. Additional savings can be obtained by storing a single node in the quadtree in lieu of any large flat area, not just the ocean. Such a representation would allow for the efficient representation of elevation for large portions of the earth's surface.

Given an elevation pyramid encoding, it may be desirable to extract a smaller region, possibly at some intermediate scale. For example, given an elevation pyramid archived at size 8K $\times$ 8K at 1:100,000 resolution, the user may desire a 512×512 subtree. This is easily obtained by traversing from the root of the pyramid to the appropriate internal node representing the root of the subtree containing the desired region, and generating only the complete elevation data for that subtree. If arbitrarily positioned regions are desired, windowing techniques such as described in [Shaf89] are appropriate. Images at double scale, quadruple scale, etc. are easily obtained by stopping before

reaching the bottom of the pyramid. Scales that are not powers of two above the base scale can also be obtained, but would require resizing and interpolating.

The motivation for the progressive transmission techniques mentioned at the end of Section 2 are to support efficient transmission of images. This was facilitated both by compact storage, and by the ability to generate ever improving resolution for the image. The elevation pyramid can be used for progressive transmission of elevation data in the same sense. The upper levels of the pyramid can be transmitted first, followed by lower levels, providing ever improving resolution.

6. STORAGE REQUIREMENTS

It should be clear that the elevation pyramid provides enormous space savings for topographic data as compared to the standard grid method when the topography data has high resolution, e.g., 8-16 bits/pixel. Region quadtrees and related data structures must expect to require even more storage to represent elevation data since they require additional overhead with no benefit from aggregation of data. One might expect, however, that the region quadtree would perform better for contour maps, such as the one shown in Figure 5. This image shows the 100 foot contour levels from part of a USGS quadrangle depicting the Russian river region in northern California. This map is about as simple a set of topographic data as would ever likely be used.

Surprisingly, the elevation pyramid compares quite favorably to the quadtree even for this simple image. The original image is 400×450 pixels, yielding an image array of 360 kbytes if each pixel requires 16 bits. Since there are only 11 contour levels in this particular image, we could compress this to 4 bits/pixel, requiring 80 kbytes. The elevation pyramid would require about 350,000 nodes at 2 bits/node. This gives a total of 87 kbytes of storage, regardless of the vertical resolution. The quadtree for this image requires about 25,000 leaf nodes. As an example of an actual quadtree GIS implementation, the **QUILT** system [Shaf86], which uses a linear quadtree [Garg82] organized by a B-tree for efficient access requires 8 bytes/node, spread onto a disk file requiring a total of about 270 kbytes. However, this implementation does not provide great storage savings over the array; its goal it to provide efficient manipulation of maps.

A storage optimizing linear quadtree implementation might forego the B-tree in favor of a simple linear list. The data portion of the linear quadtree node could be reduced to 2 bytes, requiring a total of 6 bytes/node. This yields a total requirement of 144 kbytes. Even better space performance could be obtained by a DF-expression [Kawa80] representation for the quadtree. This structure could store full resolution nodes requiring only 2 bytes/node, but would store 4/3 the total number of leaf nodes (about 25,000). Thus the total storage requirements would be 67 kbytes. Using only 4 bits/node would reduce the requirements to only 17 kbytes. This is much better than the most compact array or the elevation pyramid. However, this relies on (1) a very sparse quadtree; (2) low vertical resolution; and (3) sacrifices random access to the quadtree nodes.

The main points to be observed are that the elevation pyramid is easily manipulated (i.e., windows and point queries can easily be performed), has high vertical resolution, and still compares favorably in storage requirements to the array or quadtree under all but the most favorable conditions for those latter structures.

7. CONCLUSIONS AND FUTURE WORK

The elevation pyramid allows for full gridded elevation data to be stored in a manner that requires less than 3 bits/pixel. It's storage and time requirements should compare quite favorably to traditional non-grid methods for storing elevation data. Many variants on the basic scheme are possible, and a few are described. No doubt, better refinements will be discovered in the future. Extension of the elevation pyramid to 3 dimensions (for example, to store 3-D densities) are straightforward.

Our plans for the immediate future are to test the elevation pyramid algorithms on wide range of elevation data to determine the amount of information loss in typical images. We will implement and compare several of the variants described. Finally, we will develop characterizations for the types of images that can be encoded without loss of information for the most promising variants.

8. ACKNOWLEDGEMENTS

I would like to thank Dave Boldery for implementing the algorithms described in this paper, and for double-checking and correcting my calculations.

9. REFERENCES

1. [Aho83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison Wesley, Reading MA, 1960.

2. [Anto86] R. Antony and P.J. Emmerman, Spatial reasoning and knowledge representation, in *Geographic Information Systems in Government*, vol. 2, B.K. Opitz, Ed., A. Deepak Publishing, Hampton, VA, 795-813.

3. [Cebr85] J.A. Cebrian, J.E. Mower, and D.M. Mark, Analysis and display and digital elevation models within a quadtree-based geographic information system, *Proceedings, Auto-Carto 7*, Washington, D.C., 1985, 55-65.

4. [Chen86] Z.T. Chen and W.R
Proceedings of Auto-Carto Londo

5. [Dutt83] G.H. Dutton, Efficie
for processing land data with a
Land Policy Monograph Series, 1

6. [Fred60] E. Fredkin, Trie me
1960), 490-499.

7. [Garg82] I. Gargantini, An eff
the ACM 25, 12(December 1982

8. [Hard84] D.M. Hardas and S.
ing coded binary trees: Algorit
Analysis and Machine Intelligen

9. [Hill83] F.S. Hill, Jr., W. Shel
using progressive transmission, C

10. [Kawa80] E. Kawaguchi and
and its application to data com
Machine Intelligence 2, 1(Janua

11. [Kell77] R.E. Kelly, E.P.H.
tomapping system, *Photogramm*

12. [Know80] K. Knowlton, Pro
by simple, efficient, and lossless
1980), 885-896.

13. [Leif87] L.A. Leifer and D.M
using quadtrees and orthogonal
MD, 1987, 650-659.

14. [Mand82] B.B. Mandelbrot,
1982.

15. [Mark85] D.M. Mark and J.
information systems at continent
ington, D.C., 1985, 355-364.

16. [Mort66] G.M. Morton, A co
in file sequencing, IBM Canada.

4. [Chen86] Z.T. Chen and W.R. Tobler, Quadtree representations of digital terrain, *Proceedings of Auto-Carto London*, Vol. 1, London, September 1986, 475-484.

5. [Dutt83] G.H. Dutton, Efficient encoding of gridded surfaces, in *Spatial algorithms for processing land data with a microcomputer*, Cambridge MA: Lincoln Institute for Land Policy Monograph Series, 1983.

6. [Fred60] E. Fredkin, Trie memory, *Communications of the ACM 3*, 9(September 1960), 490-499.

7. [Garg82] I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

8. [Hard84] D.M. Hardas and S.N. Srihari, Progressive refinement of 3-D images using coded binary trees: Algorithms and architecture, *IEEE Transactions on Pattern Analysis and Machine Intelligence 6*, 6(November 1984), 748-757.

9. [Hill83] F.S. Hill, Jr., W. Sheldon, Jr., and F. Gao, Interactive image query system using progressive transmission, Computer Graphics 17, 3(July 1983), 323-330.

10. [Kawa80] E. Kawaguchi and T. Endo, On a method of binary picture representation and its application to data compression, *IEEE Transactions on Pattern Analysis and Machine Intelligence 2*, 1(January 1980), 27-35.

11. [Kell77] R.E. Kelly, E.P.H. McConnell, and S.J. Mildenberger, The Gestalt photomapping system, *Photogramm. Engng Rem. Sens. 43*, (11), 1407-1417.

12. [Know80] K. Knowlton, Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes, *Proceedings of the IEEE 68*, 7(July 1980), 885-896.

13. [Leif87] L.A. Leifer and D.M Mark, Recursive approximation of topographic data using quadtrees and orthogonal polynomials, *Proceedings of Auto-Carto 8*, Baltimore, MD, 1987, 650-659.

14. [Mand82] B.B. Mandelbrot, *The Fractal Geometry of Nature*, Freeman, New York, 1982.

15. [Mark85] D.M. Mark and J.P. Lauzon, Approaches for quadtree-based geographic information systems at continental or global scales, *Proceedings of Auto-Carto 7*, Washington, D.C., 1985, 355-364.

16. [Mort66] G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, IBM Canada, 1966.

17. [Niev84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik, The grid file: an adaptable, symmetric multikey file structure, *ACM Transactions on Database Systems 9*, 1(March 1984), 38-71.

18. [Peuk78] T.K. Peuker, R.J. Fowler, J.J. Little, and D.M. Mark, The triangulated irregular network, in *Proceedings of the DTM Symposium, American Society of Photogrammetry - American Congress on Survey and Mapping*, St. Louis, MO, 1978, 24-31.

19. [Same84] - H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

20. [Same85a] H. Samet, A top-down quadtree traversal algorithm, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 1 (January 1985), 94-98.

21. [Same85b] H. Samet and R.E. Webber, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182-222.

22. [Shaf87a] - C.A. Shaffer, H. Samet, and R.C. Nelson, **QUILT**: a geographic information system based on quadtrees, University of Maryland TR 1885, July 1987.

23. [Shaf87b] C.A. Shaffer and H. Samet, An in-core hierarchical data structure organization for a geographic database, Computer Science TR 1886, University of Maryland, College Park, MD, July 1987.

24. [Shaf89] C.A. Shaffer and H. Samet, Set operations for unaligned linear quadtrees, to appear in *Computer Vision, Graphics, and Image Processing*, also Department of Computer Science TR 88-31, Virginia Polytechnic Institute and State University, Blacksburg, VA, September 1988.

25. [Sloa79] K.R. Sloan and S.L. Tanimoto, Progressive refinement of raster images, *IEEE Transactions on Computers 28*, 11(November 1979), 871-874.

26. [Tamm81] M. Tamminen, The EXCELL method for efficient geometric access to data, *Acta Polytechnica Scandinavia*, Mathematics and Computer Science Series No. 34, Helsinki, 1981.

27. [Tamm84c] - M. Tamminen, Encoding trees, *Computer Vision, Graphics, and Image Processing 28*, 1(October 1984), 44-57.

28. [Tani75] S. Tanimoto and T. Pavlidis, A hierarchical data structure for picture processing, *Computer Graphics and Image Processing 4*, 2(1975), 104-119.

10. APPENDIX

```
function DECODE(root : ↑ P
  { Generate the elevation valu
    and size 2^n × 2^n. ALLMIN
    the entire pyramid. While r
    nodes, in practice they wo
    pyramid. }
var
  currnode : ↑ PYR;
  m, v, currx, curry, currlev, h
begin
  currnode := root; m := ALL
  currx := 0; curry := 0; curr
  repeat
    currlev := currlev − 1; hal
    if x >= currx+ half then
      if y >= curry+ half th
        begin child := SW; c
      else begin child := NW
    else if y >= curry+ half
      begin child := SE; curr
    else child := NE;
    case currnode.value[child
      '00': v := v/2;
      '01': ;
      '10': begin v := v/2; m
      '11': m := m + v
    end{ Case }
    currnode := CHILDOF(c
  until currlev = 0;
  return (m)
end;
```

Algorithm 1. Decoding algorit

, The grid file: an adaptable,
)atabase Systems 9, 1(March

.M. Mark, The triangulated
ı, American Society of Pho-
, St. Louis, MO, 1978, 24-31.

chical data structures, ACM

gorithm, IEEE Transactions
ry 1985), 94-98.

tion of polygons using quad-
2-222.

QUILT: a geographic infor-
ı TR 1885, July 1987.

chical data structure organi-
386, University of Maryland,

ınaligned linear quadtrees, to
ng, also Department of Com-
State University, Blacksburg,

refinement of raster images,
, 871-874.

efficient geometric access to
Computer Science Series No.

Vision, Graphics, and Image

ıl data structure for picture
2(1975), 104-119.

10. APPENDIX

```
function DECODE(root : ↑ PYR; x, y : integer) : ELEVATION;
  { Generate the elevation value at point (x, y) in the pyramid with root pointer root
    and size 2^n × 2^n. ALLMIN and ALLVAR is the minimum and total variance for
    the entire pyramid. While root and currnode are indicated as pointers to pyramid
    nodes, in practice they would likely be indexes into an array implementing the
    pyramid. }
var
  currnode : ↑ PYR;
  m, v, currx, curry, currlev, half : integer;
begin
  currnode := root; m := ALLMIN; v := ALLVAR;
  currx := 0; curry := 0; currlev := n;
  repeat
    currlev := currlev − 1; half := 2^currlev;
    if x >= currx+ half then
      if y >= curry+ half then
        begin child := SW; currx := currx+ half; curry := curry+ half end
      else begin child := NW; currx := currx+ half end
    else if y >= curry+ half then
      begin child := SE; curry := curry+ half end
    else child := NE;
    case currnode.value[child] of
      '00': v := v/2;
      '01': ;
      '10': begin v := v/2; m := m + v end;
      '11': m := m + v
    end{ Case }
    currnode := CHILDOF(currnode, child);
  until currlev = 0;
  return (m)
end;
```

Algorithm 1. Decoding algorithm.

```
procedure PASS2(root : ↑ NODE; n : integer);
  { Second pass of pyramid construction. Produces the pyramid in depth-first order.
    root is the root of a subtree in the intermediate pyramid containing minimum and
    variance values for each node. }
var
  q : integer;
begin
  for q in { NW, NE, SW, SE } do
    OUTPUT(MAKECHILDCODE(root.min, root.var, root.child[q].min,
                          root.child[q].var));
  if n <> 1 do
    for q in { NW, NE, SW, SE } do PASS2(root.child[q], n - 1)
end; { PASS2 }

function MAKECHILDCODE(currmin, currvar : integer;
                       var childmin, childvar : integer) : CODE
  { Create the child's code. In the process, set the child's computed estimage of mini-
    mum and variance values into the pyramid for future use. }
var code : CODE;
begin
  if ((currmin + currvar) <= childmin) then
    begin code := '11'; childmin := currmin + currvar; childvar := currvarend
  else if ((currmin + currvar/2) <= childmin) then begin
    code := '10'; childmin := currmin + currvar/2; childvar := currvar/2 end
  else if (currvar <= childvar) then
    begin code := '01'; childmin := currmin; childvar := currvar end
  else
    begin code := '00'; childmin := currmin; childvar := currvar/2 end
end; { MAKECHILDECODE }
```

Algorithm 2. Encoding algorithm: second pass.

```
function MAKECHILDCODE(cu
    var childmin, childvar : int
var code : CODEVAL;
begin
  if level <> 0 then { not botton
    if ((currmin + currvar) <=
      begin code := '11'; childm
        if level <> 1 then child
      end
    else if ((currmin + currvar/
      begin
        code := '10'; childmin :=
    else if (currvar <= childva
      begin code := '01'; childn
    else begin code := '00'; chi
  else { bottom level }
    if ((currmin + 3 * currvar/
    else if ((currmin + currva
    else if ((currmin + currva
    else code := '00';
end; { MAKECHILDECODE }
```

Algorithm 3. Modified encodin
the pyramid.

```
function MAKECHILDCODE(currmin, currvar : integer;
      var childmin, childvar : integer): CODE; { Modified child code generator. }
var code : CODEVAL;
begin
  if level <> 0 then { not bottom level }
    if ((currmin + currvar) <= childmin) then
      begin code := '11'; childmin := currmin + currvar;
        if level <> 1 then childvar := currvar else childvar := currvar/2
      end
    else if ((currmin + currvar/2) <= childmin) then
      begin
        code := '10'; childmin := currmin + currvar/2; childvar := currvar/2 end
    else if (currvar <= childvar) then
      begin code := '01'; childmin := currmin; childvar := currvar end
    else begin code := '00'; childmin := currmin; childvar := currvar/2 end
  else { bottom level }
    if ((currmin + 3 * currvar/2) <= childmin then code := '11'
    else if ((currmin + currvar) = childmin then code := '10'
    else if ((currmin + currvar/2) = childmin then code := '01'
    else code := '00';
end; { MAKECHILDECODE }
```

Algorithm 3. Modified encoding algorithm with special codes at bottom two levels of the pyramid.

0	1	1	1	2	3	4	4
1	2	2	2	2	3	4	4
1	2	3	3	3	4	4	4
1	1	2	2	3	4	4	4
2	2	2	2	3	4	5	6
3	2	2	2	3	4	5	6
4	3	3	3	3	4	5	6
4	3	3	3	4	4	6	6

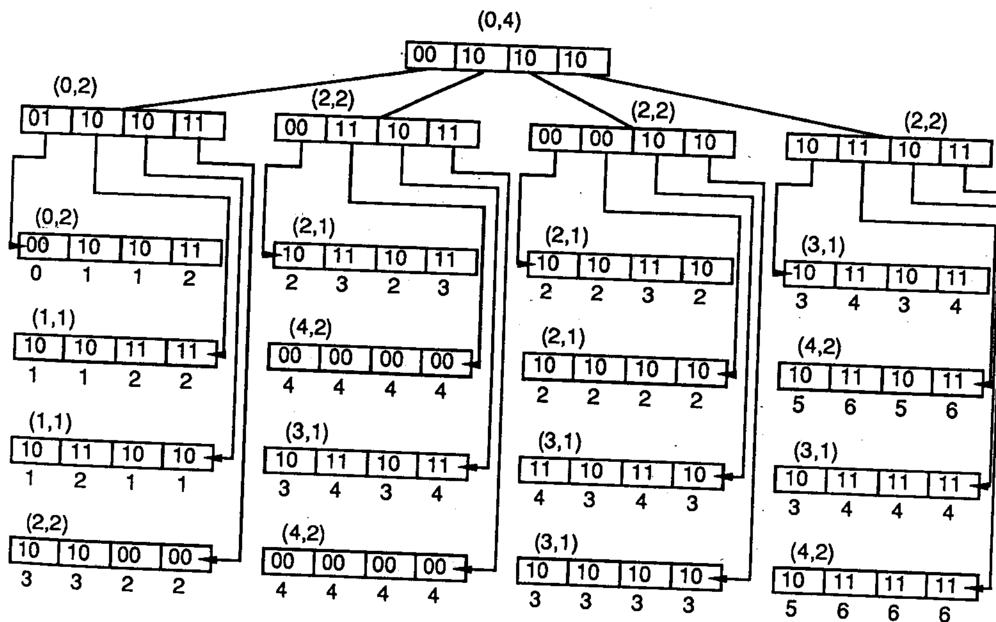


Figure 1. An elevation grid and its elevation pyramid using standard coding.

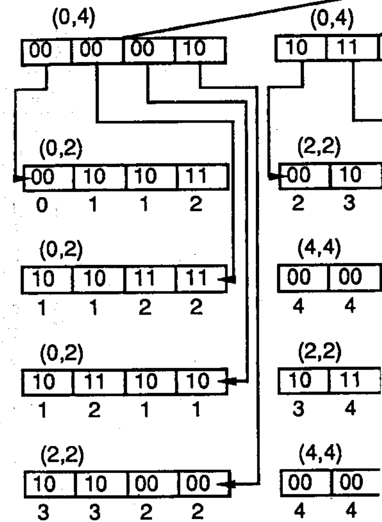
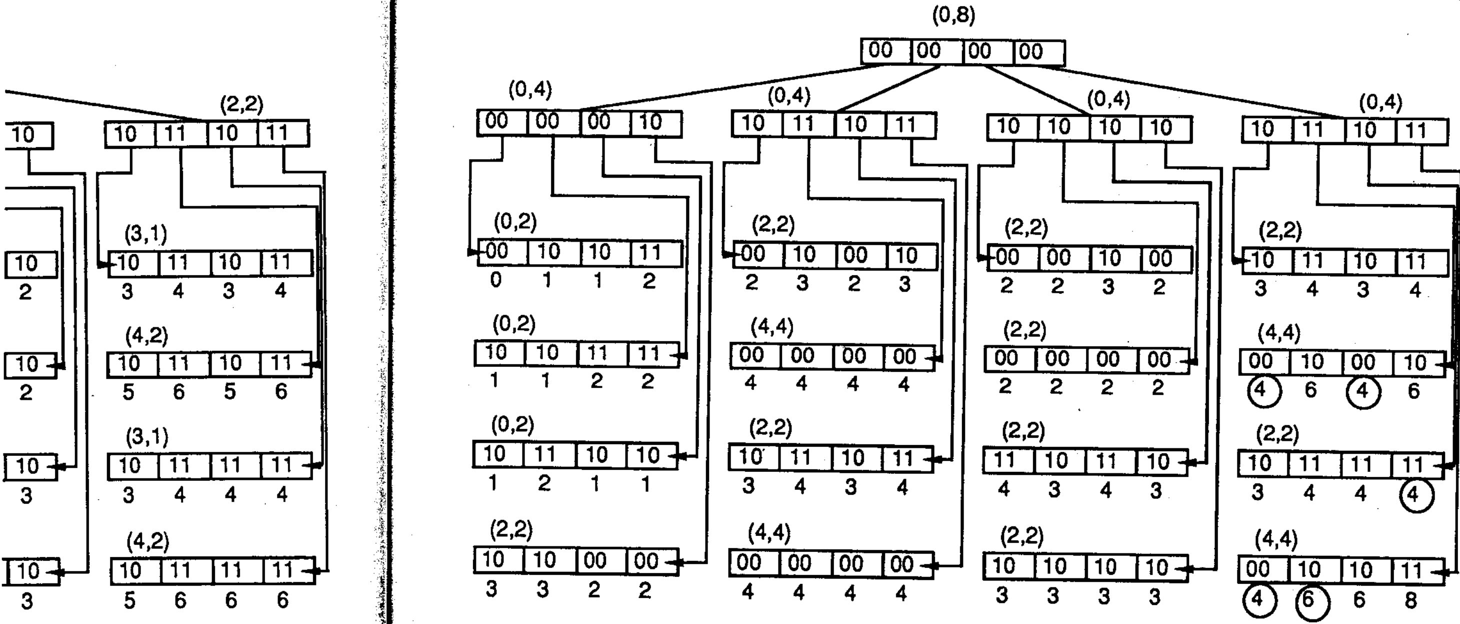


Figure 2. Elevation grid an
grid whose values rise too r
values. Incorrectly valued p

0	1	1	1	2	3	4	4
1	2	2	2	2	3	4	4
1	2	3	3	3	4	4	4
1	1	2	2	3	4	4	4
2	2	2	2	3	4	5	6
3	2	2	2	3	4	5	6
4	3	3	3	3	4	5	7
4	3	3	3	4	5	6	8



id using standard coding.

Figure 2. Elevation grid and pyramid using standard coding for an elevation grid whose values rise too rapidly for the pyramid to faithfully represent all grid values. Incorrectly valued pixels are circled.

8	7	7	7	6	5	4	4
7	6	6	6	6	5	4	4
7	6	5	5	5	4	4	4
7	7	6	6	5	4	4	4
6	6	6	6	5	4	3	2
5	6	6	6	5	4	3	2
4	5	5	5	5	4	3	1
4	5	5	5	4	3	2	0

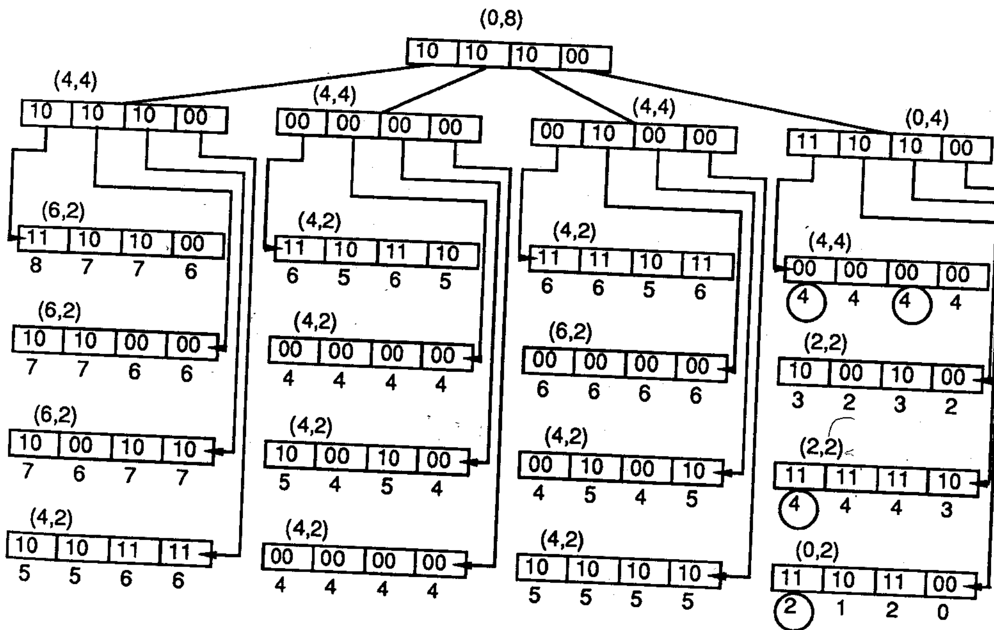


Figure 3. Elevation grid and pyramid for the same grid as in Figure 2, except that all values v have been replaced by 8 - v. Incorrectly labeled values in the pyramid are circled.

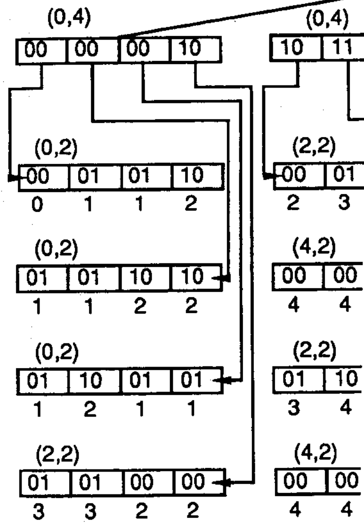


Figure 4. The elevation gri
level-modified coding rules.

0	1	1	1	2	3	4	4
1	2	2	2	2	3	4	4
1	2	3	3	3	4	4	4
1	1	2	2	3	4	4	4
2	2	2	2	3	4	5	6
3	2	2	2	3	4	5	6
4	3	3	3	3	4	5	7
4	3	3	3	4	5	6	8

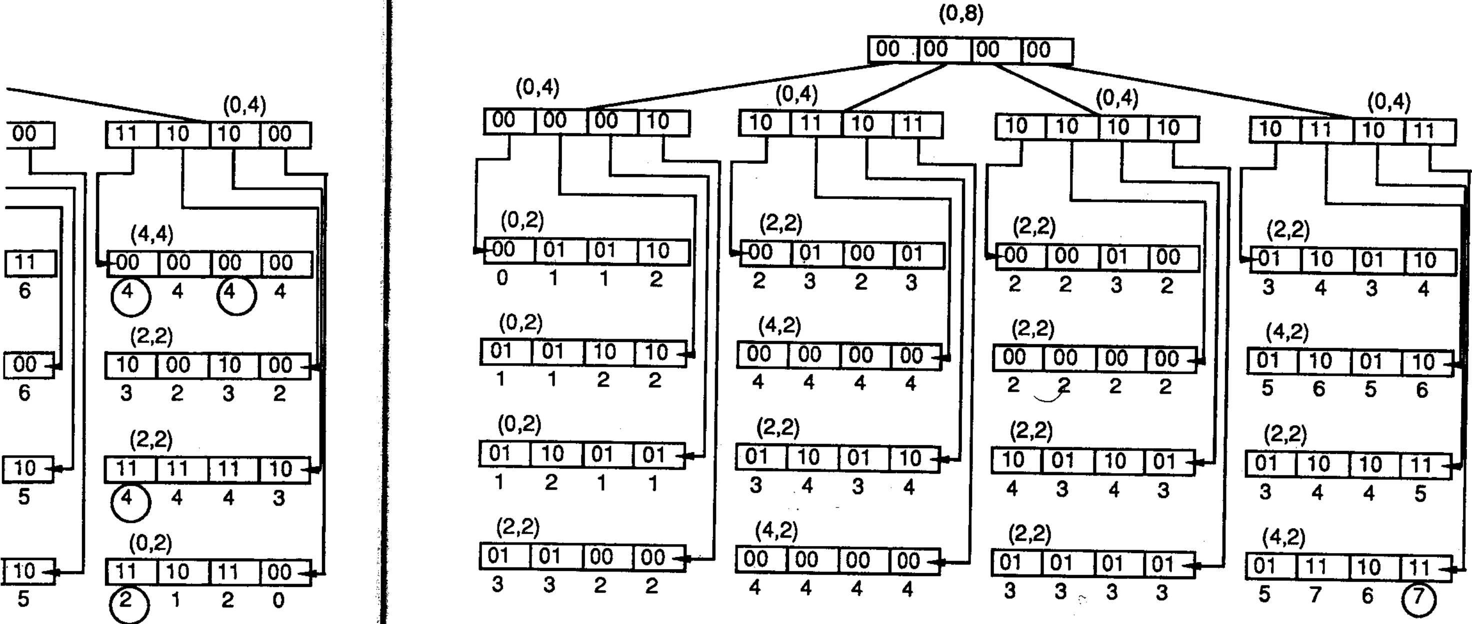


Figure 4. The elevation grid of Figure 2 and its elevation pyramid derived using the level-modified coding rules. Pixels incorrectly labeled by the pyramid are circled.

as in Figure 2, except that
ɔeled values in the

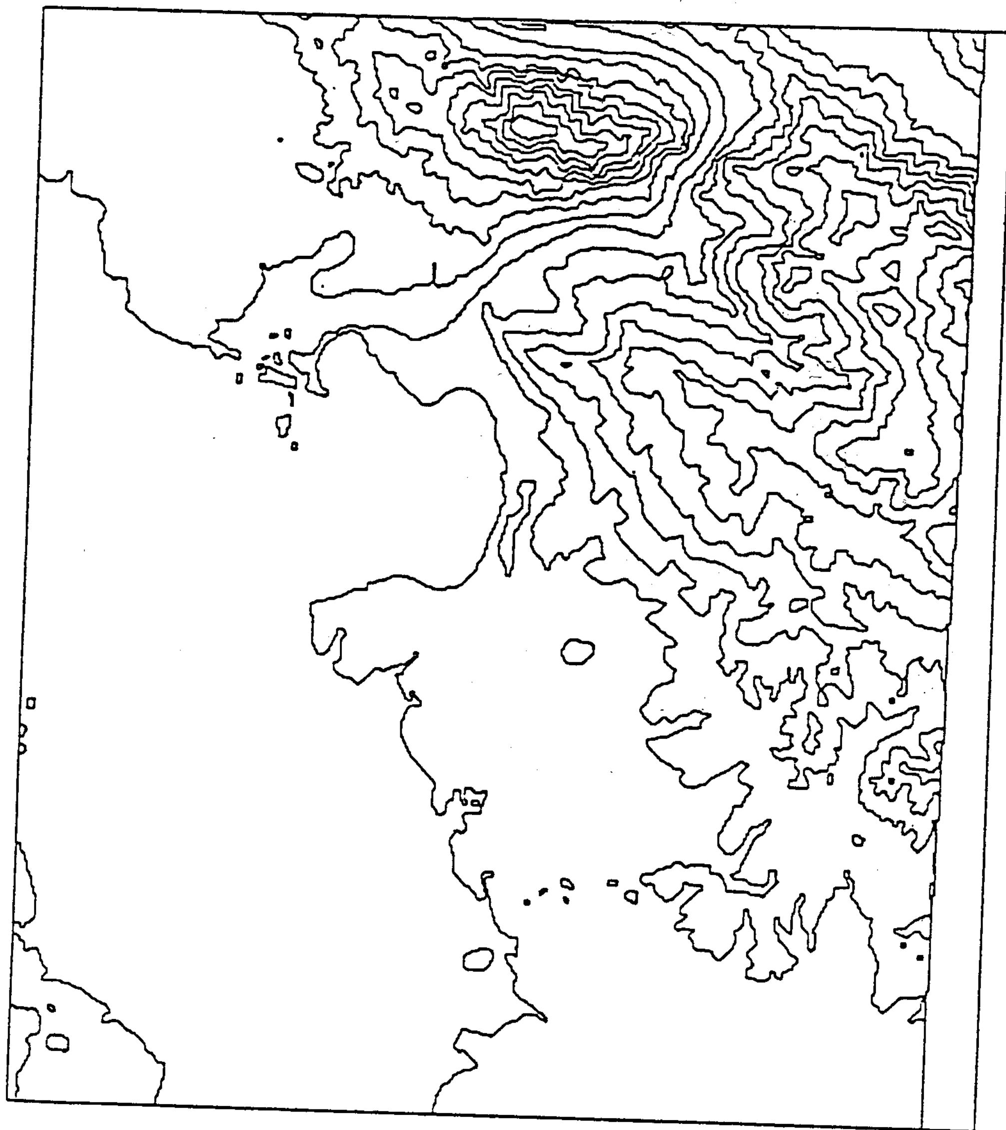


Figure 5. A simple elevation map.

System a