In David Maguire, David Rhind and Mike Goodchild, editors, "GIS: Principles and Practice", Longman Higher Education and Reference, London UK, 1991, Vol. 1, pp. 215-225.

15

COMPUTER SYSTEMS AND LOW-LEVEL DATA STRUCTURES FOR GIS

WM. R FRANKLIN

This chapter is an introduction to several aspects of computer science that are applicable to Geographical Information Systems. First, some of the choices in computer systems and the likely coming advances in hardware and software are discussed. Then some of the general properties of low-level data structures, with principles like abstract data structures versus their implementations, and examples like hash tables, extendible arrays, uniform grids and Voronoi diagrams are considered. Next there is an introduction to algorithms analysis, its limitations when applied to practical problems, new trends such as randomized algorithms and implementation considerations. This is followed by an introduction to software engineering, including the traditional waterfall model, the newer rapid prototyping technique and the importance of profiling the program to improve it.

COMPUTER SYSTEMS

Classification of systems

Computer systems may be partitioned by their memory size and I/O (input/output) speed into personal computers, workstations, minicomputers, mainframes and supercomputers. In general, within a class the larger machines may be more cost efficient, but between classes of machines, the smaller classes are more cost efficient. That is, it is most economical to run a computation on the smallest class of machine that will support it. A workstation in the early 1990s typically has the following features:

- 16 megabytes (i.e. about 16 million bytes) of real memory;
- three times as much virtual memory;
- a 500 megabyte hard disk;
- a bit mapped display with 1 million pixels;
- a 10 MIPS (10 million instructions executed per second) processor. However, note that MIPS is only a rough measure of speed, and differences of under 50 per cent can probably be ignored. Floating point performance is also important;
- a 32 bit wide I/O bus to carry data between the processor, memory, and the input/output devices;
- a timesharing operating system, such as Unix (AT&T Bell Laboratories 1978, 1984; Quarterman, Silberschatz and Peterson 1985);
- high level languages, such as C, Modula, or Fortran-77.

Optional, but increasingly common features include the following:

- colour and grey-scale displays;
- CD-ROMs (compact disk-read only memory) and optical disks (laser disks) to store large, fixed databases up to about 1 gigabyte (10^9 bytes);

73-1

- database tools, or 'fourth generation languages', such as Oracle, object-oriented languages such as C++, artificial intelligence aids such as expert systems, and numerical libraries such as IMSL or NAG (Numerical Algorithm Group).

A personal computer, in contrast, would have each measure of capacity, such as memory and disk space, several times smaller. Each of these capacities and speeds is increasing at 20–100 per cent per year. The contrast in cost between disk and main memory is notable since while disk is 10 000 times slower to access, it is only 10 times cheaper.

All this means that a 100 megabyte database can now be processed easily on a workstation. For larger databases, the workstation can be supplemented by access to a supercomputer. Extremely large databases, such as those obtained from earth observation satellites and now stored on magnetic tape, can be over 50 Tb (terabytes, 10^{12} bytes) and are still growing. Mel *et al.* (1988) give a view of the personal computer in the year 2000.

For most GIS researchers, the workstation is probably the relevant machine. The personal computer currently has too little memory for processing large databases and, unlike workstations, often does not have virtual memory or a timesharing environment. In addition, after peripherals are added, personal computers are almost as expensive as workstations. Current workstations have as much memory and are as fast as the mainframes of a few years ago. Because of their much better price performance, workstations have made minicomputers and mainframes obsolete for many purposes.

Hardware advances

Trends that will affect computer systems in the next few years include hardware advances like parallel processing, practical storage of audio and video, and more efficient communication. Although massively parallel machines such as Thinking Machines Corp's 64 000 processor CM-2 exist, they are not yet recommended for production use because of their cost, scarcity, and lack of software. Operating systems are still rudimentary; for example, there may not even be code to buffer or efficiently route messages between processors. In the future, a generally useful parallel machine will probably involve dozens of powerful processors, such as might be used alone in a workstation, and each processor will have a considerable amount of memory. At the present time, the best method actually to compute something is to use a fast serial machine. The best reason for a parallel implementation is to learn the techniques for the time when parallel machines become useful. This must happen sometime since the speed of a single processor is limited by physical laws such as the speed of light, some machines already being so fast that an electrical signal can travel only 10 centimetres during one cycle of the central processing unit.

Parallel machines may be classified by whether the multiple processors execute the same instruction – SIMD (single instruction multiple data), or whether each processor may execute a different instruction – MIMD (multiple instruction multiple data). SIMD machines, such as the CM-2, are simpler since the same instruction is decoded and then broadcast to all the processors which have separate memories. To implement an if–then–else, processors may also be selectively disabled from executing instructions. In the following code:

if (test) then true-code else false-code

each processor executes *test* to compute a bit. Then those processors whose bit is 0 disable themselves and the remainder execute *true-code*. Finally, those processors whose bit was 1 disable themselves, while the others enable themselves and execute *false-code*.

A MIMD machine is like a Hypercube or Sequent, or like a roomful of workstations. Each processor executes a different program, and communicates with other processors by some network. This is more powerful, but also more expensive per processor than a SIMD machine.

Practical storage and processing of audio and video data will arrive in the next few years, assisted both by larger mass storage, and by sufficient processing power to compress and expand the data in real time. This will be assisted by faster networks – several hundred megabit per second local area networks and megabit per second wide area networks built on telecommunication standards such as ISDN (integrated services digital network), a standard for future communications.

Computers typically have large flat address

spaces of main memory, which means that a programmer can address any one of eight or more Mb equally easily. However, since the processors are faster than the I/O bus, the processors also have on-chip caches of typically 8 Kb to contain the most recently used data. Therefore, programs whose data references hit the cache more often will be more efficient. Even if memory were free, programs would still need to conserve it. Indeed, there is a cost associated with retrieving data over the bus and larger data sets mean fewer cache hits.

Software advances

The term 'software engineering' was coined some years ago in an attempt to imbue the software production process with some of the rigour associated with traditional engineering. That has been a limited success. The goal has been to devise more and more powerful stratagems so that software production becomes more and more automatic. The continued failure to achieve this has led Brooks (1987) to suggest that automated software production is not possible because the problem is inherently difficult.

Whatever software advances occur in computer systems in the future are likely to be less spectacular than in the past. Artificial intelligence is not having the impact predicted a few years ago and formal methods of program design are still being extended (Hoare 1987). Cohen (1988) describes logic programming techniques that look promising for the future. Some applications to geometry and the map overlay problem are described by Franklin and Wu (1987) and Franklin *et al.* (1986). The most important advance would be the more widespread availability of tools that have existed for many years, such as code management and debugging tools, WYSIWYG (what you see is what you get) text processors, database managers, fourth generation languages, and so on (Boehm 1987; Rich and Waters 1988). Many universities have no access to such tools as already exist.

The other significant software advance is the spread of standards, such as the Unix operating system and X Windows. Standards allow portability not only of the program to other machines and projects, but also of the programmer to other projects. Frequently a standard is not quite state-of-the-art, such as vendors' proprietary versions of Fortran that each contain a few features unique to that vendor that are not in the standard. A programmer must consider whether the extra advantages of these features compensate for being locked in to that vendor.

LOW-LEVEL DATA STRUCTURES

Abstract data structures versus their physical realizations

When designing a data structure, it is important to separate the abstract data structure from its physical realization (Aho, Hopcroft and Ullman 1983). This separation is assisted by newer languages with information hiding, such as Ada (Pyle 1985), or object-oriented languages, such as C++ (Stroustrup 1987) and Smalltalk (Goldberg 1984). However, these newer languages are usually not universal enough, or are too inefficient, to use in GIS systems. Programmers must, therefore, enforce this separation themselves.

The abstract data structure is defined by the set of elements that are to be stored, the operations to be performed on them and, perhaps, the possible error conditions. For example, consider the problem of choosing a data structure for representing a map, or planar graph.

1. *Elements*: Polygons representing states or other regions.
2. *Operations*:
   (a) Draw the whole map or any polygon.
   (b) Verify the internal consistency of a map.
   (c) Calculate any property of a polygon, such as area.
   (d) Find the adjacent polygons to a polygon.
   (e) Scale a map or transform it to another coordinate system.
   (f) Overlay two maps.
3. *Design criteria*:
   (a) Amount of storage used.
   (b) Likelihood of errors.

(c) Ease of implementation.
(d) Efficiency of execution.

The physical realization of the data structure covers how it is actually implemented. There are two major choices here, depending on the major element of the database.

- *Polygon based*: The polygons are stored explicitly, as a sequence of points. Thus each point on an interior edge is stored twice, once for each of the adjacent polygons. With this method, it is easy to operate on individual polygons. However, more storage is used, and inconsistencies can arise if the two versions of each point are not identical down to the last bit.
- *Edge based*: Here the edges or chains separating the polygons are stored. The polygons exist only in the ID numbers used to refer to them. Each edge contains the numbers of its two adjacent polygons. This method is more compact than the previous one. However, operating on one polygon requires finding all the edges on it.

With the edge-based method there are two choices, whether to store a complete chain of points, or to store each edge separately. The former is more compact but has variable length data elements that are more complicated to work with.

A clean implementation of the logical map data structure would not allow most of the program direct access to the data. Instead the desired operations would be decomposed into the primitive operations like the following:

- Return the total number of polygons.
- Return the external name of the polygon that is internally *i*.
- Return the number of points in polygon *i*.
- Return point *j* of polygon *i*.

Then a user-callable procedure would be made available for each primitive operation.

The advantage of this separation of logical from physical realization is that the implementation may be changed as the system evolves, as experience with the data structure's actual use is obtained by performance monitoring. The prime disadvantage is that in a low level language, such as one without macros, the clean implementation of this concept requires many subroutine calls, which is clumsy and slow.

Examples of data structures

Some examples of generally useful data structures are presented in this section. For a broader guide to the field consult any standard text, such as Knuth (1973) or Aho, Hopcroft and Ullman (1983).

Hashing

In abstract terms, a hash table is a mapping from a key, such as a character string or a number, to an entry in the table. Unlike an array where the elements are addressed 1,2,3,..., in a hash table, a list of 10 countries can be accessed by their names, or information about 100 people can be stored and accessed by their 9 digit social security number. The following is a brief description that conveys the flavour of hashing while omitting complexities and options.

Suppose the key K is an integer in the range $1...B]$. If there are N records to store, a table of size $M = 3N/2$ or so is allocated. The problem is to reduce a key ranging up to B to a table location L ranging up to M. The simplest method is to use $L = (K \text{ modulo } M) + 1$, where modulo is the remainder function (e.g. 10 modulo 3 = 1 because dividing 10 by 3 leaves a remainder of 1). The + 1 occurs because modulo returns numbers in the range $0...M - 1]$ whereas the table is assumed to be indexed from 1 to M.

The collision of two different records that hash to the same location can be solved in two ways. All the colliding records may be chained together with a linked list. Alternatively, the colliding record may be placed in the next available location, which means that when retrieving a record its hash location must be computed and successive locations checked until a free location is seen. If the table is nearly full, then runs of consecutive used table locations can grow surprisingly. If the load factor, or fraction of table slots occupied, is α, then the average number of slots that must be examined in an unsuccessful search for a key is $(1 + 1/(1-\alpha)^2)/2$. In our example, with $\alpha = 2/3$, this is 4–5 slots. If $\alpha = 0.9$, this would be 50, which is totally unacceptable.

734

On the other hand, if the overflow records are chained together, then the pointers occupy storage.

If the key is a character string, such as *Switzerland*, then each 4 bytes can be considered as an integer. In this case, the three integers which have bit codes corresponding to *Swit*, *zerl*, and *and* must be combined into one integer, perhaps by multiplying the ith integer by i, ignoring overflows, adding them, and taking the absolute value. Then the above modulo process can be used.

Hash tables are very efficient for inserting and retrieving records when the total number is known in advance. They are less efficient for deleting records and for growing tables dynamically to an initially unknown size, since the whole table must be recomputed in a larger area whenever it overflows. It is impossible to retrieve the records in order from a hash table without sorting them. If these operations are necessary, a more complicated data structure, such as a B-tree, is preferable.

Extendible arrays

Often there is a requirement to use a logical array, where read and write elements are keyed by the subscripts which are small integers from 1 up, but where there is no *a priori* idea of how large the array will grow. None of the conventional choices for implementation are appealing. A binary tree carries a large time overhead of $\theta(\log(N))$ to access an element, if there are already N elements (the notation $\theta(\log(N))$ means that time rises with the logarithm of the number of elements. It also carries a coding overhead in that unless the tree is kept balanced by rotation, that access time may deteriorate to $\theta(N)$ (rising more steeply, in proportion to N). A hash table has the earlier mentioned problems. A simple array requires advance estimation of the maximum likely N, and the array must be copied over if this is exceeded. Therefore, this last choice is not often used.

However, the extendible array, where the array is reallocated and recopied when it overflows, is actually quite efficient. Assume that each such time, a new array is allocated that is bigger by a factor of α. Then if the array grows from a size of 1 up to a very large size, the average element is copied only $1/(\alpha-1)$ times. If the array is doubled each time, then the average element is copied only once. Indeed, the last half of the elements are never copied after being first inserted. The previous quarter are copied once, the previous eighth are copied twice, and so on. In a language such as C, the array can be grown and copied with one procedure call, to *realloc*. If there is a restricted language with only static storage allocation at compile time, then special dynamic allocation routines must be implemented to suballocate storage from a large static array. However, this must be undertaken for any of the methods.

Thus extendible arrays are an efficient method for implementing arrays when the initial size is unknown.

Uniform grids

A fundamental low-level operation in overlaying maps and testing for interference is that of finding all the intersections among a large set of small edges. The theoretically optimal method is by Chazelle and Edelsbrunner (1988), which finds all K intersections of N edges in time $\theta((K + N)\log N)$. It has been implemented, but is complicated and is not obviously parallelizable. A simpler method using a scan line is by Bentley and Ottmann (1979), with slightly larger time. Note that if $K = N^2/2$ then this is worse than the naive time of $\theta(N^2)$.

The uniform grid of Franklin *et al.* (1988, 1989, 1990) is an alternative method. It is simple to program, parallelizable, and has expected execution time $\theta(K + N)$. The worst case time is $\theta(N^2)$, but this has not been observed in practice. The uniform grid is a flat, non-hierarchical grid overlaid on the edges to be intersected. The grid size is a function of the number and length of the edges. The actual size is not critical; a factor of three variation either way from the optimum tends to increase the execution time less than 50 per cent. The method is fast even for unevenly spaced data; hierarchical methods such as quadtrees are not necessary here. When implemented on a 16 processor Sequent Balance 21000, the program runs 10 times as fast as when one processor is used. When implemented on a 32 processor hypercube, the communication costs are only one-third of the total time, and the slowest processor to finish is only twice as slow as the average time. On a Sun 4/280 serial machine, finding all 144 666 intersections of the 116 896 edges of the US Geological Survey's Digital Line Graph sampler tape (of Chicamauga, Tennessee) takes only 37 CPU seconds.

The technique also extends to higher level operations, such as finding the areas of the intersection polygons resulting from overlaying two

73-5

maps (Franklin 1990). When applied to two maps, the conterminous states of the United States, with 1081 vertices, 892 edges, and 49 polygons, and July isotherms at 10°F intervals, with 920 vertices, 892 edges, and 6 polygons, calculating the areas of all the output polygons, once the data had been read in, took only 7.2 CPU seconds on a Sun 3/60.

Voronoi diagrams

How should a set of N (one-dimensional) numbers be stored so that the closest existing number to a new one can be located? The obvious data structure is a sorted array. Sorting the array to build the structure takes $T_{preprocessing}(N) = \theta(N\log N)$, while determining the closest old number to some new one takes $T_{query}(N) = \theta(\log N)$ with a binary search.

A Voronoi diagram (Preparata and Shamos 1985; Sedgewick 1983) allows N 2-D points to be stored so that it is possible to insert a new point, delete a point, determine the closest old point to a new point, and perform many other location problems in time $T_{query}(N) = \theta(\log N)$. The Voronoi diagram, which can be built in $T_{preprocessing}(N) = \theta(N\log N)$, partitions the plane into a set of polygons, also called Thiessen polygons, one around each input point. The dual of this diagram, called a Delaunay triangulation, has an edge joining a pair of original points whenever their polygons are adjacent. This is useful for interpolation of a value at any point given the values of some function only at the set of N randomly located input points. To calculate the function value at the new point, interpolate the values at the vertices of the Delaunay triangle containing it.

Generalized Voronoi diagrams allow other objects than just points (Drysdale 1979). A scene with points, edges, circles, and so on, can be pre-processed so that when a new point is presented, the closest old object can be determined quickly. However, generalized Voronoi diagrams are much more complicated to implement.

Large databases

Many data structure textbooks are not completely appropriate for the size of data structures that are reasonable to use today, where a workstation may have several megabytes of memory and many megabytes of disk. For example, consider that perennial favourite the binary tree, if used to organize 1 million 4-byte integers. If the pointer to each integer also requires 4 bytes, then this tree would occupy 8 megabytes of memory. Why would this tree be the wrong solution?

If the tree were perfectly balanced, it would be 20 levels deep, so that accessing any element would require following 20 pointers, probably to 20 widely separated locations in memory. This would activate 20 pages in the cache or virtual memory handler. Worse, the tree could not be perfectly balanced if it were being modified. A practical tree here would be an AVL tree (Aho, Hopcroft and Ullman 1983), which allows the left and right sub-trees of each node to differ in height by up to one. An AVL tree with 1 million elements would be perhaps 35 levels deep, which is even worse. Basically a binary tree is obsolete for large databases.

A B-tree, where each node can have from M to $2M-1$ sons for some fixed M, such as 100, would be much more efficient, and in this case would result in a tree about six levels deep. The B-tree can also be used to organize data on a disk. If M is made so large that one node is the size of one track of the disk, or whatever the most efficient quantity of data is to read, and if the first two levels or so of the tree are stored in main memory, then any element of a 10^9 byte tree can be read in one or two disk accesses.

For smaller data sets, the best version of a binary tree is the splay tree (Tarjan 1987). Here the tree is readjusted during every query to move that record to the root. This is not so bad as it sounds since locating the record requires $\theta(\log N)$ work and moving the record to the root requires only additional work proportional to that. This causes the average access time to be minimized even when some elements are retrieved more often than others.

ALGORITHMS

Theoretical analysis

Deep theoretical analysis of algorithms and data structures is desirable when possible. Some standard books dealing with this include Aho, Hopcroft and Ullman (1983) and Knuth (1973). Tarjan's Turing Award lecture (Tarjan 1987) is also excellent. Unfortunately, this analysis is subject to

error. Even highly selective theoretical computer science conferences and monographs have published repeated errors. A more serious limitation, at the moment, is that what is theoretically analysable is not always what is most needed. Cartography needs expected time analysis as much as worst case analysis; however, expected time analysis is still almost impossible for sophisticated data structures.

Some useful notation for theoretical analysis follows. $T(N) = \theta(f(N))$ means that the worst-case execution time of an algorithm for all problems of size N grows proportional to $f(N)$ as $N \to \infty$. $T(N) = \theta(f(N))$ means that the execution time grows at most as fast as $f(N)$, and $T(N) = \Omega(f(N))$ means that it grows at least as fast as $f(N)$. Thus since it is known how to multiply two $N \times N$ matrices in cubic time, and that it takes at least quadratic time, therefore, here $T_{matrix\text{-}mult}(N) = \theta(N^2)$ and $T_{matrix\text{-}mult}(N) = \Omega(N^2)$. The actual minimum time is not known. This notation, which was devised to hide dependencies of the time on particular machines, can hide too much. If one algorithm's time is N^4, while another's is $1\,000\,000N^2$, then the latter is asymptotically faster, but practically slower for $N < 1\,000\,000$. Finally, it is necessary to distinguish between a particular problem and the various algorithms to solve it. The problem of matrix multiplication has several algorithms, from the naive, with $T_{naive}(N) = \theta(N^3)$, to the Schönhage–Strassen (Schönhage and Strassen 1971), with $T_{SS}(N) = \theta(N^{2.81})$, to even (asymptotically) faster ones.

Expected time analysis is difficult in cartography because the statistical distribution of the input data is not well characterized. However, it is known that cartographic data can be distributed much worse than if it were independent and uniform. Consider, for example, the edges of a map of a city such as Chicago, where each edge is one block of a street. Along each street there will be dozens of correlated, collinear, edges, which could not happen if the edges were random. This severe degeneracy can cause problems for a scan line algorithm.

The US Geological Survey's Digital Line Graph sampler tape illustrates how statistically unusual GIS data can be. If the 116 896 piecewise straight edges are scaled to fall in a 1000×1000 square, then the mean edge length is 2, but with a standard deviation of 8, which is a very skewed distribution (since there are no negative lengths). Another unusual feature is that 34 edges have zero length, to six significant digits. This non-randomness, which is typical of cartographic databases, can destroy an algorithm that makes uniformity or normality assumptions.

Randomized algorithms

A powerful recent concept in algorithm design is randomization, that is, the algorithm 'flips a coin' and alters its actions depending on the outcome. A simple example of this is when running a scan line up a database of city streets, where there is one edge per block. For each position there might be a requirement to process the active edges, or those which cross that scan line. If both the scan line and the streets are horizontal, when the scan line coincides with a long street then there will be many horizontal active edges. If the program is comparing all active edges against each other, perhaps to build up chains, then this can be a serious problem. The solution is to rotate the map by a random angle before processing.

Another example occurs in the operation of sorting N numbers by a quicksort. Here it is possible to proceed by divide-and-conquer, a common method of algorithm design. A pivot number is chosen from the set and the set is partitioned into those numbers smaller than, those equal to, and those larger than the pivot. The first and last subsets are then recursively quicksorted. If all inputs are equally likely then the average time is $E(T(N)) = \theta(N \log N)$, which is quite good. The problem occurs if the pivot element is the smallest number in the set, which will happen if the set is already sorted, which is not so unusual, and the first element is chosen as the pivot. Now $T(N) = \theta(N^2)$, which is totally unacceptable. However, if the pivot element is selected randomly, there are now no bad inputs. Even if the set is already sorted, $E(T(N)) = \theta(N \log N)$, where there is averaging over different outputs of the random number generator. Every time the algorithm is run on some fixed input, it will take a different time. However, the average of all those times for the same input is fast.

The virtue of simplicity

For many low-level data structures and algorithms, it has been determined after extensive analysis that

73-7

the best choice is the simplest. For example, in random number generation, a linear congruential generator:

$$x_{i+1} = (7^5 x_i) \bmod (2^{31} - 1)$$

is better than many more complicated methods, including those used in many packages and textbooks (Park and Miller 1988). The coefficients in the above equation were chosen with great care to achieve this.

In hashing, a simple modulo operation for the key to bucket transformation is excellent, and for handling overflows, similar simple methods are adequate. A second example may be used to illustrate this point. In virtual memory paging, an important question is which current page is to be paged out so that a new page may come in to satisfy a page fault. The simple concept of writing out the LRU (least recently used) page is quite adequate. Finally, there are many problems, called NP (non-deterministic polynomial time), for which the best known (deterministic) algorithm requires exponential time (Garey and Johnson 1979). However, many of them, such as the travelling salesperson problem, have obvious heuristics with which an almost optimal solution can be found in linear time.

Finally, when locating a number in a long sorted list, guessing where the number should be usually beats a binary search (Perl, Itai and Avni 1978). For example, when trying to locate the number 73 in a list of 100 independent numbers ranging from 1 to 1000, the first probe should be at the seventh number, not at the fiftieth. With this interpolation searching, if the numbers are independently and uniformly distributed, then the average number of probes to find a number in a sorted list of N numbers is $E(T_{query}) = \log_2(\log_2 N)$, although the worst case is $T_{query} \leq N$. This can be compared to the average and maximum in a binary search, which are both $T_{query} = \log_2 N$. For example, if $N = 1$ million, then binary search takes 20 probes, while interpolation search may take about five.

Of course heuristics do fail at times. In addition, simple concepts may not be simple to implement in practice; as Einstein said, 'A theory should be as simple as possible but no simpler'. When attempting to wring the last bit of performance out of a system, a design might get quite complicated. However, if machine speeds are doubling each year, then a 40 per cent increase in speed can be achieved by waiting six months for a new machine, which is better than spending more than six months getting that same degree of improvement from the software.

Implementability

A major advantage of simplicity is that the system is then more implementable. Instead of programmers spending all their time getting the system just to compile correctly, there is time to profile and improve the working system. In addition, the largest feasible system is determined by the point at which details are forgotten and new bugs are created as fast as system details are remembered and old bugs fixed. Simpler systems mean that the largest feasible system will have greater functionality.

Size–time trade-offs

Many data structures and algorithms have different possible versions depending on the relative costs of memory versus CPU cycles. For instance, consider allocating memory to a routine whose needs will gradually increase, as in the extendible array described above. When the block that has been allocated is full, a bigger region must be allocated, the complete old block copied over into the new region and any pointers updated. If bigger blocks are allocated then more memory is used but reallocations will be fewer. Another example occurs in storing and transmitting digital images. It is possible to compress typical 8 bit per pixel black and white images by a factor of ten with little visible degradation of the image quality. A video image such as a TV transmission can be compressed by a factor of 100 by the expenditure of considerable, and so far impracticable, processing. In the design of hardware, such as multipliers, there is also a continuum of trade-offs between multiplication time and cost.

Nevertheless, sometimes smaller is also faster. Many small computers are limited by the speeds of the I/O bus and memory since those technologies are advancing more slowly than processor technology. If some very big, often accessed, data structure can use, say, 2 byte integers instead of 4

73-8

byte integers, then I/O requirements are halved, and twice as many data elements will fit in the cache, raising the cache hit percentage. The disadvantage is that the program will hit a hard limit when 2 bytes becomes insufficient. Hard limits such as this are the most common reason for the death of a type of hardware. Consider, for example, the enormous problems caused for users of IBM PCs by Intel's decision to use first 16 and then 20 bit addressing for the CPU in the PC. This restricts the system to 2^{20}, or 1 megabyte of memory. After the operating system's requirements, the user is left with only 640 Kilobytes of memory. Thankfully, the processors in the IBM PS/2s have removed this restriction.

Given relative costs of memory, disk space and processing power, it is possible to calculate how often a word in memory must be referenced before it is better to swap it out to disk, and the trade-offs between storing and recalculating intermediate results. A good summary of data compression techniques, useful for implementing the tradeoff, is Lelewer and Hirschberg (1987).

SOFTWARE DEVELOPMENT STRATEGY

Waterfall model

The traditional software development strategy is the waterfall model, named from a popular method of diagramming the process, which has several stages. This can be illustrated with the problem of determining the area of New York State above 1000 metres elevation.

1. *Requirements analysis*: Global analysis of the problem, deciding whether to use satellite photographs, US Geological Survey databases, or even renting a plane, flying around the state and estimating by eye. This step is about 5 per cent of total life-cycle system cost.
2. *Specification*: Deciding the user-visible parts of the program, such as commands and database formats – 5 per cent.
3. *Design*: Designing the internal details of the system, such as data structures and subroutine interfaces – 10 per cent.
4. *Coding*: Actually writing the programs – 5 per cent.
5. *Debugging and testing*: Getting the code to work, and, if verification and validation are involved, proving its correctness. The system is released to the customer at the conclusion of this step – 25 per cent.
6. *Maintenance*: Modifying the released system to account for new hardware or users' needs, or to fix newly discovered bugs – 50 per cent.

This model is best when it is possible to predict in advance the system requirements. It is now being partly replaced by rapid prototyping.

Rapid prototyping

Here a small prototype with stubs of the major components is readied for the user's comments. Then new functions are gradually added. This has the advantage of better feedback from users, who may be unable to specify the requirements in advance, but will know what they want when they see it. However, the evolutionary development may allow major irreversible decisions to be taken before all the implications are known, and this can lock the designer into a bad choice.

For example, suppose a mapping display system is being designed. It might be pointless to specify everything, such as legend format, in detail in advance, if it is necessary to see some finished maps to know whether the total effect was appropriate. On the other hand, through lack of planning ahead, a data structure might be picked that does not allow later program porting from a low resolution colour display device to a high resolution black and white one. This might happen if certain special, unused, values were used for the colour pixels to code for properties such as the pixels used for the legend. A higher resolution device, with only 1 bit per pixel, would have no unused pixel values and allocating extra bits per pixel might not be possible because of the total number of pixels.

Another case of an innocent early decision causing later problems appears when porting a system written for European users into Japanese. The idea that message characters are chosen from a small possible set, such as ASCII, each character

73-9

representable in 7 or 8 bits, is so thoroughly embedded in most program libraries that converting to Kanji is a major task. Even the appropriate data structures are totally different. Dispatch tables, which are used to parse input text by branching on each character, are more suited to 128 entries than to 2000 entries.

Profiling and improvement

It is impossible to predict in advance how most systems will be used and which parts will consume the most resources. Thus it is important to profile the system in actual use to determine which parts should be improved. Then the data structures and algorithms that are too simple can be selectively upgraded to more complicated but faster versions. This is how Unix was originally developed. It was all written in C, then profiled and about 10 per cent recoded in machine language.

In Unix, typical profiling tools include *lint* to detect most syntax errors and to perform an elementary static flow analysis to detect unused variables, *prof* to profile the CPU time used in each procedure, and *tcov* to count the number of times each statement is executed (Sun Microsystems 1985). With these tools the critical parts of the system can selectively be improved.

Profiling a system, perhaps by adding extra counters and timers, is also critical in understanding how a complicated program is actually behaving inside. For example, how full does a hash table get? How many collisions occur during insertion, or during retrieval? When splitting edges with a uniform grid, how many segments does the average edge get split into, or the worst edge? How many edges does the average cell contain, or the worst cell? Most people have not the faintest idea. The process of software development does not end when the program compiles correctly, or even when it can process a small test case. To produce a work of art, cleanly designed and coded, and efficient, requires following systematic design steps and then watching how the program performs in actual use.

CONCLUSION

If computer science has any meaning other than as a sterile intellectual exercise, it is to help other disciplines use the computer more effectively. GIS practitioners need not recapitulate computer scientists' painful process of learning if they can learn from their experience, as described above in this chapter. It is not sufficient to design GIS to use current computer systems; they must be planned for the computer systems that will be available when the new GIS is ready.

REFERENCES

Aho A V, Hopcroft J E, Ullman J D (1983) *Data Structures and Algorithms*. Addison-Wesley, Reading Massachusetts
AT&T Bell Laboratories (1978) *Bell System Technical Journal* **57** (6)
AT&T Bell Laboratories (1984) *Bell System Technical Journal* **63** (8)

Bentley J L, Ottmann T A (1979) Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computing* **C-28** (9): 643–7
Boehm B W (1987) Improving software productivity. *Computer (IEEE)* **20** (9): 43–57
Brooks F P (1987) No silver bullet – essence and accidents of software engineering. *Computer (IEEE)* **20** (4): 10–19

Chazelle B, Edelsbrunner H (1988) An optimal algorithm for intersecting line segments in the plane. *Proceedings, 29th Annual Symposium on Foundations of Computer Science, White Plains*
Cohen J (1988) A view of the origins and development of prolog. *Communications ACM* **31** (1): 26–36

Drysdale R L (1979) *Generalized Voronoi Diagrams and Geometric Searching*. Unpublished PhD dissertation, Department of Computer Science, Stanford University

Franklin W R (1990) Calculating map overlay polygon areas without explicitly calculating the polygons – implementation. *Proceedings of the 4th International Symposium on Spatial Data Handling*. Zurich International Geographical Union, Ohio, pp. 151–60
Franklin W R, Wu P Y F (1987) A polygon overlay system in prolog. *Proceedings of AUTOCARTO8*. ASPRS/ ACSM, Falls Church Virginia, pp. 97–106
Franklin W R, Wu P Y F, Samaddar S, Nichols M (1986) Geometry in prolog. In: Kunii T (ed.) *Advanced Computer Graphics, Proceedings of Computer Graphics Tokyo '86*, pp. 71–8
Franklin W R, Chandrasekhar N, Kankanhalli M, Seshan M, Akman V (1988) Efficiency of uniform grids for intersection detection on serial and parallel machines. In: Magnenat-Thalmann N, Thalmann D (eds.) *New Trends in Computer Graphics (Proceedings, Computer Graphics International '88)*. Springer-Verlag, New York
Franklin W R, Chandrasekhar N, Kankanhalli M, Sun D,

73-10

Zhou M-C, Wu P Y F (1989) Uniform grids: a technique for intersection detection on serial and parallel machines. *Proceedings of AUTOCARTO9*. ASPRS/ACSM, Bethesda Maryland, pp. 100–9

Franklin W R, Chandrasekhar N, Kankanhalli M, Akman V, Wu P Y F (1990) Efficient geometric operations for CAD. In Wozny M J, Turner J U, Preiss K (eds.) *Geometric Modeling for Product Engineering*. Elsevier, Amsterdam, pp. 485–98

Garey M R, Johnson D S (1979) *Computers and Intractibility: a guide to the theory of incompleteness*. Freeman, San Francisco

Goldberg A (1984) *Smalltalk-80: the interactive programming environment*. Addison-Wesley, Reading Massachusetts

Hoare C A R (1987) An overview of some formal methods for program design. *Computer (IEEE)* **20** (9): 85–91

Knuth D E (1973) *The Art of Computer Programming*. Addison-Wesley, Reading Massachusetts

Lelewer D A, Hirschberg D S (1987) Data compression. *ACM Computing Surveys* **19** (3): 261–96

Mel B W, Omohundro S M, Robinson A D, Skiena S S, Thearling K H, Young L T, Wolfram S (1988) Tablet: personal computer in the year 2000. *Communications ACM* **31** (6): 639–46

Park S E, Miller K W (1988) Random number generators: good ones are hard to find. *Communications ACM* **31** (10): 1192–201

Perl Y, Itai A, Avni H (1978) Interpolation search – a log log n search. *Communications ACM* **21** (7): 550–3

Preparata F P, Shamos M I (1985) *Computational Geometry: an introduction*. Springer-Verlag, New York

Pyle I C (1985) *The Ada Programming Language: a guide for programmers*. Prentice-Hall, Englewood Cliffs New Jersey

Quarterman J S, Silberschatz A, Peterson J L (1985) 4.2bsd and 4.3bsd as examples of the Unix system. *ACM Computing Surveys* **17** (4): 379–418

Rich C, Waters R C (1988) The programmer's apprentice: a research overview. *Computer (IEEE)* **21** (11): 10–25

Schönhage S, Strassen V (1971) Schnelle multiplikation grosser zahlen. *Computing* **7**: 281–92

Sedgewick R (1983) *Algorithms*. Addison-Wesley, Reading Massachusetts

Stroustrup B (1987) *The C++ Programming Language*. Addison-Wesley, Reading Massachusetts

Sun Microsystems (1985) *Programming Utilities for the Sun Workstation*. Sun Microsystems

Tarjan R E (1987) Algorithm design. *Communications ACM* **30** (3): 205–12

7311