The Elevation Pyramid

By Clifford A. Shaffer and Dave B. Boldery

TR 90-29

THE ELEVATION PYRAMID

Clifford A. Shaffer
Dave B. Boldery

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

ABSTRACT

The elevation pyramid, a pyramid-based representation for storing gridded elevation data, is described. Associated with he root of the pyramid is the corresponding grid's minimum elevation and range. The elevation value for a specified grid pixel is calculated by traversing a path from the pyramid root to the corresponding leaf node. As the traversal proceeds, the minimum and range values are refined by interpreting the codes stored at each node along the path. At the leaf level, the final minimum value equals the associated elevation value. We present results from experiments using 2, 3 and 4 bit code words. For the two bit code, 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 corresponds to a 5:1 compression rate for 16 bit gridded elevation data. The elevation pyramid 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". Rapid changes in elevation between adjacent pixels will be smoothed. Most data sets contain relatively few pixels that cannot be perfectly encoded by the techniques studied. Such pixels can efficiently be stored in an auxilliary table if perfect reconstruction is required. "Elevation" pyramids can be used to store any 2D surface or 3D density data.

This work was partially supported by General Dynamics and by the Virginia Center for Innovative Technology. A preliminary version of this paper appeared as [Shaf90a].

1. INTRODUCTION

As scientific studies of the earth advance, so too does the need for more accurate and diverse geographic data. Today, storing the sheer volume of data generated is one of the most significant problems facing the earth sciences. Geographic data can be classified into two general categories: *cloroplethic* and *topographic* data. From the representation standpoint, the most important characteristic of cloroplethic data is that it typically contains large uniform areas. Storage requirements can be reduced considerably through data structures which capitalize on this uniformity. Examples of traditional representations for cloroplethic data include chain codes, run-length codes, vector based topological representations, block codes, and quadtrees (see [Burr86] for a complete discussion). Unfortunately, topographic data (which for the purposes of this discussion can include the representation of any surface in 2D or density data in 3D), unless extremely coarse, typically does not contain uniform regions. Since elevation values are usually different even between adjacent samples, methods suitable to representing uniform data are not appropriate.

The need for high quality topographic data combined with the difficulties encountered when processing such large volumes of data have resulted in many representations. The most common is the Digital Elevation Model, which typically stores 8 or 16 bit elevation values in a matrix or grid. The size of this grid for large areas may be unacceptable. Other techniques attempt to generalize the data into smaller sets which are then stored and manipulated. One such method is to store a collection of benchmark points along with their associated elevation values. Assuming appropriate point selection has been made, values for locations not explicitly stored can be derived by interpolating the values of nearby data points. This approach is time consuming since the nearby points must be located, and the interpolation performed. Various interpolation procedures (again, see [Burr86] for examples) are possible, but their accuracy is questionable. One popular variation on the benchmark point approach is the Triangulated Irregular Network [Peuk78].

The quadtree [Same89, Same90] 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 [Shaf90c] 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 that require spatial indexing. For the remainder of this section, we loosely use the term "quadtree" to refer to all such spatial indexing methods.

The quadtree approach attempts to save both space and time over 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 – see [Same90]). Such approaches have not been successful in the past for representing elevation data since such data does not fall into either category. The region quadtree will not be space efficient since there are no homogeneous blocks to aggregate. Another approach to applying a quadtree representation to topographic data is to derive a relatively small collection of spatial data objects from the data, which will then be stored by means of a spatial index. For example, a collection of surface patches can be generated that approximate the topographic surface. [Chen86, Leif87] apply quadtree indexing to this technique.

The benchmark point method may appear more amenable to spatial indexing methods since the data points could be indexed with a Point-Region (PR) quadtree [Same89] 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. If the number of data points is high in relation to the desired resolution, then 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 a 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.

This paper presents a new approach to representing elevation data in compressed form,

with little or no degradation from the elevation grid from which it is derived, depending on the smoothness of the surface modeled. Termed the *elevation pyramid*, our data structure uses an image pyramid structure [Tani75] to allow aggregation of the elevation data at higher levels in the pyramid. As compared to the elevation grid, the new representation reduces storage requirements, while still allowing random access to the data. Depending on the roughness of the elevation data and the faithfulness of the reconstruction required, data sets are represented at an amortized cost of between 2.67 and 5.3 bits per grid cell. The new representation can be viewed as a generalization of the DEPTH coding scheme of Dutton[Dutt83, Dutt84], although our approach uses a very different encoding method. An error-free data amortization scheme for gray-scale images based on quadtrees and variable length codes is presented in [Dürs89].

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 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.

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. 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. [Dürs89] applies the trie to compressing elevation data in precisely this way. One drawback to trie methods is that information about the structure of the tree must be maintained.

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 some form of "average" value for the image, and utilize a pyramid structure to store "differences" indicating how the refinement takes place as one proceeds through the pyramid. 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 progressive transmission methods do not utilize data aggregation for storage compression.

3. THE ELEVATION PYRAMID

The pyramid [Tani75] can be viewed as a stack of arrays which stores at the bottom (or 0'th) level the entire image of size $2^n \times 2^n$. At the next level, each disjoint 2×2 block of grid cells 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 to the nth level, where the entire grid is represented by 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.

Our initial formulation for the elevation pyramid represents an elevation grid as follows. Associated with the pyramid are two global variables, *allmin* and *allrng*. Initially, *allmin* contains the minimum elevation value within the grid, and *allrng* equals the greatest power of 2 less than or equal to the difference between the minimum and maximum elevation values within the grid. Each node of the pyramid stores a 2-bit code value which determines how the parent's minimum and range values are to be refined. To determine the elevation for a given grid cell, a path is taken through the pyramid from the root to the leaf node associated with the cell. As the traversal proceeds, *allmin* and *allrng* are refined based upon the code stored at each node along the path. When the leaf node is reached, and the final refinement of *allmin* is performed, its value equals the reconstructed elevation for the corresponding grid cell. Figure 1 shows a sample elevation grid and the corresponding elevation pyramid. Note the pointers in Figure 1 are drawn between nodes purely as a visual aid since the actual implementation stores the nodes as a heap.

For this version of the elevation pyramid, each internal node is 8 bits long, divided into 4 fields of 2 bits each. Strictly speaking, these four fields contain the code required to modify the

local copies of *allmin* and *allrng* for the node's four children. This groups sibling codes to form a single byte value. Since the four 2-bit code values are grouped into a single node, there are only n levels in the pyramid for a $2^n \times 2^n$ grid. Since level i contains $4^{(n-i)}$ nodes, where $1 \leq i \leq n$, the complete pyramid contains $\frac{4^n-1}{3}$ or about $\frac{1}{3}N$ nodes where N is the number of grid cells. The total amount of storage is $\frac{8}{3}N$ bits or $\frac{8}{3}$ bits per grid cell.

In general, the first bit of a code indicates modification to the minimum elevation for the corresponding subtree, while the second bit indicates modification to the range. Specifically, the two bits are defined by the following operations on the current copies of *allmin* and *allrng* (denoted as M_{old} and R_{old}, respectively). The new values of *allmin* and *allrng* (denoted as M_{new} and R_{new}, respectively) are passed to the child for future operations. The coding method is as follows:

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

Algorithm 1 of the Appendix presents a Pascal-like pseudocode function which decodes and returns the elevation value at any specified grid cell. The function DECODE_PT follows a path from the root to the level 1 node associated with the query grid cell to evaluate the elevation value. When an entire grid is to be evaluated, DECODE_PT is inefficient since the pyramid has to be traversed separately for each value. A procedure which traverses the entire pyramid in a depth-first fashion, visiting each node only once, can easily be derived from Algorithm 1.

Our encoding procedure creates the elevation pyramid in two steps. The first creates a structure termed the *temporary pyramid*, which stores at each node the minimum elevation and range for the associated region of the original elevation grid. For example, the root contains the minimum and range for the entire grid, and its four children each contain the minimum and range of their associated quadrant. At level 0 each node contains the actual elevation value of the associated grid cell. Figure 2 shows the temporary pyramid as it would represent the elevation grid of Figure 1. A grid of size $2^n \times 2^n$ requires a temporary pyramid with $n+1$ levels. Assuming minimum, range,

and elevation values each require two bytes, and that only internal nodes store range values, a temporary pyramid will occupy approximately $(\frac{10}{3})4^n$ bytes. Procedure PASS1 of Algorithm 2 builds the temporary pyramid in main memory with a globally defined elevation grid.

The temporary pyramid is used to construct the final elevation pyramid as follows. Variables *allmin* and *allrng* are set to the minimum and the greatest power of two less than or equal to the range of the entire elevation grid, respectively. The minimum and range values stored in the root of the temporary pyramid are then replaced by *allmin* and *allrng*. For each child of the root, *allmin* and *allrng* are compared to the child's minimum and range. Based upon this comparison, a code which best refines *allmin* and *allrng* to the minimum and range of the child node is selected. After a code is selected, the refined values of *allmin* and *allrng* replace the actual minimum and range in the child node. The codes derived for the root's four children make up the one byte value stored at the root of the elevation pyramid. This process repeats for each internal node of the temporary pyramid. As the lower levels of the temporary pyramid are reached, the value of *allmin* approaches the true elevation value. The function MAKE_CHILD_CODE$_1$ generates the code based on our initial coding method. This function compares the current values of *allmin* and *allrng* (associated with the parent node) to the corresponding minimum and range values of the child nodes. The proper code is selected and returned, and the minimum and range values for each child are set to the refined *allmin* and *allrng*.

Subtle changes in the code generation technique can lead to significant variations in the quality of the codes produced. For example, the range value is defined as the greatest power of 2 *less than or equal to* the true range, as opposed to the greatest power of 2 *less than* the true range. Otherwise, a range of 1 would always be reduced to 0, after which *allmin* could never be changed for any child cells. An important feature of MAKE_CHILD_CODE$_1$ is that it first checks, and if possible modifies, the minimum value for the child without regard to the child's range. In the case of code 11, this could lead to rapid increases of the minimum value while maintaining the range

even when an area's true range is rather low. However, in general it is reasonable to expect that if the minimum value is changing rapidly, the range will remain high. Conversely, code 10 allows a moderate increase in the minimum value, but at the same time forces a decrease in the range. One might be concerned that an area with a slightly higher minimum value but a high range might be incorrectly coded since the range is artificially reduced to allow an increase in the minimum value. However, this is normally not a problem since small increases in the minimum value can only occur when the range is already low enough to allow them (since the smallest increase is $R/2$).

The construction procedures described here assume that both the temporary pyramid and the initial elevation grid will fit into internal memory. For large grids, this probably will be impossible. [Bold90] describes algorithms for disk-based construction of elevation pyramids.

4. ELEVATION PYRAMID VARIANTS

Ideally, the elevation pyramid would provide a completely faithful representation of the original elevation grid. The term "faithful" is used instead of "accurate" because the accuracy of the the source for elevation data is often suspect. A 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 3. In this figure, 29 grid cells (whose values are circled) were incorrectly represented. In each incorrect case, the computed value is one less than the true value.

When the elevation pyramid incorrectly represents an elevation point, it is important to realize that the representation is not radically incorrect, but instead represents a smoothing of rapid changes in the elevation. The extent of this problem depends on the relationship between horizontal and vertical resolution. If the original elevation data is not extremely accurate, minor degradation in the representation may be within the range of error for the original data. However, greater faithfulness can be achieved through various modifications to the coding algorithm. We also present approaches for perfect reconstruction of the original data; however, such methods incur

greater penalty in terms of computational overhead and storage.

One class of variants results from modifying the definition of the codes based on the current level in the pyramid. In particular, we note that modifications to the range at the bottom level (level 1) are not significant, except for its affect when added to the minimum. A better coding for the bottom level would be to increase the effect of the range, as follows.

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

Further, we expect that the bottom level will require a lower range than higher levels. Thus, we modify Code 11 for level 2 to be $M_{new} := M_{old} + R_{old}$; $R_{new} := R_{old}/2$. Figure 4 shows the elevation grid for Figure 3 as it would be coded using the level modified codes. Now, only 16 grid cells are incorrectly encoded. The function MAKE_CHILD_CODE$_2$ formalizes the second coding method.

These modifications improve reconstruction performance, however there are still limitations to what can be correctly encoded at the lower levels. For instance, examine the codes that MAKE_CHILD_CODE$_2$ generates for levels one and two of the elevation pyramid. When the value of *allrng* is four and a code is being generated for level one, no code exists which will permit *allmin* to be refined by one or three. Consider the elevation pyramid in Figure 4, specifically the nodes at level one which have incoming values of *allrng* equal to 4. At these nodes the inability to add one or three to *allmin* causes 16 elevation values to be incorrectly coded. Producing an effective code for level two is also difficult when the value of *allrng* is large (i.e., when $allrng \geq 8$). When *allrng* is greater than eight, a difference of four or more must exist between *allmin* and the observed *childmin* before *allmin* is refined to a larger value (i.e., before a code of 11 or 10 is selected). Failure to modify *allmin* may result in a high difference between *allmin* and *childmin* at level one. In fact, the difference could make it impossible for the final remaining two bit code to correctly encode the elevation value. To help reduce the problems at the lowest levels, we can let

the encoding bits represent the quantity to be "added" to the minimum value when *allrng* falls below a specified tolerance value. The most effective way has been to adapt these linear-coding rules at levels one and two of the elevation pyramid when the value of *allrng* falls at or below 8. In these cases, the code is as follows:

Code 00 - $M_{new} := M_{old}$;
Code 01 - $M_{new} := M_{old} + 2^{(level-1)}$;
Code 10 - $M_{new} := M_{old} + 2 * 2^{(level-1)}$;
Code 11 - $M_{new} := M_{old} + 3 * 2^{(level-1)}$;

Note that the change to the minimum is a function of the current level in the elevation pyramid. At level 2, the value of *allmin* may be refined by 0, 2, 4, or 6, and at level 1 the value of *allmin* is refined by 0, 1, 2, or 3. By allowing *allmin* to be refined by larger amounts at level 2, higher local variations are encoded. The above modification improves performance since *allmin* is refined independently of *allrng* when *allrng* becomes an "ineffective" value. Figure 5 presents the elevation pyramid for the same elevation grid with the linear-coding rules. Now, only two elevation values are incorrectly encoded. Function MAKE_CHILD_CODE$_3$ presents this third coding method. In MAKE_CHILD_CODE$_3$, both the level and the value of *allrng* are used to determine how to encode. Our test data (see Section 5) show this technique to be consistently the most effective coding scheme studied of those methods using a two bit code in terms of the number of correctly encoded grid cells.

The previously discussed encoding methods effectively encode data grids which exhibit relatively small local surface variation (i.e., variations of 3 or less between adjacent grid cells). In situations where the grid exhibits larger variations, it can be expected that two bit codes will not provide a faithful reconstruction. Regardless of how well any encoding method works at higher levels in the elevation pyramid, the final result is highly dependent upon the observed range between grid cells at the lower levels in the temporary pyramid. A logical extension is to increase the number of encoding possibilities at lower levels of the pyramid. One technique is to use a three bit code at

levels 3, 2, and 1 in a manner similiar to that used in method 3. Above level 3, it is more efficient to modify *allmin* by one half of (a large) *allrng* value than to add a linear factor. This modification raises the amortized storage requirements to slightly less than 4 bits per grid cell, but it provides a considerable increase in performance. With this technique the codes at the bottom three levels of the elevation pyramid are interpreted using linear encoding as follows.

Code 000 - $M_{new} := M_{old}$;
Code 001 - $M_{new} := M_{old} + 2^{(level-1)}$;
Code 010 - $M_{new} := M_{old} + 2 * 2^{(level-1)}$;
Code 011 - $M_{new} := M_{old} + 3 * 2^{(level-1)}$;
Code 100 - $M_{new} := M_{old} + 4 * 2^{(level-1)}$;
Code 101 - $M_{new} := M_{old} + 5 * 2^{(level-1)}$;
Code 110 - $M_{new} := M_{old} + 6 * 2^{(level-1)}$;
Code 111 - $M_{new} := M_{old} + 7 * 2^{(level-1)}$;

Modifications are again a function of the current level in the pyramid. Thus, at higher levels a larger difference may be added to *allmin* to capture large local variations. When the value of *allmin* at level 1 is refined perfectly by the upper levels of the elevation pyramid, the differences between the grid cells can be less than or equal to six, which is greater than the variation between adjacent pixels for most of our test data. This will be referred to as the fourth method in our empirical comparisons; its implementation is only a slight modification to MAKE_CHILD_NODE$_3$. It should be noted that the three bit codes are more difficult to store and retrieve since they cross the boundaries of adjacent bytes in the heap representation of the encoded pyramid. This problem can be managed through direct bit operators available in many languages (e.g. "C") but at additional cost in processing time.

Our final linear coding method uses a four bit code at the lower three levels of the elevation pyramid, and a two bit code for all higher levels. In using this representation, the amortized cost is raised to about 5.3 bits per grid cell. Although the four bit codes require more space than the previous two methods, they prove to be much easier to process since the codes do not cross byte boundaries. At the bottom four levels, as in the previous methods, the code value represents the

amount to be added to *allmin*, and is again a function of the current level in the pyramid. The four bit code permits reconstruction of many additional elevation values. This will be referred to as method 5 in the empirical comparisons; its implementation is only a slight modification of MAKE_CHILE_NODE$_3$. As shown in Section 5, this is easily the most effective method studied, but it is also the most costly in bits per grid cell.

Our last two variants use a bintree instead of a quadtree pyramid. There are two child nodes for each internal node of the binary temporary pyramid. Each child node is associated with exactly one half of the parent's region, alternating between vertical and horizontal bisectors of the quadrants. For each internal node along a path in the quadtree, there are two equivalent nodes in the bintree; thus an elevation pyramid has twice the depth of the quadtree pyramid. This increase in depth provides additional opportunity to correctly modify the minimum and range. Two variants of the binary pyramid have been considered. The first uses a two bit code at each level in the pyramid; this representation requires 4.0 bits per grid cell due to the increased number of internal nodes. With this method, the codes used are identical to that discussed for method 3; this is referred to as method 6 in the empirical tests. The other variant based upon the binary pyramid uses a one bit code, reducing the amortized requirements to almost exactly 2 bits per grid cell. This method can be advantageous in situations were surface variation is very low, or when only a very rough approximation to the original data is required. The function MAKE_CHILD_CODE$_7$ presents this last encoding method.

Our coding methods store and modify values based on the local minimum and range. It is possible that other values could be used, for example, the average elevation value for the subtree and the range could be stored, with a suitable modification being to add or subtract the current range at each branch in the tree (such as used in [Dutt83]). These values and others considered (maximum, mean, median, etc.), were not found to have the clarity or efficiency of the minimum and range.

5. EXPERIMENTAL RESULTS

We now analyze the seven coding methods in terms of their performance on real elevation data. Six sets of elevation data were selected to test the coding methods. The first is a 1000×1000 pixel grid representing a lakebed bottom. Grid values vary between 1 and 47, indicating the depth below the surface level of the lakebed [Oska90]. The remaining test cases were selected from various areas of Europe. These elevation grids are measured in meters, have an absolute vertical accuracy of ± 30 meters, a relative vertical accuracy of ± 5 meters, and a horizontal resolution of approximately 100 meters. The test data are taken from areas near Munich, West Germany; Warsaw, Poland; Cambridge, England; the Carpathian mountains; and a less mountainous region in the Moldavian S.S.R. These data sets provide a wide range in terms of roughness and difficulty to encode. Figure 6a-f presents 2D grey-scale images of each data set. To visually compare the images and their surface features, they all (excluding Figure 6e) use the same elevation color coding.

To understand a coding method's performance on a particular elevation grid, more insight is required than is provided by visual inspection of the grey-scale images. A quantitative characterization, termed the *range statistics*, identifies the frequency at which an observed range occurs at each level in the temporary pyramid. It also provides general statistics about the range at each pyramid level, specifically the maximum, minimum, average, and standard deviation. Tables 1-6a present the range statistics for the elevation grids. Consider Table 1a, the range statistics for the lakebed bottom. The statistics reveal that the average difference between associated elevation grids cells at level one in the temporary pyramid is small at only 0.23 meters. This average suggests that the lakebed is relatively smooth, in fact only a small number of sibling elevation grid cells have a difference of more than 3. The other test cases are not nearly so smooth. Table 2a indicates the average difference between elevation values of the test case over Munich is 2.96 meters with a standard deviation of 3.84. The average difference is not high, but the standard deviation indicates that higher differences do exist. In fact, the column labeled "≥ 10" indicates that 17,277 sets of

sibling cells differ by 10 or more. With these characteristics, Munich is considered moderately rough. The other cases vary in their roughness and difficulty to encode.

With the observed high differences between the adjacent elevation values, it is expected that the encoding methods will not perfectly encode the grids. However, by using such "rough" elevation data the performances of the encoding methods are considered to approach the worst case results (i.e., we expect that the methods can only do better when higher resolution data is available).

Seven encoding methods were executed on the elevation grids with the results summarized for each test case in Tables 1-6b, respectively. We performed our tests on a Macintosh II equipped with 10 Megabytes of internal memory and a 68020 processor with a clock speed of 16MHz. The operating system was Apple's Unix (A/UX), and the programming was done using the "C" programming language. The summary tables include information to evaluate each method's performance. The statistics include the following: (1) the amortized bits per grid cell (i.e., the number of bits per grid cell required by the associated representation); (2) the number of incorrectly encoded grid cells; (3) the hit ratio (i.e., the percentage of grid cells that were correctly encoded); (4) the maximum miss (i.e., the largest observed difference between an incorrect encoded value and an actual elevation value); (5) the average size of a miss (the sum of the differences between all incorrectly coded values and the actual elevation values divided by the number of misses, i.e., average error *of the misses*), and (6) the overall average error (i.e., the sum of the differences between all incorrectly coded values and the actual elevation values divided by the total number of pixels).

Of the three methods requiring 2.67 bits per grid cell, method 3's performance was significantly the best for all test cases in terms of the number of correctly coded grid cells. For the lakebed bottom, method 3 correctly encoded all but 401 of the elevation values, with the average miss less than 2 meters. These misses were expected from the information found in the range statistics, Table 1a, which indicated some elevation values existed with a difference greater than 3. Where

the differences exceeded 3, the two bit codes could not reliably encode every value. It is interesting to note that many of the high differences between associated grid cells in the lakebed were actual errors in the test data. Thus, we discovered that the elevation pyramid can also be useful to detect errors in data. For the other test cases, the two bit code methods performed quite poorly with the hit ratio being very low. For the Carpathian Mountains, the hit ratio for method 3 was only 17%. Poor levels of performance are related to the average difference between associated grid cells. As the difference increases, the probability of correctly encoding the elevation values decrease substantially. However, notice that for method 3 the average error is by less than 1.5% of the elevation range for all cases except the mountain range, where it was 2.27%. If only an approximation to the orginal surface is required, method 3 may provide an adequate reconstruction.

To cope with the high number of misses method 4, which added bits at the lower levels, was developed. This method performed significantly better in all cases with the hit ratios improving on the average by about 30%. Moldavian S.S.R. was represented much more accurately with the hit ratio improved to 78%. For the lakebed bottom, the results were near perfect with only 4 elevation values being incorrectly encoded (all apparently errors in the original grid). In the other cases, the results were improved with hit ratios of 91% for Munich, and 98% or above for Warsaw and Cambridge. The only poorly represented case was again the Carpathian Mountains with the hit ratio being only 29%. The most effective encoding technique was method 5, which raised the amortized cost to 5.30 bits per grid cell, and performed well on all test cases, except the mountain range. For this test case, only 54% of the elevation values were correctly encoded. For the lakebed, method 5 produced a perfect representation. For the other data, method 5 produced near perfect representations with the hit ratios being above 99%. It is interesting to note that for the more effective reconstructions (methods 4 and 5), the average miss size (when there is a miss) actually increases. This is because the grid cells with low differences between siblings have all been correctly encoded, leaving only the extemely difficult cases for incorrect encoding. Also, the total average

error for methods 4 and 5 is not guarenteed to be lower than for method 3 (although it usually is).

The bintrees did not perform as well as the quadtree representations. Method 6 was consistently out performed by the slightly less expensive method 4. Method 7, which used the one bit codes, performed better than methods 1 and 2. One bit bintree codes could be effective for very smooth terrain, but this technique gives a poor reconstruction in general.

6. PROVIDING A PERFECT REPRESENTATION

In this section, we characterize elevation grids that can be perfectly encoded by the linear coding of methods 3, 4, and 5. Given these characterizations, the minimal cost encoding method can be selected for a particular piece of data without actually trying each coding scheme. The following characterizations have been proven in [Bold90] for slightly modified versions of the coding methods described above. Note the characterizations are actually constraints on values in the temporary pyramid, specifically, the values of the observed range.

For method 3, perfect representation is possible for an elevation grid when the values of the observed range values (denoted as or_i) in the temporary pyramid conform to the following constraints at each level i:

$$or_i \leq \begin{cases} 2, & \text{if i = 1;} \\ 6, & \text{if i = 2;} \\ 2^i, & \text{if } 3 \leq i \leq n. \end{cases}$$

Thus, while two adjacent grid cells that are siblings can differ by at most 2, adjacent cells that are cousins in the pyramid can differ by as much as 6. At higher levels the possible differences increase expontentially.

For method 4, perfect representation is possible when the values of the range in the pyramid conform to the following constraints at each level:

$$or_i \leq \begin{cases} 7i - 1, & \text{if } 1 \leq i \leq 3; \\ 2^i, & \text{if } 4 \leq i \leq n. \end{cases}$$

For method 5 (which requires 5.30 bits per grid cell), perfect representation is possible when the values of the range in the pyramid conform to the following constraints at each level:

$$or_i \leq \begin{cases} 15i - 1, & \text{if } 1 \leq i \leq 3; \\ 12i, & \text{if } 4 \leq i \leq 5; \\ 2^i, & \text{if } 6 \leq i \leq n. \end{cases}$$

Note that the restrictions must be met for all range values at each level i in the temporary pyramid. Given a temporary pyramid, a simple traversal can be performed to examine the values of or_i. If the values of or_i at any node in the pyramid exceed the value specified by the characteristic, then perfect representation can not be assured by that method.

Most elevation grids do contain cases where the range between sibling values is quite large, but these cases are relatively rare. Some encoding methods added additional bits to improve performance. However, when the high differences are infrequent, adding more bits is not cost effective since the additional bits added are not required in most areas. Thus, longer codes would not always be the best solution to providing a perfect representation. One space efficient alternative is to maintain a list of those grid cells which were not correctly encoded by the pyramid structure. With a suitable encoding method, the list should be small compared to the total number of grid cells. For example, the test case Moldavian S.S.R. requires a list of 20,941 values for the missed grid cells from method 5. Since the elevation grid was $2^{10} \times 2^{10}$, this amounts to about 2 percent of the elevation values. Assuming each table entry contains the elevation value (2 bytes) and the (x, y) coordinates (4 bytes), the additional table would require about 125 kilobytes. Although this seems large, the added space raises the amortized cost for perfect representation by less than 1 bit per grid cell. Note that this was an extreme case because most tables are smaller and usually contain much less than 1 percent of the elevation values. The table is not difficult to maintain or traverse since it is static and can be ordered to allow a binary search by coordinate.

Table 7 presents the required number of bits per grid cell to perfectly encode each test case using this method. For the lakebed bottom, method 3 along with a table storing the 401

missed elevation values can be perfectly represented by about 2.7 bits per grid cell. The other test cases, except for the mountain range, were perfectly encoded with only minor increases in their amortized storage cost. The worst test case was the Carpathian Mountains, which would require 27.56 bits per grid cell for perfect encoding. Obviously, for this test case the original elevation grid is the most effective method for assuring perfect representation. For the other cases, the worst required 6.26 bits per grid cells for perfect encoding. In some cases, method 4 required less storage than method 5. However, method 4 also requires larger tables, suggesting that the search time may be intolerable. In addition, the 3-bit codes are not so easily manipulated. For these reasons, method 5 combined with the table appears to be the most effective method for providing perfect representation.

The only shortcoming to using the table is the added expense of searching for elevation values. When the list is small, search time is not a significant factor, but the added search time could become noticeable if a large table is accessed frequently. To minimize accesses to the table, it is important to search the table only when it is highly probable that a miss has occurred. For methods 3, 4, and 5, this is easily determined at level one of the elevation pyramid. Notice at level one, the amounts by which *allmin* may be refined ranges from 0 through 7 for method 4, and 0 through 15 for method 5. If the code value is less than the largest possible value, then the table does not need to be searched. However, when the code value equals 7 or 15, for methods 4 and 5 respectively, it is possible that the elevation value may not be correctly encoded. At this point, if perfect representation is required, the table is searched. For method 3, the table is searched when the value of the code at level one equals 3, or when the value of *allrng* is greater than 8 at level one. Of course it is possible that the table will be accessed when a value is correctly coded, but for the test cases studied the decision to search the table is correct at least 95% percent of the time.

7. ARCHIVAL STORAGE, RETRIEVAL, AND TRANSMISSION

Our primary motivation for the elevation pyramid technique is not 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 such applications, the user would like to interactively retrieve portions of the database for processing. Such retrieval must be done quickly, so long term archival methods are not desirable if they slow the retrieval process. Not only does the elevation pyramid approach to compression allow for direct manipulation of the encoded data (as opposed to decompressing the data and then processing), but we expect that the elevation pyramid can actually reduce processing time since the amount of disk activity is reduced.

The region to be represented may not be a simple square region of land, but may include significant portions of ocean or lake without associated elevation data. 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 [Mark85]. This upper-level quadtree would be used primarily to distinguish water 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 quadtree node for any flat area, not just the ocean.

The upper-level quadtree could be further extended to divide the land area into regions smooth enough to apply an elevation pyramid. This technique is effective when isolated areas of the grid exhibit high variation. When this is the case, the grid may be divided into 4 quadrants. Each of these quadrants may then be represented by different elevation pyramid variants. If necessary, quadrants may be further divided with the intent that some of the new regions may be represented by a pyramid. When a region exhibits high range values across the entire area, further division is useless. For such (relatively small) regions, no representation other than the actual elevation grid will be accurate. Such a dynamic approach would allow for the efficient representation of elevation

for large portions of the earth's surface. It would even be possible to fit the entire earth's land surface elevation data onto a single CD-ROM at 1:1,000,000 scale.

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 × 8K, 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 previously 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.

8. CONCLUSIONS

The elevation pyramid can provide enormous space savings for topographic data as compared to the standard grid method when the topographic data has high resolution. When the data corresponds to a relatively smooth surface, the basic representation requiring 2.67 bits/grid cell is quite effective, and provides a 5:1 compression over a 16 bit/grid cell representation. However, empirical tests indicate that many data sets are not smooth. For these cases, the basic representation only provides an approximation to the surface, which may be acceptable for some applications. When additional performance is required, several variants are available. These variants are much

more robust, but they require additional storage space. The most effective method developed required 5.3 bits/grid cell. However, even this method could not perfectly encode all of the test data. When perfect representation is required, a good approach is to maintain a list of the incorrectly encoded values. As seen from the test cases, the list is usually small. The elevation pyramid codes indicate when the list must be searched.

Additional work remains to be done. One task is to implement the upper-level quadtree structure for large land areas, such as a continent or the entire earth's surface. Another is to continue the refinement of the encoding methods. We expect that other variants of the elevation pyramid, not yet discovered, are possible which either reduce storage requirements or increase accuracy. However, it is expected that no large increases in performance will result from new variants developed using the concept of the elevation pyramid, i.e., they will be constrained by the same type of problems already encountered in this research. Another area of work is to test the utility of the elevation pyramid for grayscale images. Images have correlated data, though not so well correlated as smooth surfaces. In particular, edges of objects will be miscoded. The result may still be adequate for some purposes. Alternatively, it is possible that augmented with an edge representation, the elvation pyramid will prove highly faithful in reconstruction. The next extension would then be to color images. Color images can be coded as three intensity bands (red, green, and blue). However, better compression would result if the three bands can be coded as a single value such that close shades in (r, g, b) space would be close in the single dimensional space. Finally, use of the elevation pyramid in encoding video data should be examined.

9. REFERENCES

1. [Aho83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison Wesley, Reading MA, 1983.

2. [Burr86] P.A. Burrough, *Principles of Geographical Information Systems for Land Resources Assessment*, Oxford Science Publications, New York, 1986.

3. [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.

4. [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.

5. [Dutt84] G. Dutton, Geodesic modelling of planetary relief, *Cartographica 21*, 1984, 188-207.

6. [Dürs88] M.J. Dürst and T.L. Kunii, Error-free image compression with gray scale quadtrees, *Proceedings of the International Workshop on Discrete Algorithms and Complexity*, Hiroshi Imai, Ed., Fukuoka, Japan, Nov 1989, 115-121.

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

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. [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.

11. [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.

12. [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.

13. [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.

14. [Oska90] D.N. Oskard, T.H. Hong, and C.A. Shaffer, Real-time algorithms and data structures for underwater mapping, to appear in *IEEE Transactions on Systems, Man, and Cybernetics.*

15. [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

16. [Same89] H. Samet, *Applications of Spatial Data Structures; Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading MA, 1989.

17. [Same90] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading MA, 1990.

18. [Shaf90a] C.A. Shaffer, A full resolution elevation representation requiring three bits per pixel, in *Design and Implementation of Large Spatial Databases*, A. Buchmann, O. Gunther, T.R. Smith and Y.-F. Wang, Eds., Springer Verlag, Berlin, 1990, 45-64,

19. [Shaf90b] C.A. Shaffer and H. Samet, Set operations for unaligned linear quadtrees, *Computer Vision, Graphics, and Image Processing 50*, (April, 1990), 29-49.

20. [Shaf90c] C.A. Shaffer, H. Samet, and R.C. Nelson, **QUILT**: a geographic information system based on quadtrees, to appear in *International Journal of Geographical Information Systems*, 1990.

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

22. [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_PT(root : ↑PYR_NODE; n, M, R, x, y : integer) : integer;
  { Generate the elevation value at point (x, y) in the elevation grid of size 2^n × 2^n from the
    elevation pyramid. The pointer root points to the root of the pyramid, while M and R indicate
    the minimum and total range for the entire grid. Although root is shown as a pointer to
    pyramid nodes, in practice it is likely to be an index into an array implementing the pyramid
    as a heap.}
var half : integer;
begin
   repeat
      n := n − 1; half := 2^n;
      if (x < half) then
         if (y < half) then child := NW;
         else begin child := NE; x := x − half end
      else if (y < half) then begin child := SW; y := y − half end
      else begin child := SE; x := x − half; y := y − half end;
      case root.code[child] of { Modify M and R based on code value}
         '00' : R := R/2;
         '01' : ;{ No modification to values}
         '10' : R := R/2; M := M + R;
         '11' : M := M + R
      end; {Case }
      root := CHILDOF(root, child) { Traverse down pyramid}
   until n = 0;
   return (M) { Return the elevation value}
end;
```

Algorithm 1. An algorithm to decode one elevation point from the pyramid.

```
procedure PASS1(root : ↑NODE; n, x, y : integer);
  { Construct a temporary pyramid, where root points to the root, n is the current level, and (x, y)
    is the coordinate of the elevation grid associated with the current leaf node when at level 0}
var cx, cy, half, max, currquad : integer;
begin
   if (n > 0) then {if not at bottom level}
      begin
         n := n − 1; half := 2^n; cx := x + half; cy := y + half;
         PASS1(root.child[NW], n, x, y); { Process NW quadrant}
         PASS1(root.child[NE], n, cx, y); { Process NE quadrant}
         PASS1(root.child[SW], n, x, cy); { Process SW quadrants}
         PASS1(root.child[SE], n, cx, cy); { Process SE quadrant}
         { Set minimum and range of internal node pointed to by root}
         root.min := MIN(root.child[NW].min, root.child[NE].min,
                         root.child[SW].min, root.child[SE].min);
         max := 0;
         for currquad in (NW, NE, SW, SE) do { Visit each child }
            max := MAX(max, root.child[currquad].min + root.child[currquad].range);
         root.range := max − root.min
      end
   else { At bottom level}
      { Set minimum value of the leaf node pointed to by root}
      root.min := grid(x, y) { Copy elevation value into pyramid}
end; { PASS1}

procedure PASS2(root : ↑NODE; n : integer);
  { Construct an elevation pyramid in depth-first order using MAKE_CHILD_CODE_v depending
    upon on desired variant.}
var
   currquad : integer;
   code : CODESET; {Contains the codes for one node of pyramid}
begin
   {Generate code for each child}
   for currquad in (NW, NE, SW, SE) do {Visit each quadrant to generate code}
      code[currquad] := MAKE_CHILD_CODE_v(root.min, root.rng, n,
                        root.child[currquad].min, root.child[currquad].rng);
   WRITE_DISK(code); {Write the four codes to disk}
   if (n > 1) then { If additional levels to process then continue}
      for currquad in (NW, NE, SW, SE) do PASS2(root.child[currquad], n − 1)
end; {PASS2}
```

Algorithm 2. A depth-first traversal over the temporary pyramid is performed by PASS1 to set the minimum and range at each node. Then, a traversal of the temporary pyramid is performed by PASS2 to construct and output the elevation pyramid.

```
function MAKE_CHILD_CODE1(allmin, allrng, level : integer;
                          var childmin, childrng : integer) : CODE;
  { The basic coding method compares the current values of allmin and allrng with childmin
    and childrng to generate the proper code. As a side effect, the child's minimum and range are
    updated for future use.}
var code : CODE;
begin
   if (allmin + allrng ≤ childmin) then
      begin code := '11'; childmin := allmin + allrng; childrng := allrng end
   else if (allmin + allrng/2 ≤ childmin) then
      begin code := '10'; childmin := allmin + allrng/2; childrng := allrng/2 end
   else if (allmin + allrng ≤ childmin + childrng) then
      begin code := '01'; childmin := allmin; childrng := allrng end
   else begin code := '00'; childmin := allmin; childrng := allrng/2 end;
   return (code)
end; { MAKE_CHILD_CODE1}
```

```
function MAKE_CHILD_CODE2(allmin, allrng, level : integer;
                          var childmin, childrng : integer) : CODE;
  { Improved coding is achieved by using different coding rules at levels one and two.}
var code : CODE;
begin
   if (level ≠ 1) then { if not at the bottom of elevation pyramid}
      if (allmin + allrng ≤ childmin) then
         begin code := '11'; childmin := allmin + allrng;
            if level ≠ 1 then childrng := allrng
            else childrng := allrng/2
         end
      else if (allmin + allrng/2 ≤ childmin) then
         begin code := '10'; childmin := allmin + allrng/2; childvar := allrng/2 end
      else if (allmin + allrng ≤ childmin + childrng) then
         begin code := '01'; childmin := allmin; childrng := allrng end
      else begin code := '00'; childmin := allmin; childrng := allrng/2 end
  else {at bottom level}
      if (allmin + 3 * allrng/2 ≤ childmin) then code := '11'
      else if (allmin + allrng ≤ childmin) then code := '10'
      else if (allmin + allrng/2 ≤ childmin) then code := '01'
      else code := '00';
  return (code)
end{MAKE_CHILD_CODE2}
```

```
function MAKE_CHILD_CODE3(allmin, allrng, level : integer;
                          var childmin, childrng : integer) : CODE;
  { At levels 1 and 2 allmin is refined as a function of level when allrng ≤ 8.}
var code : CODE;
begin
   if (level > 2) or (allrng > 8)
      return (MAKE_CHILD_CODE2(allmin, allrng, level, childmin, childrng));
   else { At bottom two levels of elevation pyramid with allrng ≤ 8}
      begin
         code = '00';
         while (allmin + 2^(level-1)* VAL(code) ≤ childmin) and (VAL(code)< 4) do
            { Increment code value}
            code := code + 1;
         childmin := childmin + 2^(level-1)* VAL(code - 1)
      end; { else }
   return (code - 1)
end; {MAKECHILDCODE3}
```

```
function MAKE_CHILD_CODE7(allmin, allrng, level : integer;
                          var childmin, childrng : integer) : CODE;
  {An encoding method for a binary pyramid. The 1-bit code requires an amortized cost of only
   2.0 bits/grid cell. The method is most appropriate for very smooth terrain.}
var code : CODE;
begin
   if (level > 3) { Not at bottom three levels}
      begin
         if (allmin + allrng ≤ childmin) then
            begin code := '1'; childmin := allmin + allrng; childrng := allrng; end
         else begin
               code := '0'; childmin := allmin;
               if (allrng = 1) then childrng := 1
               else childrng := allrng/2
            end;
         return (code)
      end
   else { At bottom three levels}
      begin
         code := '0';
         while ((allmin + 2^(level-1)* VAL(code) ≤ childmin) and (VAL(code) < 2)) do
            code = code + 1;{ Increment code value}
         childmin := childmin + 2^(level-1)* VAL(code - 1);
         return (code - 1)
      end
end; {MAKE_CHILD_CODE7}
```

Algorithm 3. Four variants for MAKE_CHILD_CODE.

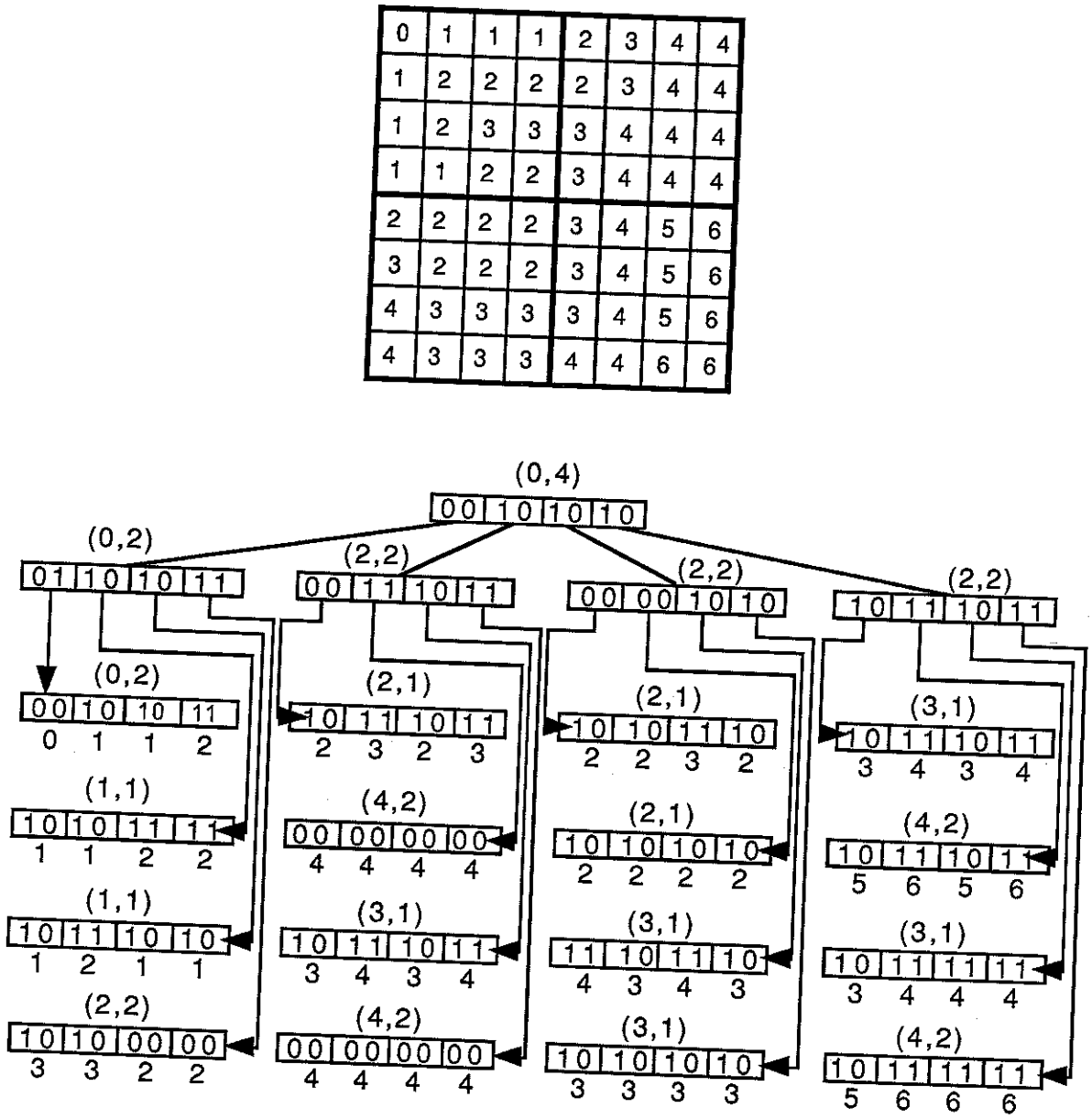


Figure 1. A sample elevation grid and its corresponding elevation pyramid using the basic coding method. Each 2-bit field indicates the required modification to the minimum and range from the current node to generate the corresponding values for the associated quadrant.

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

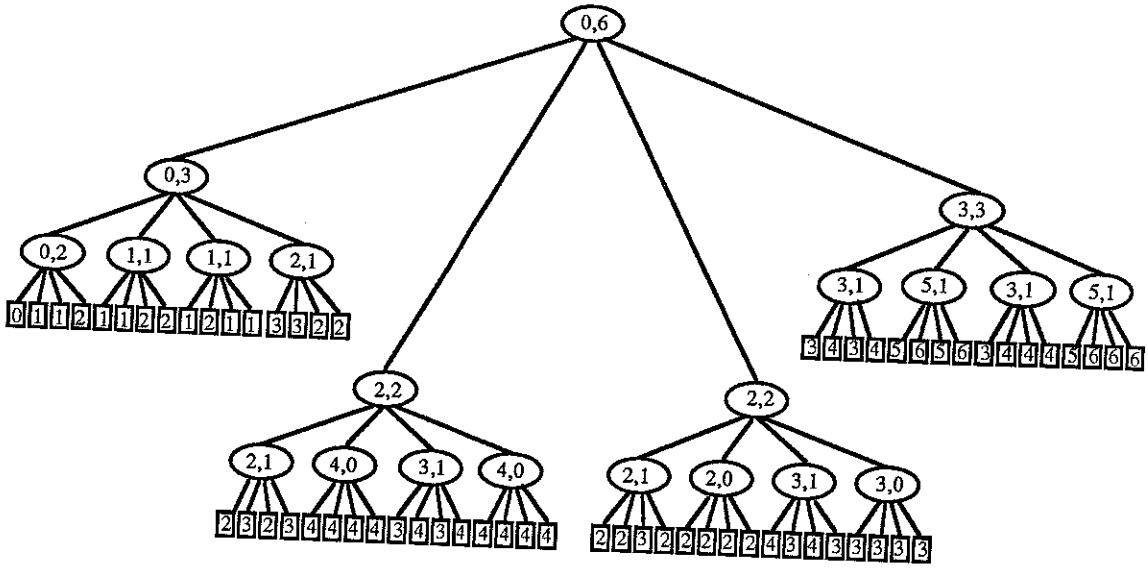


Figure 2. The elevation grid of Figure 1, and its temporary pyramid storing at each internal node the minimum and range for the corresponding area of the grid. The leaf nodes store only the elevation values.

0	2	4	5	8	10	12	14
2	3	4	7	9	11	13	14
3	4	4	6	9	10	13	14
4	5	5	6	9	12	13	15
5	6	6	7	9	12	13	14
6	7	7	8	10	11	12	13
7	8	8	9	10	11	11	12
8	8	9	9	9	10	10	11

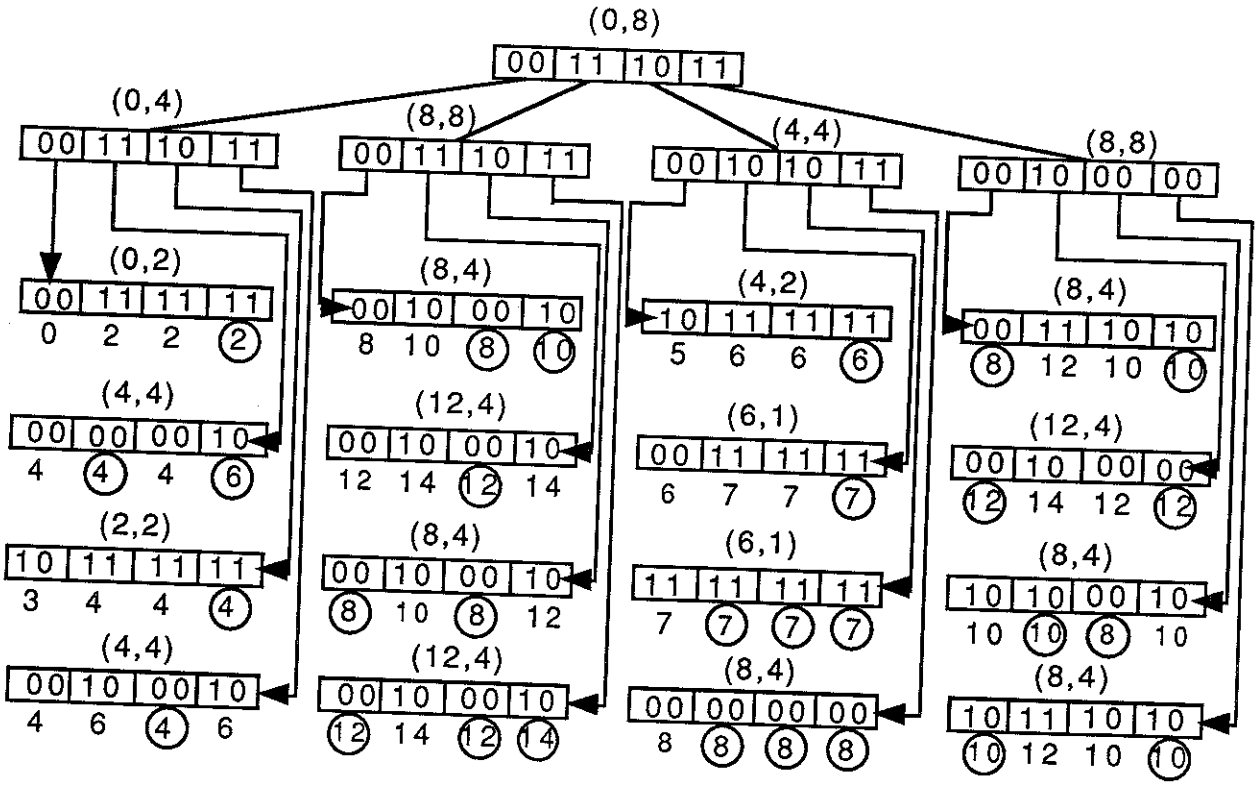


Figure 3. An elevation grid and its corresponding elevation pyramid using the basic coding method. The elevation values rise too rapidly in the lower right quadrant for the pyramid to accurately encode all grid cells. The incorrectly coded cells are cirled.

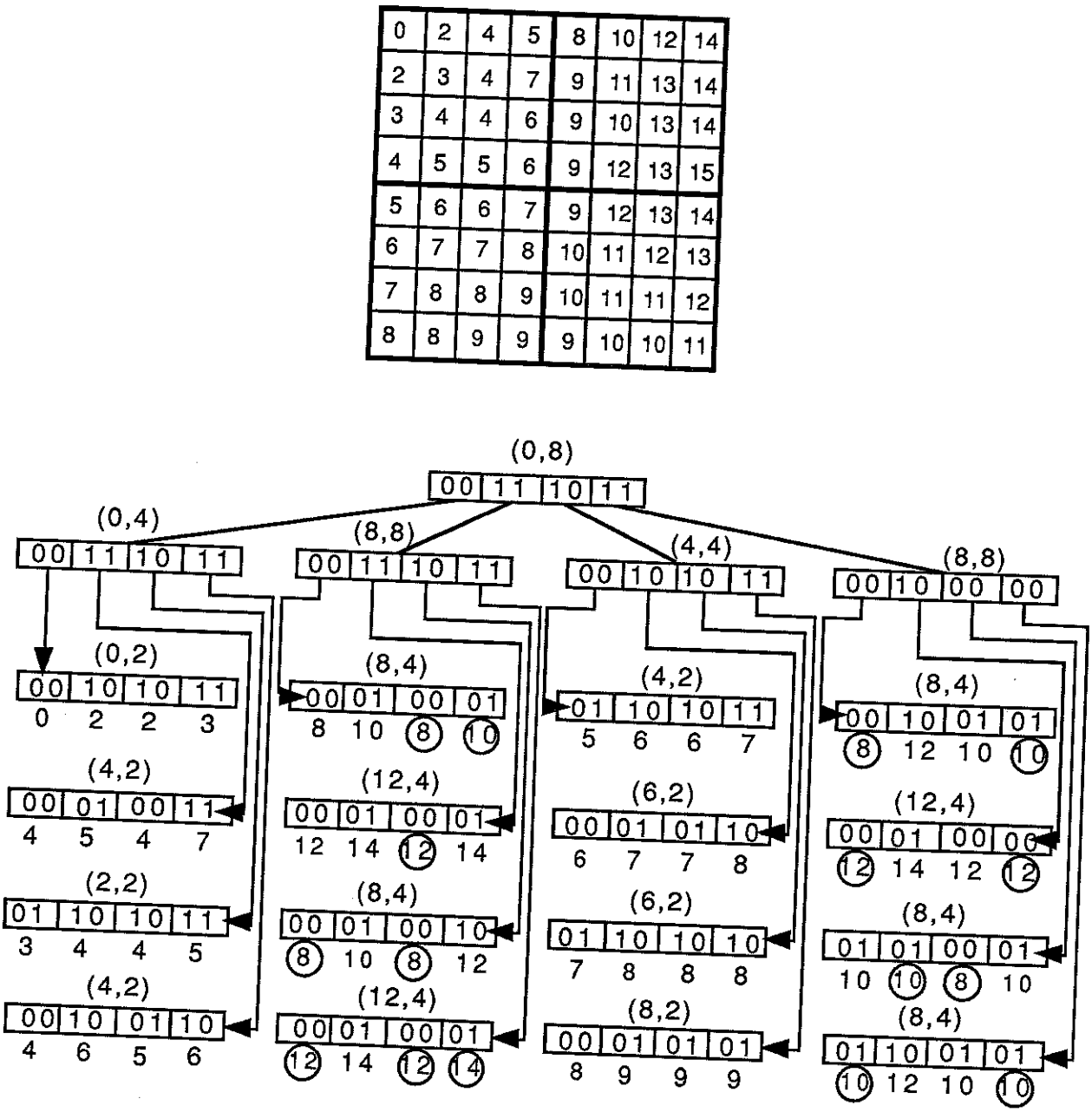


Figure 4. The elevation grid of Figure 3 and its elevation pyramid derived using the level-modified coding rules described in MAKE_CHILD_CODE2. The values of *allrng* at level one prevent 16 elevation values from being correctly encoded.

0	2	4	5	8	10	12	14
2	3	4	7	9	11	13	14
3	4	4	6	9	10	13	14
4	5	5	6	9	12	13	15
5	6	6	7	9	12	13	14
6	7	7	8	10	11	12	13
7	8	8	9	10	11	11	12
8	8	9	9	9	10	10	11

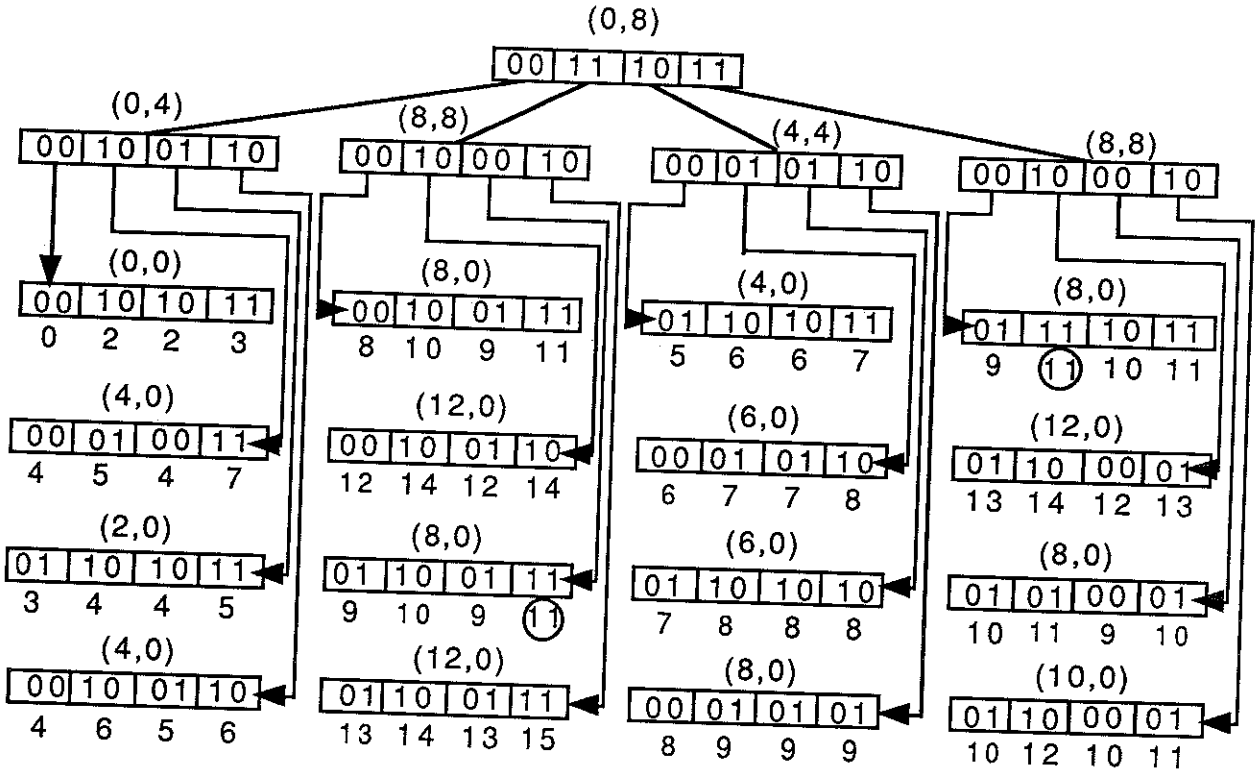
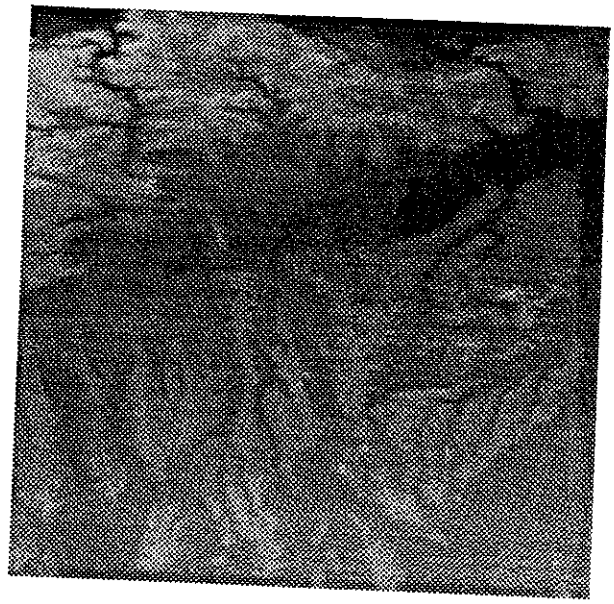
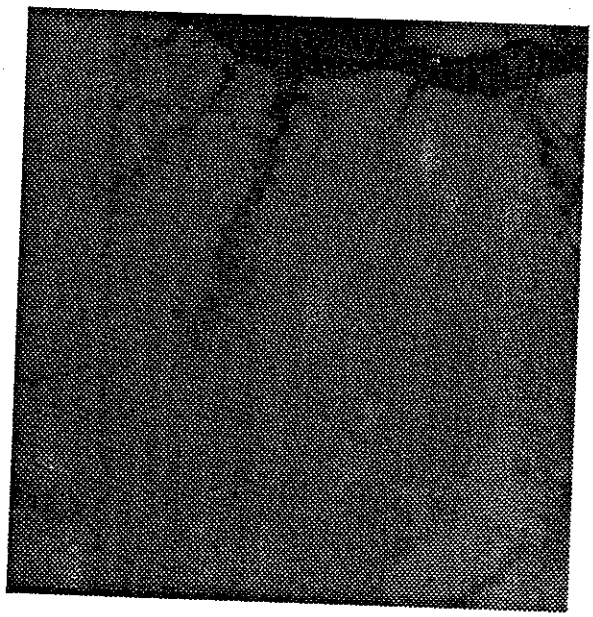


Figure 5. The elevation grid of Figure 3 and its elevation pyramid derived using the linear coding rules described in MAKE_CHILD_CODE3. With these modifications, only 2 values are incorrectly coded.

(a)

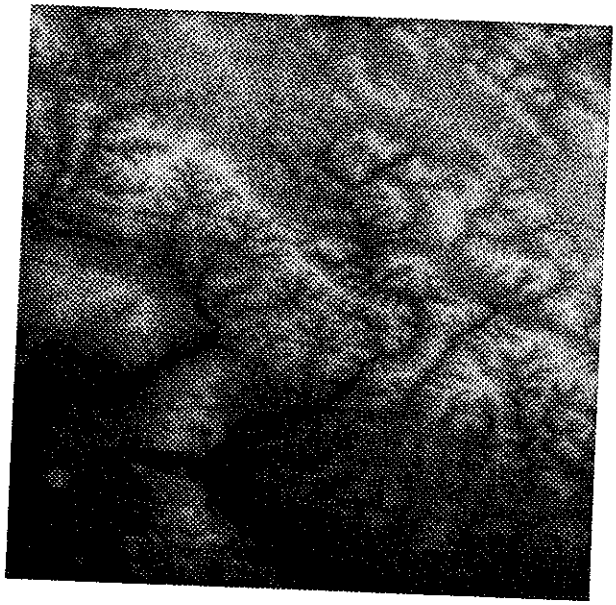


(b)

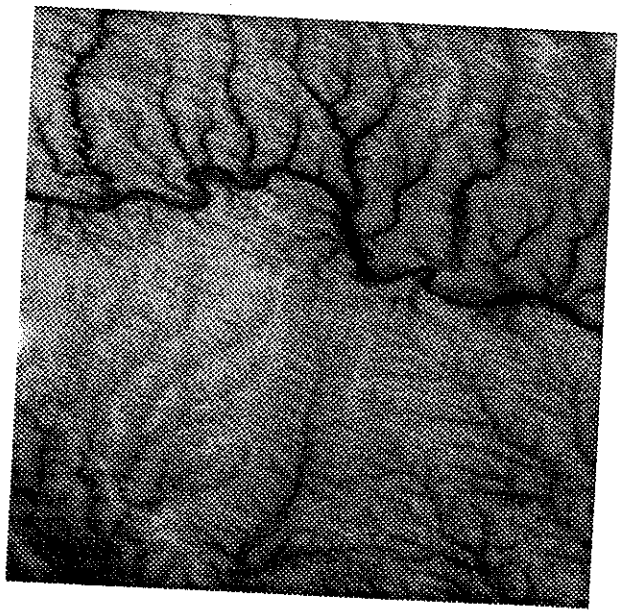


(c)

(d)



(e)



(f)

Figure 6. Five test elevation grids. (a) Lakebed. (b) Munich. (c) Warsaw. (d) Cambridge. (e) Carpathian Mountains. (f) Moldavian S.S.R.

Table 1a. Range statistics for the 1000 x 1000 elevation grid of the lakebed bottom. The grid is relatively smooth with elevation values between 1 and 46.

Level	Observed Range														
	0	1	2	3	4	5	6	7	8	9	≥10	Min	Max	Avg.	Std. Dev.
1	191,934	56,266	1,480	213	71	25	8	1	2	0	0	0	8	0.23	0.45
2	28,403	27,146	5,226	1,201	331	115	30	17	19	6	13	0	13	0.67	0.81
3	3,051	6,495	3,320	1,450	706	321	150	71	29	11	21	0	15	1.46	1.40
4	240	906	918	737	466	211	174	117	79	44	77	0	16	2.85	2.32
5	17	71	134	149	158	121	89	71	52	41	121	0	20	5.28	3.61
6	1	3	5	6	18	23	27	27	25	14	107	0	28	9.62	5.21
7	0	0	0	0	1	0	4	4	1	1	55	4	34	16.91	6.98
8	0	0	0	0	0	0	0	0	0	0	16	12	46	27.88	10.35
9	0	0	0	0	0	0	0	0	0	0	4	27	46	36.75	9.64
10	0	0	0	0	0	0	0	0	0	0	1	46	46	46.00	0.00

Table 1b. The results of executing the seven encoding methods on the lakebed.

Encoding Methods	Amortized Bits / Grid Cell	Incorrectly Encoded Grid Cells	Encoding Hit Ratio	Maximum Miss	Average Miss	Total Average Error
One	2.67	99,358	90.06%	9	1.06	0.11
Two	2.67	13,662	98.63%	10	1.60	0.02
Three	2.67	401	99.96%	6	1.67	0.00
Four	3.90	4	99.99%	2	1.50	0.00
Five	5.30	0	100.00%	0	0.00	0.00
Six	4.00	70	99.99%	4	1.54	0.00
Seven	2.00	166,317	83.37%	26	4.91	0.82

Table 2a. Range statisics for the 1024 x 1024 elevation grid located over Munich, West Germany. The SW corner of the grid is located at 10° east longitude, and 48° north latitude with elevation values between 390 and 778 meters. These statistics suggest the surface is fairly rough with at least 17,277 sets of sibling elevation values having a difference of 10 or more.

Level	Observed Range														Std.
	0	1	2	3	4	5	6	7	8	9	≥10	Min	Max	Avg.	Dev.
1	116,276	2,675	1,939	57,885	26,915	1,659	16,409	17,720	351	3,068	17,277	0	72	2.96	3.84
2	13,432	358	587	9,618	4,647	683	4,270	5,030	275	1,847	14,789	0	115	8.61	9.22
3	1,071	71	170	1,487	735	202	659	817	101	328	10,743	0	174	18.61	16.71
4	45	16	22	130	72	35	202	161	27	62	3,395	0	227	34.18	25.81
5	0	1	0	2	5	4	35	11	2	13	1,014	1	243	56.05	34.12
6	0	0	0	0	0	0	4	0	0	1	255	9	256	86.41	39.81
7	0	0	0	0	0	0	0	0	0	0	64	56	338	125.72	47.72
8	0	0	0	0	0	0	0	0	0	0	16	114	351	182.38	63.38
9	0	0	0	0	0	0	0	0	0	0	4	220	355	226.00	60.84
10	0	0	0	0	0	0	0	0	0	0	1	338	388	388.00	0.00

Table 2a. The results of executing the seven encoding methods on Munich, West Germany.

Encoding Methods	Amoritized Bits / Grid Cell	Incorrectly Encoded Grid Cells	Encoding Hit Ratio	Maximum Miss	Average Miss	Total Average Error
One	2.67	661,751	36.89%	70	2.82	1.78
Two	2.67	608,334	41.98%	72	2.26	1.31
Three	2.67	486,771	53.57%	77	4.52	2.10
Four	3.90	90,747	91.34%	234	13.81	1.20
Five	5.30	9,973	99.26%	178	17.25	0.16
Six	4.00	150,211	85.67%	195	5.19	0.74
Seven	2.00	552,054	47.35%	228	22.05	11.61

Table 3a. Range statisitics for the 512 x 512 elevation grid located near Warsaw, Poland. The SW corner of the grid is located at 21° east longitude, and 51° north latitude with elevation values between 113 and 344 meters. Only a small percentage of the elevation values differ by 10 or more.

Level	Observed Range														
	0	1	2	3	4	5	6	7	8	9	≥10	Min	Max	Avg.	Std. Dev.
1	6,628	25,255	17,363	7,658	3,614	1,848	1,078	673	394	276	749	0	40	2.04	2.00
2	44	739	2,276	2,993	2,551	2,056	1,389	979	689	522	2,136	0	64	5.69	4.89
3	0	1	24	120	272	366	423	382	326	277	1,905	1	77	11.63	8.80
4	0	0	0	0	2	7	11	31	29	59	885	4	112	20.54	13.45
5	0	0	0	0	0	0	0	0	0	1	255	9	132	33.96	19.63
6	0	0	0	0	0	0	0	0	0	0	64	19	144	52.56	24.92
7	0	0	0	0	0	0	0	0	0	0	16	48	168	76.75	30.07
8	0	0	0	0	0	0	0	0	0	0	4	79	178	114.75	44.51
9	0	0	0	0	0	0	0	0	0	0	1	231	231	231.00	0.00

Table 3b. The results of executing the seven encoding methods on Warsaw.

Encoding Methods	Amortized Bits / Grid Cell	Incorrectly Encoded Grid Cells	Encoding Hit Ratio	Maximum Miss	Average Miss	Total Average Error
One	2.67	120,503	54.03%	55	1.82	0.84
Two	2.67	101,283	61.36%	59	1.75	0.68
Three	2.67	53,346	79.27%	68	3.56	0.72
Four	3.90	4,869	98.14%	81	7.97	0.15
Five	5.30	272	99.90%	26	5.06	0.01
Six	4.00	8,463	96.77%	53	3.36	0.11
Seven	2.00	114,365	56.37%	98	9.42	4.11

Table 4a. Range statistics for the 512 x 512 elevation grid located over Cambridge, England. The SW corner is located at 1° west longitude, and 52° north latitude with elevation values between 1 to 182 meters. The terrain is considered as moderately rough.

Level	Observed Range													Std.
	0	1	2	3	4	5	6	7	8	9	≥10	Min	Max	Avg.	Dev.
1	16,727	19,763	11,938	6,456	3,686	2,270	1,486	893	645	446	1,226	0	30	1.97	2.40
2	1,376	1,842	2,261	2,028	1,608	1,258	1,040	860	665	546	2,900	0	50	5.52	5.22
3	85	137	202	224	236	248	229	234	201	202	2,098	0	84	11.87	8.68
4	4	7	5	8	18	18	18	24	20	22	880	0	96	22.40	11.79
5	0	0	0	0	0	0	1	1	1	0	253	6	96	38.30	14.18
6	0	0	0	0	0	0	0	0	0	0	64	29	122	58.86	16.57
7	0	0	0	0	0	0	0	0	0	0	16	49	122	82.44	22.22
8	0	0	0	0	0	0	0	0	0	0	4	91	159	126.00	27.98
9	0	0	0	0	0	0	0	0	0	0	1	181	181	181.00	0.00

Table 4b. The results of executing the seven encoding methods on Cambridge.

Encoding Methods	Amortized Bits / Grid Cell	Incorrectly Encoded Grid Cells	Encoding Hit Ratio	Maximum Miss	Average Miss	Total Average Error
One	2.67	133,925	48.91%	25	1.91	0.98
Two	2.67	119,885	54.27%	26	1.79	0.82
Three	2.67	65,687	74.94%	31	3.88	0.97
Four	3.90	5,421	97.93%	49	4.06	0.08
Five	5.30	303	99.87%	15	3.81	0.00
Six	4.00	10,896	95.84%	38	3.05	0.13
Seven	2.00	135,887	48.16%	76	9.90	5.13

Table 5a. Range statistics on the 1024 x 1024 elevation grid located in the Carpathian Mountains of the Soviet Union. The SW corner is located at 23° east longitude and 48° north latitude with elevation values between 107 and 1693. Located in a mountain range, the area is extremely rough with 183,494 sets of sibling elevation values differing by 10 or more.

Level	Observed Range														Std.
	0	1	2	3	4	5	6	7	8	9	≥10	Min	Max	Avg.	Dev.
1	1,152	7,331	10,769	10,300	9,116	8,279	7,934	7,740	8,081	7,948	183,494	0	146	19	14.35
2	0	25	245	608	946	1,037	1,023	918	855	799	59,080	1	281	55	38.72
3	0	0	0	2	9	20	55	107	108	144	15,939	3	441	117	75.91
4	0	0	0	0	0	0	0	0	0	3	40,93	9	691	209	125.73
5	0	0	0	0	0	0	0	0	0	0	1,024	14	929	334	180.98
6	0	0	0	0	0	0	0	0	0	0	256	26	1,059	491	229.94
7	0	0	0	0	0	0	0	0	0	0	64	168	1,244	691	265.49
8	0	0	0	0	0	0	0	0	0	0	16	381	1,434	947	309.22
9	0	0	0	0	0	0	0	0	0	0	4	897	1,449	1,287	262.37
10	0	0	0	0	0	0	0	0	0	0	1	1,586	1,586	1,586	0.0

Table 5b. The results of executing the encoding methods on the Carpathian Mountains.

Encoding Methods	Amortized Bits / Grid Cell	Incorrectly Encoded Grid Cells	Encoding Hit Ratio	Maximum Miss	Average Miss	Total Average Error
One	2.67	877,011	16.36%	606	37.32	31.21
Two	2.67	885,770	15.53%	622	40.48	34.19
Three	2.67	867,383	17.28%	622	43.51	35.99
Four	3.90	739,888	29.44%	808	98.08	69.21
Five	5.30	486,354	53.62%	752	91.47	42.43
Six	4.00	839,739	19.92%	974	143.25	114.42
Seven	2.00	953,280	09.09%	862	98.26	89.33

Table6a. Range statistics for the 1024 x 1024 elevation g rid located over Moldavian S.S.R. The SW corner is located at 27° east longitude and 48° north latitude with elevation values between 61 and 355 meters. Located near the Carpathian mountains, the area is very rough. Specifically, 35,766 sets of sibling cells differ by 10 or more.

Level	Observed Range														Std.
	0	1	2	3	4	5	6	7	8	9	≥10	Min	Max	Avg.	Dev.
1	3,415	15,846	29,636	35,857	36,008	32,003	26,549	20,575	15,285	11,254	35,766	0	65	5.76	4.26
2	124	38	288	836	1,369	1,982	2,646	3,069	3,345	3,560	48,279	0	135	16.21	10.57
3	6	2	0	3	4	25	33	39	65	93	16,114	0	186	33.77	18.90
4	0	0	0	0	0	0	0	0	0	0	4,096	12	190	58.53	26.89
5	0	0	0	0	0	0	0	0	0	0	1024	34	246	88.84	34.04
6	0	0	0	0	0	0	0	0	0	0	246	56	248	122.02	38.02
7	0	0	0	0	0	0	0	0	0	0	64	97	250	157.91	38.29
8	0	0	0	0	0	0	0	0	0	0	16	130	254	198.00	36.44
9	0	0	0	0	0	0	0	0	0	0	4	243	292	261.00	22.27
10	0	0	0	0	0	0	0	0	0	0	1	294	294	294.00	0.00

Table 6b. The results of executing the seven encoding methods on Moldavian S.S.R.

Encoding Methods	Amortized Bits / Grid Cell	Incorrectly Encoded Grid Cells	Encoding Hit Ratio	Maximum Miss	Average Miss	Total Average Error
One	2.67	860,155	17.97%	72	4.14	3.40
Two	2.67	816,575	22.13%	73	3.83	2.98
Three	2.67	795,230	24.16%	82	5.70	4.32
Four	3.90	225,514	78.49%	228	14.13	3.04
Five	5.30	20,941	98.01%	172	13.35	0.27
Six	4.00	283,866	72.93%	189	6.89	1.87
Seven	2.00	917,881	12.46%	229	40.17	35.16

Table 7. The number of bits per grid cell required to perfectly represent each of the test cases. The cost includes the elevation pyramid plus the additional table to maintain any incorrectly coded elevation values.

Encoding Methods	Lakebed bottom	Munich, W. Germany	Warsaw, Poland	Carpathian Mountains	Moldavian S.S.R.	Cambridge, England
One	7.44	32.96	24.73	42.92	42.04	27.20
Two	3.33	30.52	32.12	43.22	37.38	24.62
Three	**2.69**	24.95	12.43	43.51	39.07	14.70
Four	3.90	8.05	**4.79**	37.79	14.22	**4.89**
Five	5.30	**5.76**	5.35	27.56	**6.26**	5.35
Six	4.01	10.88	5.55	42.44	16.99	6.00
Seven	9.98	27.27	22.94	45.68	44.02	26.88