A Tutorial on Database Technology for Geospatial Applications DRAFT Confidential

T. H. Merrett[*], Y. Bédard[†], D. J. Coleman[‡], J. Han[§], B. Moulin[¶], B. Nickerson[||], C. V. Tao[**]

May 27, 2002

[1]

Abstract

We show how to use a general-purpose programming language for databases on secondary storage, with no specifically spatial constructs, to implement algorithms from computational geometry, data warehousing, data mining, and Internet collaboration, for geographical information systems.

Index Terms relational, multidimensional, object-oriented; edge-intersection, overlay, point-in-polygon, Voronoi; data cube; data mining: association, classification, generalization; collaboration, concurrency, events.

Contents

[*]School of Computer Science, McGill University, 3480 University, Montréal, Canada H3A 2A7. E-mail: tim@cs.mcgill.ca

[†]Dept. des sciences géomatiques, Université Laval, Courriel: Yvan.Bedard@scg.ulaval.ca

[‡]Dept. of Geodesy & Geomatics Engineering, University of New Brunswick, E-mail: dcoleman@unb.ca

[§]Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, E-mail: hanj@cs.uiuc.edu

[¶]Dept. d'informatique, Université Laval, Courriel: bernard.moulin@ift.laval.ca

[||]Dept. Computer Science, University of New Brunswick, E-mail: bgn@unb.ca

[**]Dept. of Earth and Atmospheric Science, York University, E-mail: tao@yorku.ca

1 Introduction

Geospatial data typically comes in very large quantities. For example, the 13,000 maps in the 1:50,000 series from Energy, Mines and Resources, Canada need gigabytes (10^9 bytes, or a thousand megabytes) in raw form. The NASA Earth Observing System generates petabytes

(10^{15} bytes, or a million gigabytes) per year. These quantites are too big for RAM, the main memory, of most computers, and so the data requires secondary storage.

This tutorial describes geospatial data applications—geographical information systems, spatial data warehousing and mining, collaboration over the Internet with geospatial data—on the basis of a unified programming language for secondary storage. It attempts to end the gap that has grown between processing purely spatial data and processing the associated non-spatial data. This gap has become the norm in mapping and geospatial data analysis.

Since programming languages for primary memory (FORTRAN, Lisp, Algol, Pascal, C, Java, etc.) make no special provisions for spatial data (and yet are used for spatial programming), it is anomalous that languages for secondary storage do. This tutorial shows that such special provisions are not necessary if the language is well conceived. Once such special provisions are not needed, integration of spatial with non-spatial processing becomes straightforward.

Spatial *data structures* are needed, and the language must support them. We adapt a significant spatial data structure, the quad-edge representation, to secondary storage, and show how to program with it. Our language, however, does not depend on adopting this one particular data structure.

Secondary storage languages have mostly been database languages, and these have mostly been intended for querying and transaction processing (updating). These are limitations in the technology of databases, and yet databases have significant ideas and methods to offer. For this reason, this tutorial is based on database technology, and seeks to build on its strengths while reducing its limitations.

Just as spatiotemporal data should require no special provisions in the programing language, so should there be no language distinction between transactions, on one hand, and analysis, on the other. We discuss data warehousing and data mining, and, of course, spatial versions of each.

Elsewhere [7, 18] we have discussed tools and available software in the global context. In the present tutorial, we focus on the underlying structures, which can be built into new tools and systems. We will discuss and illustrate the basic operators of *programming* languages for geospatial applications, so that the reader can think directly about significant problems, rather than look at packages for the reader to buy. This tutorial treats G.I.S as an intellectual challenge, rather than as a systems problem.

Because this is a tutorial, our treatment is intuitive, by examples rather than theorems or formal algorithms. The examples are simple, even "toy" examples: it is the the role of the *language* to be able to scale them up. The tutorial makes clear that such scaling is implicit in the approach and, thus, how to do it.

The examples are directed towards conceptual ease rather than machine efficiency: we intend to show how to apply secondary storage programming constructs rather than to teach practitioners what they already know about geospatial algorithms, data warehousing, data mining, or collaborative work.

The paper is a tutorial in that it attempts to be conceptually self-contained, which is why formal material is kept to a minimum.

The next section discusses the nature of secondary storage and characterizes the level of abstraction needed for effective programming on secondary storage.

Section 3 reviews spatial data processing, starting with the capabilities of commercial and some potential geographic information systems. It looks, more deeply, at the recent discipline of computational geometry, and at some not yet assimilated results from mathematical geometry. We look at other aspects of spatial data, notably computer-aided design in both engineering and architecture. There is a considerable discussion of spatial predicates and the hierarchies that can be used to approximate them.

Section 4 introduces the quad-edge data structure, which we will use for spatial data throughout the tutorial. We point out that this data structure suits the needed level of abstraction for secondary storage while other spatial data structures do not, and we motivate it by starting with less sophisticated data structures.

In section 5, we turn to the domain algebra. This complements the relational algebra and is central to all of the work that follows in the tutorial. Although SQL can also be augmented by a complementary domain algebra, it has limitations, and we introduce a cleaner syntax for the basic operations of the relational algebra, select, project, and three joins. We show how subsuming the relational algebra into the domain algebra gives relational nesting for free, and that nested relations simplify our thinking about some problems, without adding (or needing to add) new functionality to the programming formalism.

With data structure and programming formalism behind us, we can tackle representative spatial problems. We treat polygon and map overlays and Voronoi diagram construction thoroughly (for a tutorial level of discussion, that is). The problems of finding the intersection of sets of edges and of finding which points are in which triangles are solved as steps in these discussions.

Section 7 applies the programming constructs of section 5 to on-line analytical processing, and discusses spatial variants of this technique.

Classification data mining can be seen as an application of OLAP, and section 8 shows how to build this as well as association and generalization mining engines in the secondary storage programming language. There is an extended discussion of geospatial mining of all three types.

The Internet is essential for sharing spatial data such as large maps, and we next look at techniques for cooperative work on maps represented by the quad-edge data structure in our language. This involves treatment of concurrency both by process synchronization and by event programming. The notion of editors as operators in the relational algebra is elaborated as a basis for the user interfaces. As an example, we discuss distributed workflow in collaborative mapmaking.

Finally, in section 10, we look at the place of object-orientation in secondary-storage programming. This is a suite of programming language techniques, which fits readily into the high-level relational approach. The ideas of both procedural abstraction and data abstraction, from programming languages, are covered on the way. Two examples are given: a map abstract data type, and geospatial distributed objects.

There are two appendices, a technical one (A) on the equivalence of two competing data representations for spatial and other data, and one on terminology (B), since the tutorial covers fields in which terminology varies widely and at times contradictorily.

The tutorial springs from the collaboration of the authors in a Canada-wide research project on geospatial decision-making under the aegis of the Networks of Centres of Excellence program. This has brought a wide range of interests and technical abilities fruitfully together into a single group in a rather unusual way, with the result that the approach of this tutorial is different and innovative.

2 Primitive Operations for Data on Secondary Storage

Secondary storage, whether disk storage as is usually the case, or more elaborate technologies, differs fundamentally from primary memory, or RAM. This difference is essentially that finding data on secondary storage takes much more time than transferring it, once found, to be processed. This is basically because the seek operations are mechanical (e.g., disk rotation and read head movement) while the transfer operations are electronic. The ratio of the seek time to the time required to transfer one byte is, on modern disks, close to one million, and is getting bigger as technology advances. (Fifteen years ago, it was about ten thousand.) RAM, by contrast, has an "access-transfer" ratio of about one.

This difference between secondary storage and RAM means that all computer operations involving secondary storage must be treated quite differently from the conventions that have been developed for computing when the data can be held entirely in RAM. (In many cases, sufficient data for a computation can be transferred from secondary storage to RAM, and then processing can proceed by conventional means. There are enough occasions when this is not so, especially with geospatial data, that we must learn how to handle them.)

The basic consequence of the different memory architecture governing secondary storage, the large access-transfer ratio, is that data on secondary storage must be transferred in large "blocks" or "pages". For instance, suppose we are randomly (from the point of view of the storage system) fetching items of data from all over a disk with an access-transfer ratio of a million. Then most of our time will be spent seeking, as opposed to transferring (and presumably processing), unless the blocks are a million bytes or more. Of course, there are many practical reasons why blocks cannot be a megabyte each, but the argument still holds that they should be as large as practical.

This usually means that more than one item of interest is stored in a block. We call such an item a "record", which may be a single number, or, more usually, a collection of related data including numbers, strings, images, etc. If a block size is 8 kilobytes, say, it can hold a fraction of an image, or 100 80-byte conventional IBM card-image records, or 500 eight-byte coordinate pairs, or 2000 4-byte numbers. Where the records are smaller than the block, it is more efficient to process *all* the records an application will need from the block than to refetch the block for each record. For the coordinate pairs example, the difference is one seek followed by an 8K transfer versus 500 seeks followed by 500 8K transfers, a factor of 500. (And if the data were not blocked, the cost would always be 500 seeks followed by 500 16-byte transfers, which is hardly better than the latter, since each seek is equivalent to a 1-megabyte transfer, by the access-transfer ratio of one million.)

Thus, efficient processing on secondary storage must make use of as many records from each block as possible once the block is fetched. This tells us how the data must be organized into the blocks, and what kind of primitive operations we should use on secondary storage.

Data must be "clustered" into the blocks, so that only related records are placed in any block. What is meant by "related" depends entirely on the application and the algorithms involved. Our choice of the basic operations from which algorithms for secondary storage can be programmed must reflect this need for clustering. This means that the primitive operations should not be free to refer to individual records, but only to sets of records. In other words, just as the block abstracts over sets of records, the operations will abstract over looping.

Research into databases has produced a variety of operators for secondary storage. The relational algebra offers the level of abstraction we need in the most accessible way, with the bonus that the *relation* is itself an abstraction over not only blocks but *files*. The operators we shall need are projection and selection (both unary operators, involving single relations), and joins (binary operators which combine two relations). The unary operations process each record of the operand, and the binary operations process each record of both operands, without involving the programmer in any of the looping needed over internal data.

A relation is a set of records, all defined on the same fields, or *attributes*. If the fields are simple, such as numbers or strings, the records and the relation are "flat". Fields may also have values in each record which are in turn relations, in which case the relation is "nested".

We start with flat relations. Here is a representation of a set of polygons as a flat relation. Each polygon, identified by *id*, is a sequence of coordinate pairs. We show two triangles.

polygons(*id*	*seq*	*xcoord*	*ycoord*)
F1	1	3.0	2.0
F1	2	1.0	2.0
F1	3	2.0	1.0
F2	1	3.0	2.0
F2	2	2.0	3.0
F2	3	1.0	2.0

An example of the project operator is to find the set of coordinate pairs representing all (four) vertices of these triangles.

vertices(*xcoord*	*ycoord*)
3.0	2.0
1.0	2.0
2.0	1.0
2.0	3.0

An example of the select operator is to find all data associated with the vertex (`3.0, 2.0`).

vertex12(*id*	*seq*	*xcoord*	*ycoord*)
F1	1	3.0	2.0
F2	1	3.0	2.0

These operators would be written in SQL as, respectively,

select *xcoord, ycoord*
from *polygons*

Projection in SQL

and

select *
from *polygons*
where *xcoord*=`3.0` **and** *ycoord*=`2.0`

Selection in SQL

in which **select** actually means project, and * means project on all fields. Note that selection followed by projection can easily be written in a single phrase. A syntax for assignments permits the resulting relations to be named.

SQL was designed as a query language, not for general programming. We will introduce the missing semantics in section 5. Meanwhile, we show an alternative syntax, which recognizes that projection and selection, as well as relation names by themselves, are expressions. These are usefully combined together and with other expressions to construct programs with capabilities beyond simple querying. For select and project, the new syntax merely rearranges the clauses from the SQL, and uses different keywords so as not to be confused with SQL.

[*xcoord, ycoord*] **in** *polygons*

Projection in Programming Notation

where *xcoord*=`3.0` **and** *ycoord*=`2.0` **in** *polygons*

Selection in Programming Notation

Again, selection followed by projection can easily be written in a single phrase. We will show a compound expression after discussing natural join.

To illustrate join, which is a binary operator, we need two relations as arguments. Suppose we have them as follows.

polygon1(*seq1*	*xcoord*	*ycoord*)
1	3.0	2.0
2	1.0	2.0
3	2.0	1.0

polygon2(*seq2*	*xcoord*	*ycoord*)
1	3.0	2.0
2	2.0	3.0
3	1.0	2.0

(Evidently, each of these can be obtained from *polygons* by a selection followed by a projection. Renaming the *seq* fields is required: section 5.)

Then the natural join of these two on the coordinate pairs will tell us which pairs are common to the two polygons.

sharepairs(*seq1*	*xcoord*	*ycoord*	*seq2*)
1	3.0	2.0	1
2	1.0	2.0	3

In SQL, this is written

select *
from *polygon1*, *polygon2*
where *polygon1.xcoord=polygon2.xcoord* **and** *polygon1.ycoord=polygon2.ycoord*

which closely resembles the SQL syntax for unary operations. In the expression oriented programming syntax, we write

polygon1 **join** *polygon2*

This reveals that the **join** operator selects on equality of the common fields. It is more specialized than the SQL syntax, which allows any logical combination of any conditions, but the syntax makes explicit that a specific high-level operator is involved, as well as being simpler. When we need further binary operators, we can define them explicitly, too: the useful ones turn out not to be expressible with the above SQL.

We can use a compound expression in the new notation to derive *sharepairs* directly from *polygons*, anticipating from section 5 only field renaming. We also show a syntax for assignment.

sharepairs <– ([*seq1, xcoord, ycoord*] **where** *id*=`F1` **in** *polygons*) **join**
([*seq2, xcoord, ycoord*] **where** *id*=`F2` **in** *polygons*);

Changing the above by removing the two selections (but keeping the projections) gives us a join of *polygons* with itself.

overlap <– ([*id1, seq1, xcoord, ycoord*] **in** *polygons*) **join**
([*id2, seq2, xcoord, ycoord*] **in** *polygons*);

overlap(*id1*	*seq1*	*xcoord*	*ycoord*	*id2*	*seq2*)
F1	1	3.0	2.0	F1	1
F1	1	3.0	2.0	F2	1
F2	1	3.0	2.0	F1	1
F2	1	3.0	2.0	F2	1
F1	2	1.0	2.0	F1	3
F1	2	1.0	2.0	F2	2
F2	3	1.0	2.0	F1	3
F2	3	1.0	2.0	F2	2
F1	3	2.0	1.0	F1	3
F2	2	2.0	3.0	F2	2

The result shows an important property of the natural join, that all possible combinations that match on the join (common) fields are in the result. This is a little more explicit in the SQL expression of the same result, but SQL does not always make clear when a natural join is being performed. Once this is recognized as a property of **join**, it is obvious in the more abstract notation that a complete combination of relevant records will result.

Of course, SQL can also express these, by nesting **select**s, and the choice of notation is a matter of taste, up to a point. We will proceed from here, as far as possible, in such a way that any notation for the operators can be used.

In this section, we have introduced three operators at the right level of abstraction for data on secondary storage. The unary operators, project and select, can be combined into a single expression called a *T-selector*, either in SQL syntax, or in a modified syntax which allows a more tidy presentation of compound expressions, for secondary storage *programming*. The binary operator, natural join, is hidden in SQL syntax but explicit in the programming notation. These three operators treat data at the right level of abstraction for secondary storage by ignoring individual records and treating whole files instead. In particular, the implementation is thus permitted to optimize the clustering of records into blocks so that these may be fetched as few times as possible.

There are two circumstances in which the discussion of this section, and of the whole tutorial, may be disregarded. The first is when each item of data is at least as large as any practical block, and so is economical to fetch by itself any time it is needed. This would be the case, for instance, with most images whose internal structure does not interest us. For maps, however, internal structure is precisely what is of interest and what is to be processed by a geospatial system such as a G.I.S.

The second circumstance arises when the result of processing is to retrieve from secondary storage only a very small amount of data, which can fit entirely in RAM, without needing multiple fetches. This happens in many simple queries, or retrievals of small objects. Then clustering is not needed, and naive methods may be used. Maps are large, coherent organizations of data which frequently must be processed in chunks too big for RAM, and clustering applies, as do the levels of abstraction discussed in this tutorial.

Even in these circumstances, when the processing can be done in RAM in the ordinary way, the higher-level abstractions needed for secondary storage can be useful. Because they abstract away from loops and from considering individual records, they also hide many issues of sequencing of operations. These can now be left to the implementation. In particular, the implementation might choose to use parallel processes to execute our higher-level operators. By not requiring the programmer to specify unnecessary sequencing of operations, parallel execution is not hindered by first having to remove this sequencing (the undertaking called "parallelization" in lower-level languages). Parallel processing is beyond the scope of this tutorial, but it is significant in a number of the topics we address [2, 24].

It does require considerable mental effort and acclimatization for an experienced programmer to break the ingrained habit of coding needless sequencing, however. The reader should be alert to the consequent demand for new ways of thinking.

Postscript. We have made a number of points, above, concerning secondary storage and the relational abstraction and operators, which suit it. It is worth stressing four things we have *not* said. First, we have not mentioned *pointers*. Pointers are anathema for secondary storage, at least if they point to data items (e.g., records) that are much smaller than blocks, and if the application is interested in retrieving more than a very few such data items. This is because such pointers invite the programmer to treat secondary storage as if it were RAM, and to write applications that refer repeatedly to the same block instead of permitting the implementation to fetch that block only a minimum number of times. It is a challenge of programming for secondary storage to write algorithms that do not use pointers, but that work at a higher level of abstraction.

The second thing we have not said is that records in a relation somehow represent "objects". In the triangles example, it is fitting to consider the triangles themselves as "objects", and we see that each triangle occupies three records in the relation. This is a prelude to so-called "complex objects", which are often cited as being beyond the capability of the relational formalism. This perceived limitation of relations is an artefact of deciding that an object must correspond to a single tuple (the relational term for a record). So we do not consider "objects" at all, but only secondary storage data structures which can be interpreted any way the user wishes.

The third issue is that, although relations are abstractions of files, we do not say anywhere that a file is how a relation must be implemented. A relation might occupy several files, or one file might be shared by several relations. Although the rows of the tables we have used to show relations are abstractions of records, we do not need to store a row as a record. Even if a relation is stored as a file and each row implemented as a record, we do not say anything about the structure of the file or the representation of the data. The file could be sequential or tree-structured or multidimensional. The data could be stored directly or with shorthands and compression. So we avoid implementation discussions altogether but remain at the higher level of abstraction.

In particular, we have not said that selects must be implemented by scanning a whole file, or that joins must be implemented by cumbersome operations involving whole files. Pointers are acceptable *implementation* devices, and pointer dereferencing is sometimes the appropriate implementation of a join.

Fourth, we have not done any complexity analysis. For relations of n rows, the asymptotic complexities of select, project and join are, respectively, $\mathcal{O}(n), \mathcal{O}(n \log n)$, and $\mathcal{O}(n^2)$, but these are not the most important considerations. For data too large for RAM, the *constants* in the algorithmic analyses are of crucial importance. Moreover, we do not often want complexities higher than $\mathcal{O}(n \log n)$, and then only with a large base for the logarithm. Fortunately, sorting, which is often central to the algorithms (e.g., to eliminate duplicates in projection), can be done on secondary storage with large merge factors, and so the analysis gives large-base logarithms. Fortunately also we seldom do joins that require the n^2 worst case. And selects can almost always be done in sublinear time (logarithmic or constant expected) because of special needs.

Much of the cost of running programs containing the relational operators of this section can be controlled either by the programmer or by the implementor. As we say above, this tutorial is at a level of abstraction which does not consider implementation. So we will occasionally say what the programmer can do for faster programs.

3 Overview of Spatial Data Processing

This tutorial is intended for readers from geospatial disciplines who want to know how to use databases in their fields. We start by reviewing geographical information systems, not to teach the reader heir trade, but to extract those aspects that can be helped by database and secondary storage technology.

In order to demonstrate that purely database constructs, not augmented for spatial data in any way, can successfully execute all spatial operations, we start with the spatial manipulations in GIS. It is not a perversion to wish to show this. The central difficulty with modern GIS is the problem of moving from spatial manipulation to analysis, aggregation, and integration with non-spatial fields. Since database constructs already do these things, as we shall also show, it is appropriate to integrate by having the database language do everything. So the second aspect of GIS we consider here is the analytical.

Before we look at these two aspects of GIS, we mention what we leave out, as beyond the scope of this tutorial. We do not discuss the graphical operations of drawing or editing nodes, arcs, polygons, or application-oriented geospatial features. This also leaves out constraint-based editing operations, such as snapping nodes to surfaces. We do not consider display management, such as zoom, pan, clip, or the presentation of layers. We ignore issues of data input to and output from the system. With these omissions, our database system provides the underlying engine for spatial and non-spatial data processing, but the graphical front end is missing. A hybrid of database for the processing and GIS for user interface can be built [48], and, for the time being, this might be the way to achieve full integration.

For the meanings we use for terms such as "arc", "node", etc., please see Appendix B.1. Terminology varies within the spatial communities, so we spell out our own usage.

3.1 Offerings from Commercial GIS

Five categories of spatial operation seem to characterize basic GIS: closeness, buffering, overlaps, containment, and operations on point-sets.

A *closeness* operator might compare two arcs or two polygons and return only those sides that are closest to each other.

A *buffering* operator is also concerned with distance, but puts a buffer of a certain width around a feature. This may be needed, for example, to designate a protected zone around a well, stream, or lake, or inside a parcel of land to limit construction near the neighbours. (Such an internally-directed buffer is called a setback. In robotics, buffers may be used to reshape the geometry of the environment so the robot, moving through it, may be considered a point).

An *overlaps* operator compares two features and returns sides from those features: if the features are arc or polygon, the sides are those that touch; if one feature is a point and the other an arc or polygon, the overlaps operator might return the sides of the latter that intercept perpendiculars from the point.

One operand of a *containment* operator is a polygon, and it returns those parts of the other that are strictly inside the polygon: an arc if the operand is an arc, or a set of points if it is a set of points.

Point-set operators perform set operations on the points that make up two features: arcs and polygons can have unions, intersections, set differences (sometimes called "erase", because points belonging to the second are erased from the first), and reconfigurations of one of the operands according to the other. Polygons are tricky in point-set operations, because the result may easily be more than one polygon. Hence, the operation on A and B that would trivially return A, if A and B were two sets, is not trivial if A and B are polygons. This is the reconfiguration mentioned last, above. We call it a "left join", although "identity" has misleadingly been used in G.I.S. We can correspondingly call union, intersection, and difference "union join", "intersection join", and "difference join" of polygons (or arcs), respectively, to emphasize the geometrical structure of the result point-set.

Figure 1 illustrates point-set operations on two "M"-shaped polygons. Up to fourteen polygons result, with the following structure. (Note that each component may be returned as a separate polygon, in which case the components listed below are the polygons returned; or else the result can be a minimum number of polygons. In the example, for the latter case, the difference join would then return five polygons, the intersection join four polygons, the union join one polygon with a hole, and the others one polygon each.)

Point-set operator	Resulting components
union join	1 .. 14
intersection join	1, 2, 3, 4
difference join	5, 6, 7, 8, 9
left join	1 .. 9
right join	1 .. 4, 10 .. 14

3.2 Beyond Commercial Systems

The above discussion is drawn from the commercial G.I.S.s ARC/INFO [33], ArcView [34], and MapInfo [35]. Since our objective is to allow programmers to go well beyond the capabilities of such systems, we may not stop here. We should look at operations that might be desired for geospatial information and even at geometrical operations that are possible but may not yet have been used for geospatial purposes. For the former, we can take a framework from Chrisman's book [13], with pointers to the G.I.S. literature [39], and for the latter we look at results from computational geometry. All these must be adapted to programming for secondary storage, which we do in later sections.

The five families of operations considered above are purely spatial. Maps, however, are not pure geometry but spatial representations for essentially nonspatial data: roads have classifications, counties have populations, fields have crops. These non-spatial properties are called "attributes" in the geospatial literature, but this term has a more general meaning in databases. There are rich possibilities for confusion here as we merge the two fields, so we avoid "attribute" altogether in this tutorial. We translate the geospatial use explicitly as "non-spatial properties", or just as "values" for short, and the database use as "fields" (and depend on context to distinguish this from the agricultural term or other uses).

The first thing we can do with values (non-spatial properties) is operate on them without reference to their spatial associations. We call these operations *scalar* operations, and an

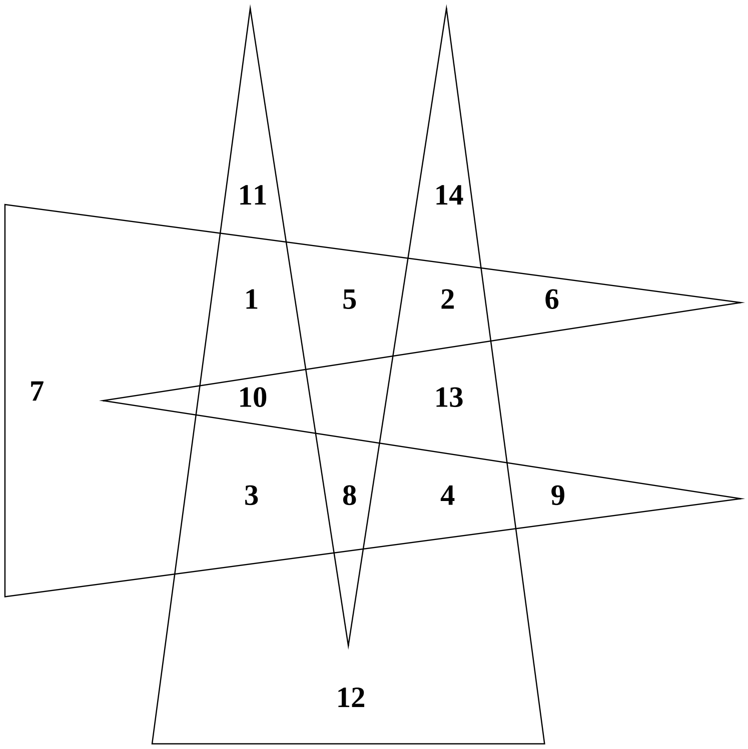


Figure 1: Polygons resulting from point-set operations on two polygons

example might be the combination of windspeed with temperature to produce a windchill factor.

We can also *aggregate* values, and this usually involves spatial relationships. An example is the average rainfall in a county, aggregated from measurements at instrumented points throughout the area.

Finally, we can combine values from two or more different spatial distributions. For example, to find hemlock forests on acidic soil, we might combine a forestry map with a soils map and select the combination from the result. This is called an *overlay* operation, and requires the point-set intersection of polygons together with a scalar combination of non-spatial properties. (It also requires *registration* of the two maps and unifying their projections so that the features coincide correctly. If the two are already registered and parts of the same map, they are called *layers*.)

A second elaboration on the G.I.S. operations in section 3.1 is a more general treatment of distance. We can seek (nearest) neighbours, we can analyse localities in various ways (usually by aggregation), and we can use Voronoi diagrams (or their dual, Delaunay triangulations) to characterize neighbourhoods. Distance may be used to permit "fuzzy tolerance" of boundaries when high-resolution maps are being registered.

Third, our initial discussion makes no mention of surfaces, of their various representations (point-heights, digital elevation matrices (DEM), contours, or triangulated irregular networks (TIN)), or of operations to transform among these representations. Surfaces, which are strictly two-dimensional in most geospatial abstractions, have components in the third dimension, such as heights and gradients, and have corresponding topologies of peaks, ridges, river courses, and watersheds. This third dimension need not be only height, but may also be a measure such as population density, or even a multi-component quantity such as air or water currents (which have both a magnitude and a direction at every point). More generally, surfaces can be strictly three-dimensional, as in overhanging cliffs, and have non-simple topologies, as in mountains with tunnels.

A fourth set of capabilities not addressed before have to do with extracting highly non-local information, such as the shortest route between two points, the drainage flow on a surface, or the "viewshed" of places from which a proposed blot on the landscape can be seen.

Finally, we have not distinguished raster from vector representations of maps. Indeed, by discussing polygons and arcs, we have inclined towards the latter. From a geometric point of view, raster representations are a special case of vector, consisting of polygons which are all rectangles (or squares), and which subdivide the map space into a regular grid. Some kinds of processing are simplified with raster representations because of this specialization and regularity. When we come to register two raster grids of different scales or orientations, we are faced with their full polygonal nature. A "choropleth" map (Greek: $\chi\omega\rho\omega\sigma$, *place*, $\pi\lambda\eta\theta\rho\omega\nu$, *a measure, which is either linear or areal*) is the hybrid of the two: it consists of polygons which are already defined for some purpose, with associated values which may be the same in neighbouring polygons. With rectangular polygons in a grid, this is exactly what a raster representation is.

Some processing which has been mainly limited to raster representations can be available for vector implementations given the above perspective. Raster *neighbourhoods* and nearest-neighbours extend readily to general polygon neighbourhoods. *Filters*, used to aggregate values in a raster neighbourhood, can likewise be extended to choropleth polygons.

3.3 Computational Geometry

Many of the above operations have been investigated generally by the computational geometry community, who have been able to specify what constitutes the optimal algorithm (and associated data structure) for many of these problems, and who have in many cases produced these optimal algorithms.

Here is a representative sample of such problems, together with the running time and storage space requirements established for the best algorithms. In keeping with the cultural home of computational geometry in computer science, these costs are given as "complexities", using order notation, in which, say, $\mathcal{O}(n \log n)$ means that for some big enough size, n, of the problem being solved, the cost never exceeds some constant times $n \log n$. For practical applications, especially on secondary storage, the constant itself is very important, and the order notation of complexity theory is excessively abstract. So these results can only be guidelines to geospatial implementations. In the table, n is the size of the input, usually number of edges or number of vertices (they are the same for planar polygons), k is the size of the result, and d is the number of dimensions (2 by default): we are not more specific here. The sources for this table are [20, 47], and a review by Godfried Toussaint of this section. ("Best worst-X" means "worst-case X for the best known algorithm".)

Problem	Best worst-time	Best worst-space
Lines meet	$\mathcal{O}(k + n \log n)$	$\mathcal{O}(n)$
Point-set	$\mathcal{O}((k + n) \log n)$	$\mathcal{O}(n)$
Triangulate polygon	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Voronoi diagram	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Delaunay triangulation	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Gabriel graph	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Relative neighbour graph	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Minimal spanning tree	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Convex hull	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Point-in-polygon	$\mathcal{O}(n)$	$\mathcal{O}(n)$
All nearest neighbour (any d)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Orthogonal range query	$\mathcal{O}(k + \log^d n)$	$\mathcal{O}(n(\log n / \log \log n)^{d-1})$
Polygon skeleton	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Visibility in polygon	$\mathcal{O}(n)$	$\mathcal{O}(n)$

The lines-meet problem finds intersections of each line pair in a set of edges, or between the edges of two sets. Naively, it would be quadratic in cost, $\mathcal{O}(n^2)$, as a result of checking each edge for intersection with every other edge (n is the number of edges), but "plane-sweep" ordering reduces it to a sorting problem. The point-set problems of section 3.1 can use lines-meet as a basis for polygon intersection, union or difference.

The next seven problems are all related. The Voronoi diagram of a set of vertices creates a polygon around each vertex inside which all points are closer to it than to any other. The Delaunay triangulation is the topological dual, in the sense that the Voronoi polygons become vertices (in fact, the original vertices), and the vertices of the Voronoi polygons (generated) become faces. These are, mostly, triangles, and the triangles have the property that their internal angles are larger than for any other set of triangles connecting the original vertices. The Gabriel graph, relative neighbour graph, minimal spanning tree, and convex hull are all subgraphs of the Delaunay triangulation. The edges of the Gabriel graph connect only those vertex pairs on opposite ends of a diameter of a circle which excludes all other vertices. The relative neighbour graph is a subgraph of the Gabriel graph, eliminating edges between vertex pairs if some other vertex is in the intersection of the two circles with *radius* equal to their separation. The minimal spanning tree removes more edges so that the graph is a tree connecting all vertices, and no other such tree has a smaller sum of edge lengths. It is a central construct for clustering algorithms: to form k clusters, just remove the $k-1$ longest edges from the minimal spanning tree. The convex hull is the polygon that is the subgraph of the Delaunay triangulation with all interior vertices disconnected: clearly it is a convex polygon. Finally, an arbitrary polygon can be subdivided into triangles, also with $\mathcal{O}(n \log n)$ cost; some special polygons of a class (which includes most contour maps) can be triangulated in almost linear time.

The point-in-polygon problem determines if an arbitrary point is inside a given polygon. It extends to determine which of a set of polygons, or of the polygonal faces of a subdivision, the point is in, and is part of the containment problems considered in section 3.1. (It can be done faster if the polygons have been preprocessed into a suitable data structure.)

The all-nearest-neighbour and orthogonal range query problems are shown for any number of dimensions, not just two, because they are important aspects of multidimensional analysis, which we consider in section 7. The results from computational geometry show that dimensionality is not a curse in these cases, at least for algorithms in primary memory. The all-nearest-neighbour problem is to find the vertex nearest to every vertex in a set. To find the nearest neighbour of one vertex is faster, but at the expense of $\mathcal{O}(n \log n)$ preprocessing time to construct a suitable data structure. The orthogonal range query, as well as of interest in computational geometry, is central to much database querying. It involves specifying a range of values along each axis of a multidimensional space, and searching for vertices, or database records, in the conjunction of these ranges.

The polygon skeleton problem is to find the medial axis of a polygon. The medial axis plays the role in a polygon that the Voronoi diagram plays in a set of vertices: it separates the interior of the polygon into regions, each of whose points are closer to a given element of the boundary than to any other.

The visibility polygon of P from a point within a polygon, P, is the subset of points in P that are visible from that point. This is related to the viewshed problem, above, and to hidden-line problems in graphics.

All of these problems are of less than quadratic complexity, and so are likely not intractable for large amounts of data requiring secondary storage. With a file of a billion records, say, making a billion passes should be considered an intractable problem, and this is what quadratic complexity would mean. In most computer science, including computational geometry, intractibility starts when the complexity exceeds polynomial, but secondary storage is much more exigent.

3.4 Spatial Mathematics

Computational geometry is a very recent discipline. Geometry, on the other hand, was the first branch of mathematics to be formalized (although only about half the age of the earliest mathematical thinking) [37]. Mathematics has been concerned about space for almost five millennia. Many insights of these centuries have been absorbed into the computational disciplines we have discussed, but much is missing, especially since the advent of calculus and the introduction of field theories in physics. We mention one intriguing and useful result out of all this, Stokes' theorem.

Stokes (1854) (actually, Kelvin, four years earlier) relates the integral, over an area, of some differentiable functions, $P(x,y), Q(x,y)$, to another integral, around the closed contour bounding the area, involving these functions. A simple version is

$$\int\int_A \left(\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}\right) dx\, dy = \oint (P\, dx + Q\, dy)$$

With polygons, the contour integral becomes a sum of simple expressions of the coordinates of the vertices, depending on the functions P and Q. For example, if $P = 0$ and $Q = x$, the contour integral is the sum around the vertices

$$\sum \frac{1}{2}(x_1 y_2 - x_2 y_1)$$

where (x_1, y_1) is one endpoint of an edge and (x_2, y_2) is the other. By convention, the sum is taken in a counterclockwise direction around the contour: this convention stems from the prior convention that positive angles are counterclockwise.

The areal integral that this sum computes is just the area. This gives us a computational algorithm for the area of an arbitrary polygon, and was the basis for the Victorian *planimeter*, a device with a wheel that can be run around the border of a feature on a map, and, given suitable scale factors, directly tells us the area.

Areas are not all that Stokes' theorem can derive. Other choices of P and Q yield the x- and y-coordinates of the centroid of a polygon, or can give the average slope of the polygon, if each of its vertices also has a height above sea-level. All of these calculations are actually used as applications of Stokes' theorem later in this tutorial, in section 6.2.

The nineteenth century produced similar theorems relating volume integrals to integrals over the surface bounding the volume in three dimensions, and, with general relativity theory, the twentieth century's triumph of geometry, in four.

3.5 Spatial Predicates

Although it is the purpose of this tutorial to show how to build spatial database facilities which go beyond queries, querying databases and GISs has been a major activity, historically and at present. Queries involve predicates. They have either the form "<*predicate*>?", to which the answer is **true** or **false**, or the form "for which items is <*predicate*> true?".

Predicates are immensely varied, and facilities for expressing and evaluating them must be very general. They can be simple, as "$x = 2$?", or elaborate, as "find departments that sell at least two thirds of the items supplied to the departments on the same floor". To appreciate how varied even the subset of spatial predicates is, reflect that almost any practical relationship which we use language to describe involves a spatial, or a spatio-temporal predicate. (Start with "Look, Jane, look! See Dick run!" Jane looking at Dick involves direction, distance, and the absence of opaque obstacles. Dick running involves a succession of positions in a certain velocity range, a gait, which is a particular succession of body positions, and so on. Try writing a program to determine if Dick is running and if Jane is looking at him!)

Processing spatial predicates often requires extensive computation. For example, to determine which regions in a map overlap (e.g., the regions where soil type is acid, and the regions where cedars grow), or to determine the amount of one region that overlaps another, requires executing line-intersection and point-in-polygon algorithms for an entire map. If we are doing data mining, whose objective is to discover predicates that characterize large amounts of data, we will not be able to afford all the processing. Even a user posing a single query, involving a single predicate, may have to wait too long to get a precise answer when a rough answer might do. So for spatial predicates in particular we need approximations to speed up processing, at the expense of accuracy. It is often better to get a rough answer, then refine it at need with more precise predicates.

We will look in this section at hierarchies of spatial predicates, based on some of the literature on geospatial and image predicates. The role of a hierarchy is to provide successive approximations. We start by inspecting the different levels of abstraction offered by different geometries, because these form a hierarchy and because abstraction is a form of approximation.

Euclid's geometry is metric. So are the two non-flat variants, for positively curved space (spheres, etc.) and for negatively curved space (rotated hyperbolas, parabolas, etc.). These geometries depend on notions of distance and direction, and contain results on equality of sides of triangles, how to bisect angles, and so on. The corresponding trigonometries are also metric, giving specialized results for right-angled triangles. These geometries and trigonometries are the basis for most spatial data, notably in their analytical, or coordinate, forms.

In the Renaissance, when painters took note of perspective, Alberti wondered what was common between two different perspectives of the same scene. This was the start of projective geometry, an abstraction from metric geometry, because the quantitative notions of distance and direction play no role. Intersections of lines and surfaces are still meaningful, and so, in most cases, is a notion of order, as in the relative positions of points in a line.

The final abstraction, topology, gets rid of order and keeps only intersections. So, in a sense, topology is the most abstract representation of space, and the basis for the roughest approximation for predicates about the space. (We will see in section 4 that it is important to store topological information explicitly in addition to the full metric representation of space, but this is only because numerical imprecision destroys our knowledge of topological relationships, which perfect precision would not.)

A caveat to this metric-projective-topological hierarchy is that it does not cope as well as it might with extended spatial structure, that is to say, shape. So another important approximation technique is to simplify shapes. In the Cartesian coordinate system of analytic Euclidean geometry, the minimum bounding rectangle (*MBR*) is such a simplification: this is given by $2d$ coordinates, where for each of d dimensions, the first coordinate is the smallest in that dimension and the second is the largest. (Note that this definition applies to any coordinate system in any number of dimensions: the MBR is a rectangle only in the two-dimensional Cartesian case.)

(The *skeleton* (see section 6.4) of a polygon is also a simplification, although, in fact, equivalent: the skeleton can be derived from the polygon, and the polygon in turn derived from the skeleton plus radius data at each skeletal vertex.)

An alternative way of thinking is slightly different. We can group abstractions (approximations) according to which spatial transformations leave them invariant, that is, do not affect them. For example, an approximation which abstracts over distance is scale-invariant: it will not be affected by a change in scale (or units) of the coordinate axes. An approximation which abstracts over direction will not be affected by rotations or reflections. Some approximations are dimension-invariant. The MBR approximation is, in a sense, shape-invariant. Metric geometries are invariant under none of these transformations, and are the least approximate. Topology is invariant under all, and is the most approximate. Since there are at least four transformations, listed above, we can see that a hierarchy of approximations need not be a simple ordering of approximations, or even a tree.

In the next three subsections, we discuss three approximations from the literature, in historical order. This order happens also to be the historical order in which the applicable geometries were discovered. In section 3.5.4, we discover the links among these approximations and integrate them into a hierarchy. The general idea of treating spatial predicates in this way is from Marchand et al. [40].

3.5.1 Metric geometry: Intervals

In an investigation of possible relationships among temporal intervals, James Allen [3] identified thirteen pairwise relationships, and went on to combine them transitively to give 97 triples (and 72 other transitive combinations of varying ambiguity). In terms of our geometric abstractions, Allen's work is basically projective, but it can easily be refined to be metric. We explain the pairwise relationships using a different representation from Allen's, one which can extend his ideas to metric geometry.

Temporal data is, of course, one-dimensional, but Allen's work can easily be extended to MBRs in any number of dimensions by taking the Cartesian product of the one-dimensional relationships. We start with Allen's one-dimensional intervals (first column of figure 2). We see that they describe the possible alignments of the interiors and the boundaries of two intervals, from disjointness to contiguity through four degrees of overlap to coincidence, and back again with the roles of the two intervals exchanged. This enables us to label them with seven numbers plus complements of the first six numbers (second column of figure 2).

The third column of figure 2 is a more basic representation of the Allen relationships, indicating whether each of four distances is negative, zero, or positive. These are the distances

Allen	Label	Pattern	1dString	9Inter		Venn
1 2		12 12 12 12 ss se es ee		i b e 2 i b e 1		g l r o
	1	+ + + + =80	1 < 2	0 0 1 0 0 1 1 1 1	− − 1 − − 0 1 0 1	1 1 1 0
	2	+ + 0 + =77	1 < 2	0 0 1 0 1 1 1 1 1	− − 1 − 0 0 1 0 1	0 1 1 0
	3	+ + − + =74	1 < 12 < 2	1 1 1 1 0 1 1 1 1	1 0 1 0 − 0 1 0 1	0 1 1 1
	4	+ + − 0 =73	1 < 12	1 1 1 0 1 1 0 0 1	1 0 1 − 0 0 − − 1	0 1 0 1
	5	+ + − − =72	1 < 12 < 1	1 1 1 0 0 1 0 0 1	1 0 1 − − 0 − − 1	= 4
	6	0 + − + =47	12 < 2	1 0 0 1 1 0 1 1 1	1 − − 0 0 − 1 0 1	0 0 1 1
	7,7’	0 + − 0 =46	12	1 0 0 0 1 0 0 0 1	1 − − − 0 − − − 1	0 0 0 1
	6’	0 + − − =45	12 < 1	= 4	= 4	= 4
	5’	− + − + =20	2 < 12 < 2	1 0 0 1 0 0 1 1 1	1 − − 0 − − 1 0 1	= 6
	4’	− + − 0 =19	2 < 12	= 6	= 6	= 6
	3’	− + − − =18	2 < 12 < 1	= 3	= 3	= 3
	2’	− 0 − − =3	2 < 1	= 2	= 2	= 2
	1’	− − − − =0	2 < 1	= 1	= 1	= 1

Figure 2: One-dimensional minimum bounding rectangles

between the endpoints of the two intervals.

- `ss` stands for the distance from the start point of interval 1 to the start point of interval 2 (the first digit refers to the first interval, the second digit to the second interval; `s` means "start", and we see next that `e` means "end");
- `se` stands for the distance from the start point of interval 1 to the end point of interval 2;
- `es` stands for the distance from the end point of interval 1 to the start point of interval 2; and
- `ee` stands for the distance from the end point of interval 1 to the end point of interval 2.

The integer assigned to each of these patterns of four distances (after the = sign in the third column of figure 2) is the value of the pattern treated as a ternary number, with `-` being 0, `0` being 1, and `+` being 2. No other of the conceivable 81 values is possible, due to the constraints that interval starts are before interval ends and that interval lengths are positive. Note that forming complements of intervals becomes, in this representation, the operation of inverting the signs and swapping `se` with `es`.

This representation shows clearly how to extend the thirteen Allen relationships into an infinite number of fully metric relationships: just replace `+` and `-` by the appropriate positive and negative distances. Thus the Allen relationships in one dimension are an approximation to metric relationships which replace an infinite number of possibilities by just thirteen. The approximation is easily calculated from the distances by determining their sign, or if zero. The resulting pattern can be reduced to the ternary value, and then to the numeric label, if this is desired.

(The remaining columns of figure 2 will be discussed in section 3.5.3.)

In two dimensions, there are 13 possible interval relationships for the x coordinate and 13 for the y coordinate, giving 169 combined relationships. The entities that relate to each other in these ways are evidently minimum bounding rectangles, with the x-intervals giving the smallest and largest x-coordinate of the region being considered, and the y-intervals similarly bracketing the y-coordinates.

We can see from the table below that approximate directions can be derived from the two-dimensional Allen relationships. If east has higher x-coordinates than west, and north has higher y-coordinates than south, then we can say that for x-coordinates with Allen labels `1, 2, 3, 4` interval 2 is east of interval 1, for x-coordinates with Allen labels `1', 2', 3', 4'` interval 2 is west of interval 1, and for the rest the intervals do not essentially differ in the east-west direction. The same goes for y-coordinates, with south replacing west and north replacing east. Combining these considerations, we get the following nine compass directions from the 169 Allen relationships in two dimensions.

$y \mid x$	`1 2 3 4`	`5 6 7 6' 5'`	`4' 3' 2' 1'`
`1 2 3 4`	NE	N	NW
`5 6 7 6' 5'`	E		W
`1' 2' 3' 4'`	SE	S	SW

169 2-D Allen relationships ⇒ 9 compass directions (MBR 2 *dir* MBR 1)

In three dimensions, there are 2197 Allen relationships between pairs of MBRs. These can in turn be reduced to 27 directions, if we wish. Note that one of these dimensions could be temporal and the other two spatial. For a full spatio-temporal accounting (Euclidean, not Einsteinian, though), we could use the 28561 four-dimensional Allen relationships.

We will call the Allen relationships between MBRs in any number of dimensions *kdAllen* relationships. The refinement to metric relationships (where all `+` and `-` signs are replaced by numbers) we call *kdAllen*(*metric*) relationships, if we need to make the distinction.

3.5.2 Projective geometry: Order

Another work not directly related to geospatial data is the paper by Chang, Shi, and Yan [11] on image indexing for searching and matching pictures. This is an example of a projective approximation: the idea is to represent images by a form of projection on their x and y axes. For example, a picture consisting of the items `a, b, c,` and `d` in the following arrangement

```
d
  b c
a a
```

has the x-"projection" `ad<ab<c` and the y-"projection" `aa<bc<d`. This is because, as we scan from left to right (x) we encounter items `a` and `d` together, then `a` and `b`, then `c`, and similarly for scanning from bottom to top (y). Thus, the two-dimensional picture is represented by two strings, and picture searching can be implemented by string searching. They call the representation "2-D Strings".

Note that an item can be extended, as `a` is, above: its tag is simply repeated in the strings. Chang et al. develop this representation to guarantee invertibility (the picture can be reconstructed from the strings), and go on to show how two pictures can be matched (for instance, if one of the two is a query, and the other is a picture from a database being tested for match). We do not need these elaborations for this tutorial.

Two useful concepts in this representation are rank and distance. The *rank* of a symbol in a string is the number of $<$ signs which occur before it in the string, and the *distance* between two different symbols is the absolute difference between their ranks. Because symbols of extended items must repeat in the strings, it is useful to distinguish between *min_rank* and *max_rank* on one hand, and between *min_dist* and *max_dist* on the other. Finally, for two dimensions, we distinguish between x and y applications of these notions, leading to eight detailed concepts: *x_min_rank, x_max_rank, x_min_dist, x_max_dist, y_min_rank, y_max_rank, y_min_dist,* and *y_max_dist.*

We can use these to derive the ideas of compass direction, and of distance in the non_metric sense of nearest_neighbours or k_nearest_neighbours. For example (assuming no holes in the regions),

$$
\begin{array}{lcl}
p\mathrm{W}q & == & \text{x\_max\_rank}(p) < \text{x\_min\_rank}(q)\& \\
 & & \text{y\_min\_rank}(p) = \text{y\_min\_rank}(q)\& \\
 & & \text{y\_max\_rank}(p) = \text{y\_max\_rank}(q) \\
p\mathrm{SW}q & == & \text{x\_max\_rank}(p) < \text{x\_min\_rank}(q)\& \\
 & & \text{y\_max\_rank}(p) < \text{y\_min\_rank}(q) \\
p = q & == & \text{x\_min\_dist}(p,q) = \text{x\_max\_dist}(p,q) = \\
 & & \text{y\_min\_dist}(p,q) = \text{y\_max\_dist}(p,q) = 0 \\
p\mathrm{nn}q & == & \text{x\_min\_dist}(p,q) \leq\ 1\ \&\text{y\_min\_dist}(p,q) \leq\ 1\ \&p \neq q
\end{array}
$$

Note that 2_D Strings can represent multiple items, not just pairs of items. It is this that allows it to support the idea of k-nearest-neighbours. Pairwise relationships, such as kdAllen, cannot. (Although, as we mentioned, Allen investigated ternary relationships as a result of transitivity, this does not help, because we would need an arbitrary number of items.)

Since there is one string per dimension, this approximation generalizes to any number of dimensions, and we call it the *kdString* approximation. In one dimension, we can relate it to the Allen relationships for intervals (MBRs): The thirteen Allen relationships generate eleven 1dStrings, because the kdString method does not recognize whether the size of the gap in cases `1` and `1'` is zero (cases `2` and `2'`, respectively) or not. These eleven 1dStrings are given in the fourth column of figure 2.

Note that kdStrings are not limited to MBRs, although they are limited to approximating shapes as a rectilinear grid of pixels. Thus the hierarchical relationship between kdStrings and kdAllen is that kdAllen is almost a special case of kdStrings, in the MBR approximation, but not quite, because kdStrings fail to distinguish separateness from contiguity. This reflects the scale-invariance of kdStrings, hence their projective nature. For higher dimensional MBRs, there are 121 2dStrings, 1331 3dStrings, 14641 4dStrings, and so on.

Our 2dString examples of compass directions do not always agree with our 2dAllen definitions, and are disputable: it would be useful to explore alternatives.

3.5.3 Topology: Intersections

Further abstractions of these spatial relationships were made for geospatial data by Egenhofer and his colleagues (e.g., [23]) and by Clementini and Di Felice and co-workers (e.g., [14]). This work is almost purely topological, abstracting over distance and direction, but not over dimension. We will consider a subset of Egenhofer's approach which is independent of dimension, and the corresponding subset of Clementini and Di Felice's work, which extends this by explicitly giving the dimension of the relationship between items.

The approaches consider each geo-spatial item to have an interior, i, a boundary, b, and an exterior, e, and they consider the binary relationships between pairs of items determined by which of these components intersects which other. This results in a 3×3 matrix of, in Egenhofer's case, booleans, and of, in Clementini and Di Felice's case, non-negative integers giving the dimensionality of any intersection and one other value indicating the absence of intersection. Clearly, Egenhofer's is a special case of the other, derived by mapping any dimension to **`true`** and the absence symbol to **false**. We call both methods *9Inter*, for nine-intersection, and, if needed, we distinguish the latter as *9Inter*(*dim*).

The fifth column of figure 2 shows both types of matrix for the special case of one-dimensional MBRs. The thirteen Allen relationships reduce to eight topological relationships in both cases. So we see that the topological relationship is a special case of the Allen relationship, except, of course, that the topological relationship is not limited to MBRs. If we restrict the Egenhofer representation in higher dimensions to *solids*, that is items with the full dimensionality of the embedding space, and with no holes, there are always only these same eight matrices. We will discuss numbers of *9Inter*(*dim*) matrices in a moment, after explaining the examples in figure 2.

Consider case `4` in figure 2. The interior of interval 1 has a one-dimensional intersection with the interior of interval 2, so the i,i entry of the *9Inter*(*dim*) matrix is `1` (and the i,i entry of the Egenhofer matrix is **`true`**, shown as `1`). The interior of interval 1 has a zero-dimensional intersection with the boundary of interval 2, so the i,b entries are `0` and **`true`**, respectively. The interior of interval 1 has a one-dimensional intersection with the exterior of interval 2, so the i,e entries are `1` and **`true`**, respectively. The boundary of interval 1 has no intersection with the interior of interval 2, so the b,i entries are `-` (the no-intersection symbol: it could be a negative integer) and **false**, respectively. And so on.

Case `6` has exactly the same matrices, although Allen distinguishes them.

1. symmetrical (under exchange of solids)														
	1D			2D			3D			dD				#
disjoint	-	-	1	-	-	2	-	-	3	-	-	d		1
	-	-	0	-	-	1	-	-	2	-	-	$d-1$		
	1	0	1	2	1	2	3	2	3	d	$d-1$	d		
touching	-	-	1	-	-	2	-	-	3	-	-	d		d
	-	0	0	-	k	1	-	k	2	-	k	$d-1$	$k = 0..d-1$	
	1	0	1	2	1	2	3	2	3	d	$d-1$	d		
overlapping	1	0	1	2	1	2	3	2	3	d	$d-1$	d		1|2
	0	-	0	1	k	1	2	k	2	$d-1$	k	$d-1$	$k = d-2, d-1$	
	1	0	1	2	1	2	3	2	3	d	$d-1$	d		
equal	1	-	-	2	-	-	3	-	-	d	-	-		1
	-	0	-	-	1	-	-	2	-	-	$d-1$	-		
	-	-	1	-	-	2	-	-	3	-	-	d		
2. unsymmetrical (transposes not shown)														
	1D			2D			3D			dD				#
contained	1	0	1	2	1	2	3	2	3	d	$d-1$	d		2
	-	-	0	-	-	1	-	-	2	-	-	$d-1$		
	-	-	1	-	-	2	-	-	3	-	-	d		
cont., touch	1	0	1	2	1	2	3	2	3	d	$d-1$	d		$2d$
	-	0	0	-	k	1	-	k	2	-	k	$d-1$	$k = 0..d-1$	
	-	-	1	-	-	2	-	-	3	-	-	d		

Figure 3: The 9-Intersection Approximation for d Dimensions

The matrix for case `4` is not symmetric: its transpose appears in cases `6'` and `4'`. In general, swapping the intervals transposes the matrix. Four of the matrices are symmetrical, in each representation, and two are not. Including these and their transposes gives all eight matrices in each representation.

In higher dimensions, the Egenhofer matrices for relationships between solids are the same eight, but the *9Inter*(*dim*) matrices increase in number to reflect the increased possibilities for intersections having different dimensions, but can still be considered in eight groups. Figure 3 is a table showing these groups for the *9Inter*(*dim*) matrices for one, two, three, and d dimensions.

It is worth noting that because the Allen relationships are between MBRs, in two or more dimensions there are *9Inter*(*dim*) relationships that Allen cannot model: these are the relationships in which one item contains the other, the contained item is touching the outer one, and the dimensionality of the contact is less than $d-1$. In two dimensions, for a contained item to touch the containing item (which may be an MBR) at only one point (0 dimensions), the inner one must, if it is rectangular, be tilted, and so cannot be an MBR.

The 9Inter relationships can be extended to ternary by using $3 \times 3 \times 3$ "matrices", thus giving 27 intersections, and so on for n-ary relationships. As with the Allen extensions, this is not practical for modelling multiple items.

Thus, k-nearest-neighbour queries cannot be posed in the 9Inter approximation, and, of course, directions are excluded by the topological nature of the approximation.

Investigations made by both Egenhofer and Clementini & Di Felice and their collaborators include topological relationships between items of lower dimensionality than the embedding space. For instance, Egenhofer and Herring [23] give 20 region-line relationships, where the line may be simple or may have branches, and 57 line-line relationships. (The 85 relationships that include these and the 8 region-region produce only 61 distinct matrices, so the dimensionality of the components must also be represented in the approximation.) Of course, in the approximation of MBRs, these all reduce to the region-region relationships discussed above.

One remaining column in figure 2 has not been discussed. This is an even coarser approximation than Egenhofer's, which may be called the *Venn* topological approximation, from the common way of representing set relationships. The four headings stand for *gap, left difference, right difference,* and *overlap*, and the entries below them indicate whether they exist for the relationship (with `1` meaning **true** and `0` meaning **false**. Here, by "*left*" and "*right*" we mean not geometrical left and right, but the formal left and right according to which interval `1` is "left" and interval `2` is "right". Thus, "left difference" means `1`−`2` and "right difference" means `2`−`1`. The Venn approximation distinguishes only six of Allen's relationships and is independent of shape and dimension, and, of course, distance and direction. (These could be written as 2×2 matrices, of which four are symmetric and two not, but transposes of each other.)

3.5.4 A hierarchy of approximations

Each of the above representations allows us to approximate a certain set of fully precise spatial predicates. Since it may be useful to approach a predicate by successive refinement, certainly in data mining applications, and even for normal queries, we would like to know how these representations are related to one another hierarchically. If there is a simple hierarchy, we can start by testing a predicate at the most approximate level, and, if the result is promising, refine it to more and more precise levels until we finally test the exact predicate. (For data mining, where very many potential predicates may need to be investigated, this will eliminate irrelevant ones early on without incurring a great burden of computation.)

From the foregoing discussion, it is apparent that kdAllen, kdString, and 9Inter do not fit into a simple hierarchy. But we can provide a partial ordering, and we show it in figure 4. Here, the decreasing refinement from metric to topological is laid out horizontally, left to right, and the decreasing refinement from full precision to most approximate is laid out vertically, from top to bottom. There is one "almost" refinement, in that kdString is not quite a refinement of kdAllen, because it fails to distinguish separateness from contiguity.

The discussions above should amplify this depiction of the hierarchy of approximations. The only approximation shown but not discussed already is *4Inter*, a precursor to 9Inter which involves only interiors and boundaries and hence forms 2×2 matrices. Since kdAllen is restricted to MBRs, any other method shown to be a special case of kdAllen is so only when restricted to MBRs.

Given the hierarchy, we can in principle derive the more approximate predicates from the less approximate ones. For example, starting with `ss, se, es,` and `ee`, we can use their signs to calculate the ternary value that gives the 1dAllen relationship, then use the rank of this to find which of the seven labels (or their complements) applies. From here, we could find the 9Inter relationship for 1dMBRs, with or without overlap dimensionality, by lookup in a table of thirteen entries; and we could find the 1dString and Venn relationships similarly. Or, we could go directly from `ss, se, es, ee` to a metric Venn relationship in 1d by defining

$$\begin{aligned} gap &= (\texttt{es}\ \mathbf{max}\ 0) - (\texttt{se}\ \mathbf{min}\ 0) \\ ldiff &= (\texttt{ss}\ \mathbf{max}\ 0) - (\texttt{ee}\ \mathbf{min}\ 0) - gap \\ rdiff &= (\texttt{ee}\ \mathbf{max}\ 0) - (\texttt{ss}\ \mathbf{min}\ 0) - gap \\ olap &= (L1 - (\texttt{ss}\ \mathbf{max}\ 0) + (\texttt{ee}\ \mathbf{min}\ 0))\mathbf{max}\ 0 \end{aligned}$$

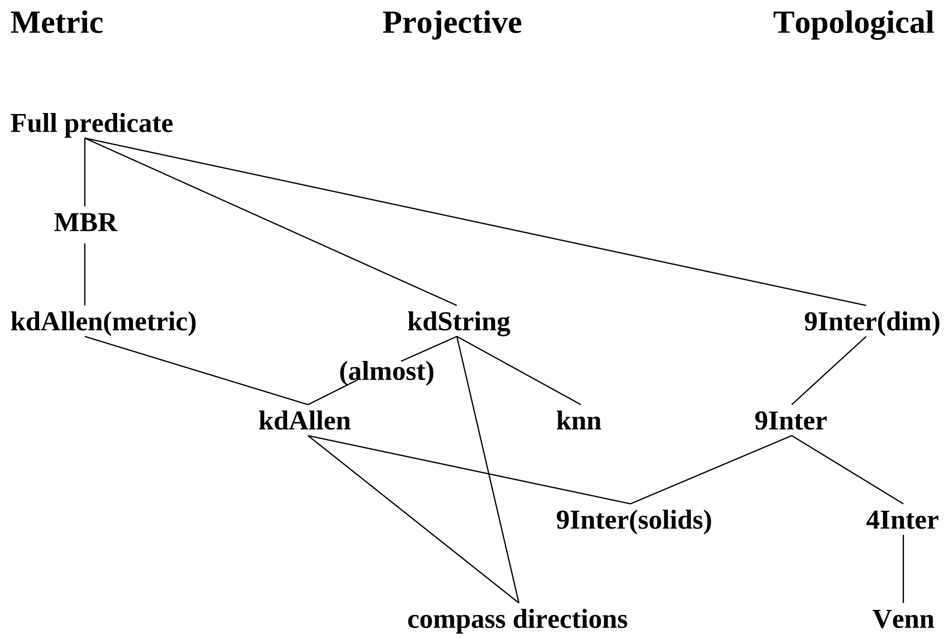


Figure 4: Hierarchy of approximations to spatial predicates

$$= (L2 - (\texttt{ee}\ \mathbf{max}\ 0) + (\texttt{ss}\ \mathbf{min}\ 0))\mathbf{max}\ 0$$

where $L1$ and $L2$ are the lengths of intervals 1 and 2, and can also be derived from `ss, se, es, ee`

$$L1 = \texttt{ss} - \texttt{es} = \texttt{se} - \texttt{ee}$$
$$L2 = \texttt{se} - \texttt{ss} = \texttt{ee} - \texttt{es}$$

When we go to higher dimensions, we can use Cartesian products for the direct calculations on MBRs, but the lookup tables become unwieldy. When we go beyond MBRs, we can no longer use Cartesian products. Thus, finding 9Inter, even for solids with arbitrary shapes in higher dimensions involves us in spatial algorithms from computational geometry such as region-region and region-line intersections. The advantage of having the approximations is offset by having to precalculate and store representations of these relationships for all pairs of items of interest in our maps. Finding kdStrings also requires spatial precomputation, but results in only one index to be stored for any map.

3.5.5 Queries and Spatial SQL

The original use, in database and secondary storage contexts, of predicates was, and continues to be, for queries. So it behooves us to discuss at least one query language for spatial predicates. Since the emphasis of this tutorial goes far beyond queries and query languages, to general geospatial algorithms and processing, we do not consider more than one query language. We pick Spatial SQL [22], because of its use of some of the above approximations (4Inter), and because of an important contribution which goes beyond the scope of this tutorial.

This contribution is the recognition that a spatial query language is inseparable from a spatial display and interaction capability, and the incorporation of this in Spatial SQL via the graphical presentation language, GPL. The simplest such interaction allows the formal query, typed from the keyboard, to refer to GUI (graphical user interface) actions to select parts of the map. This uses the PICK keyword (quite different from the **pick** operator introduced later in this tutorial) as a function in a select clause, such as WHERE *geom*=PICK.

More advanced display capabilities allow the query to specify graphical properties for its results, such as showing selected parcels in black and selected roads cross-hatched. An important display capability is to provide the context for a query: it is not adequate to display only the query results, because a viewer needs to see the setting. For example, the language allows the querier to specify that the context is parcels, buildings and roads. Given all this preparation, the query can be made for a particular street in a particular town, and the map will display the street and its context using the graphics specified. Subsequent queries can alter this map by highlighting, overlaying, etc.: a MODE operation tells what to do next.

This tutorial is concerned with the database language needed to implement a query language such as this, and more, but not with the important issues of display.

The spatial queries themselves add to conventional SQL first geometry fields and second functions and relationships involving these new types of data. The four new types of field are called spatial_*d*, where *d* is 0,1,2, or 3 dimensions. The relationships include eight 4Inter relationships. The functions are unary (length, area, boundary, etc.) or binary (distance, direction). *Geom*, above, is a field which might be of type spatial_2.

Any spatial query or processing language must include something like this, either built into the language, as in Spatial SQL, or provided as abstract data types, which we prefer, because of the immense variety of possibilities. The discussion we have just gone through of spatial predicates should alone persuade the reader that a language with all possible capabilities hard-wired into it would be immense, and yet it still may not have included related important things such as temporal relationships and functions.

3.5.6 Spatial Data Mining and Causality

Data mining, on the other hand, demands exploration of a potentially unlimited number of predicates, to see which one applies. (It will usually be more than one. If a hierarchy of related predicates is discovered, the one to be reported should be the most general: this will also be the shortest of the set.) We have introduced approximations and refinements so that this extensive search can be speeded up. But there is still an unacceptable amount of computation to be done.

It makes sense, while mining data, to limit the predicates we inspect to those that could involve causation. Causal spatial predicates certainly might involve relationships such as proximity, being downhill, downwind, downstream (or, more generally, downcurrent), or visible. Intersections and overlaps of the various kinds discussed above may arise, but are probably less important. So it might be more profitable for spatial data miners to precalculate these four from the base, metric data. Note that this data would need to include elevations, possibly temperature gradients, and current distributions for air and water.

Of these, closeness has been used extensively as a spatial data mining predicate([30] section 9.2), especially in its most general sense. This sense includes all but one of the 9-intersection relationships, and 11^d of the 13^d kdAllen relationships, because contiguity, overlap, and so on, all necessarily involve closeness. A problem with closeness is that, as a metric predicate, a threshold distance must be specified, and this will be relative to the application: a micron may be "close" in cellular biology, as opposed to a light-year in astronomy.

This brings us to distance, another notion which has various approximations and which has not been catered for by the above discussion of metric, projective, and topological approximations. It is a metric concept. In fact, a *metric* is the general name for the many varieties of distance, whose formal requirement is only that it satisfy the "triangle inequality": in a triangle of sides of lengths a, b, and c, $a + b \geq c$ is required, for any assignment of a, b, and c to the three sides of the triangle. The usual metric is crowfly, or Euclidean, $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ being the distance between two points with coordinates (x_1, y_1) and (x_2, y_2). Other metrics are possible. A useful one in the context of a regular grid of city streets is the taxi, or Manhattan, metric, $\mid x_1 - x_2 \mid + \mid y_1 - y_2 \mid$. Road distances in general cannot be reduced to a simple relationship between the coordinates of the two points, and require fairly complex geospatial processing.

Geospatial distances may often be road distances, but it is useful to approximate them with crowfly distance in the first instance, if we wish to eliminate pairs of features that are too far apart.

Of course, spatial data alone cannot capture the central ingredient of causality, which is priority in time. This suggests that spatial data mining may need to be completed by spatio-temporal data mining.

Finally, the above prescription should be taken in moderation. Two patterns in the data may have a common cause which is not revealed by the data. (An example of such a situation arises in section 6.5.3, where we do, informally, some non-causal mining on the 9-Intersection matrices in two dimensions, in order to implement them.)

3.5.7 Hierarchies of General Predicates

After all this discussion of spatial predicates, we can look briefly at two hierarchies which apply to any predicates. First, let us recall that we wish to use hierarchies to approximate predicates in such a way that analyzing the approximation can tell us something definite about the refined predicate. For example, if two features are too far apart as the crow flies, they are also too far apart by road distance.

The first such hierarchy that is applicable to predicates on any subject is the and/or hierarchy. If p **or** q is false, then p is certainly false, and, from this, p **and** q is false: this hierarchy transmits falsehood. Conversely, a truth-transmitting hierarchy goes from p **and** q to p to p **or** q.

Furthermore, we can see that **and** and **or** are the only practical means of providing such hierarchies: there are 2^{2^n} possible functions of n boolean variables, which means 16 for the two variables, p and q. The table gives them all, where f and t mean, respectively, false and true, and the names given to the sixteen functions are fairly self-evident (Dewdney [21] elaborates informatively).

p	q	$\mathbf{T}$	$\vee$	$\Leftarrow$	$\Rightarrow$	$\neg\wedge$	$\mathbf{P}$	$\mathbf{Q}$	$\equiv$	$\oplus$	$\neg\mathbf{P}$	$\neg\mathbf{Q}$	$\neg\vee$	$\neg\Leftarrow$	$\neg\Rightarrow$	$\wedge$	$\mathbf{F}$
f	f	t	f	t	t	t	f	f	f	t	t	t	t	f	f	f	f
f	t	t	t	f	t	t	f	t	t	f	f	t	f	t	f	f	f
t	f	t	t	t	f	t	t	f	t	f	t	f	f	f	t	f	f
t	t	t	t	t	t	f	t	t	f	t	f	f	f	f	f	t	f

What we are looking for in this table, say for a falsehood-transmitting hierarchy, is a function whose falsehood implies that of p or q, and another function whose falsehood is implied by that of p or q. To be useful, these functions should be symmetrical, the first should have fewer fs in its column than p or q, and the second should have more fs in its column than p or q. (The last two conditions mean that the hierarchy is actually giving an approximation, not just the same thing.) Only **or** ($\vee$) and **and** ($\wedge$) satisfy all these conditions for falsehood, and, taken in the other order, for truth.

The second kind of approximation hierarchy for predicates can be derived from approximation hierarchies on the fields that the predicates describe. For example, if we have the two-level hierarchies for time of year and direction

Year(*month*	*quarter*)	*Direction*(*angle*	*enws*)
January	1	22.5	E
February	1	67.5	NE
March	1	122.5	N
April	2	157.5	NW
May	2	202.5	W
June	2	247.5	SW
July	3	292.5	S
August	3	337.5	SE
September	3	360.0	E
October	4		
November	4		
December	4		

the predicate

quarter=`1` **and** *enws*=`"SE"`

is a falsehood-transmitting approximation to

month=`"February"` **and** *angle*=`330`.

Thus, hierarchies of field values (known also as *concept hierarchies* [30]) can induce predicate hierarchies in any domain of discourse. But we must be careful what predicates we use: *quarter*≠`1` is not a falsehood-transmitting approximation to *month*≠`"February"` (although obviously it transmits truth); and *enws*<`"SE"` certainly is not a meaningful approximation to *angle*<`330`.

Predicates can be thought of as a special kind of expression on the underlying fields. We can thus generalize predicate hierarchies to hierarchies of expressions based on concept hierarchies of fields. For example, consider the lower two rows of the table

West			Prairie			Ont			East		
BC	AL		SK		MN	ON			QC		NS
V	E	C	S	R	W	H	T	O	M	Q	H
12	2	5	1	0	0	3	8	2	3	1	1
4	1	2	1	1	1	1	3	1	1	1	1

where the third row is cities (with abbreviated names), the fourth row is thousands of immigrants last year, and the last row is the same values expressed as quartiles of the maximum (Vancouver's influx of 12,000). We can think of the last two rows as a numeric concept hierarchy in which the quartiles are an approximation to the absolute numbers. In these two rows, any expression defining the **max** or **min** of two or more entries in one row has the same meaning in the other row. Thus, the predicate telling us that Vancouver had a larger influx than Toronto would be the same expression in both cases and have the same value. The aggregate expression that answers the question which city had the largest influx with the result `"V"` is the same for both rows.

Changing gears slightly, consider the top two rows of the table, where the top row is the region top level in a geographical concept hierarchy, and the second row is the provincial level in this hierarchy. Summing the numbers in the fourth row by province gives the immigrant influx by province. Summing the provincial numbers by the regions in the top row follows exactly the same process, and, in a good programming language, uses the same expression.

This is another way of keeping the same expression with the same meaning across levels of a hierarchy.

Of course, it is easy to see that many expressions do not keep the same meaning. For instance, summing the quartiles by province or by region fails to make sense in the same way as summing the absolute numbers, without further operations. And, of course, the two levels being subjected to the same expression must be of the same type: we can subtract angles, but not the character strings `"E"`, `"NE"`, etc.

(Concept hierarchies, while applicable equally to spatial or nonspatial data, are well known to mapmakers who routinely reduce continuous numerical variables such as altitude to half a dozen colours representing, say, below sea level, very low, low, medium, high, and very high.)

3.6 Computer-Aided Design

Another important computational discipline involving spatial data is computer-aided design (CAD). This is used, for instance, by architects to plan buildings or by engineers to design mechanisms (even to the extent of directly instructing a numerically-controlled cutting tool how to make the components) or structures. Different disciplines make different demands on CAD, but they are mostly more interested in three dimensions than is G.I.S. They mostly deal with smaller models than maps, usually sufficiently small to fit into RAM and so they are of less interest to us.

Architectural CAD [36, 46] usually proceeds from two dimensions to three by translation. A two-dimensional plan is extruded upwards to give the walls of a building. A sloping roof may be made by specifying different heights for its ends, but more elaborate three-dimensional designs, such as geodesic domes, elude most architectural CAD. Mechanical engineering CAD [28] can also rotate a section to produce cylindrical three-dimensional models, or can use full constructive solid geometry (CSG). This uses point-set operations in three dimensions to build solid shapes out of a base set of simple solids.

The kinds of operations needed on the constructed models include visualization (for clients and sales brochures), and, for architecture especially, walk-throughs of the designed building. Mechanical engineers need simulations of motion of components, such as gears and linkages. Civil engineers need stress-strain analyses of designed structures often using finite element techniques (another computational adaptation of geometrical calculus).

3.7 Advanced Geospatiotemporal Applications

We cannot possibly achieve complete coverage of the field, so we have not addressed topics such as map generalization, map transformation, spatial statistics, population potentials, etc. [39]. We encourage the reader to apply the techniques we discuss below, to discover if they suffer from fundamental omissions.

4 Spatial Data Structures for Secondary Storage

The purpose of this tutorial is to unify spatial computation with all the associated data processing needed by geospatial systems, nonspatial in particular. This means that we must be able to do everything with one language. And that means that *spatial is not special*, at least as far as language is concerned. This is hardly surprising. Programming languages in which algorithms, including spatial algorithms, are written, such as the venerable FORTRAN or the more recent C, do not have spatial dialects. Their formalisms are general-purpose, and both spatial and non-spatial topics can be expressed without special syntax or semantics. Only on secondary storage has it become the practice to introduce special dialects or whole languages for spatial computation, with the result that G.I.S. have grown up independently of database languages, for instance, and with the consequence that putting the two back together is now a consuming and potentially painful effort.

In the realm of data structures, spatial *is* special, as are most particular applications. It is central to the optimal algorithms of computational geometry, for example, to use the right data structures. This section discusses spatial data structures for programming on secondary storage.

We have established the foundations of a secondary storage programming language, based on relations and relational operators, for the reasons given in section 2. So our data structures must be relational data structures. We *define* a data structure, for secondary storage purposes, to be a *set of relations*.

This definition excludes a rich collection of *implementation* data structures, such as those eruditely covered by Samet [44] and more recently extended by van Oosterom [49]. Such data structures are at a lower level of abstraction than geospatial programmers should deal with. They offer much scope for the implementation of relations and relational operators (remember we said that relations do not have to be implemented as simple files), but the present discussion must be able to avoid such levels of detail.

We must reveal one caveat about this exclusion. Since relations abstract whole files, and since sublinear computational complexity, e.g., $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$, involves only part of a file of n blocks, none of the algorithms we discuss can be shown to have sublinear complexity without invoking the implementation. When we claim, below, sublinear complexity for any algorithm or any part of an algorithm, we depend on knowledge, beyond the scope of this tutorial, of the implementation. (It is for this reason that this tutorial does not discuss spatial indexing, a technique which is important for implmenting sublinear searches, but not usually useful for more complex processing.)

4.1 Enumerated Sequences versus Element-Pairs

We can start with the data structure illustrated in section 2. This represented triangles as sequences of vertices, and can be extended to any polygons. The *id* field distinguishes one polygon from another. A *type* field could be added to distinguish polygons (closed sequences) from polylines (open sequences), and a zero-dimensional item, consisting of a vertex by itself, could be considered a special case of either, with only one element in the sequence. The *type* information would inform any calculations being done on the data structure that a polygon has an extra edge connecting the last with the first vertex in the sequence, while a polyline does not.

Thus, the representation of section 2 is already quite versatile, capable of representing zero-, one-, and two-dimensional features of a map or two-dimensional construct, using only one relation.

This data structure uses a sequence number to order tuples in the relation. An equivalent representation of a sequence represents each edge in the polygon or polyline as an ordered pair of vertices. Here are the triangles of section 2 in this new form.

polygons $'$(*id*	x_1	y_1	x_2	y_2)
F1	3.0	2.0	1.0	2.0
F1	1.0	2.0	2.0	1.0
F1	2.0	1.0	3.0	2.0
F2	3.0	2.0	2.0	3.0
F2	2.0	3.0	1.0	2.0
F2	1.0	2.0	3.0	2.0

We call the representation using the sequence numbers an *enumerated sequence* and that using pairs of vertices an *element-pair* representation. For cycles, i.e., closed sequences, appendix A shows the equivalence of these two representations, by giving the programs for secondary storage that transform each into the other.

An advantage of the latter is that each tuple can to some extent be considered independently of other tuples (although a set of tuples is still required to represent a whole polygon). Calculations are often easier with independent tuples, except when the whole sequence must be put in order. It is also easier for an implementation to execute operations in parallel if the processing does not depend on a sequence. A potentially serious disavantage, however, arises in representing sequences with repeated elements: the sequence, (a, b, a, c) is unambiguous as the set $\{(a, 1), (b, 2), (a, 3), (c, 4)\}$, but cannot be reconstructed from the set $\{(a, b), (b, a), (a, c)\}$. (Planar and spherical polygons do not have repeated vertices, but polygons on a torus may. We will discuss data structures that do not care about the underlying topology, and so work well for tunnels and bridges as well as more conventional geospatial data.)

Drawbacks are that the resulting relation has more fields, the connections between the edges are not so obvious to the eye, polygons must be distinguished from polylines by the explicit presence of the closing edge, and isolated vertices do not fit nicely into the scheme.

On the other hand, the enumerated sequence representation is easier for the human viewer, precisely because it shows the whole sequence clearly when the tuples are displayed in sequence order.

A difficulty with the element-pair representation as given above is that the connections of the edges to each other depend on the coordinates given for their endpoints. A little roundoff error in one of the coordinates may cause the connection to be missed. This is easily fixed by storing the coordinates in another relation and giving each vertex a name or identifier to link the endpoints both to the coordinates and to each other for the connectedness. This amounts to separating the geometry from the topology by separating out the coordinates.

A further advantage of separate coordinates is that the dimensionality of the space is also separated from the topology. The coordinates can be three-dimensional, for instance, and algorithms involving the topology need not be concerned. (Three-dimensional, nonplanar, polygons are of interest to computational biologists studying proteins, and, of course, for maps.)

With separate coordinates, each of the above two data structures (enumerated sequence, and element-pair) now consists of two relations, Here is the latter for the two triangles. Notice that the sharing of two vertices by the triangles is now explicit.

polygons $''$(*id*	v_1	v_2)
F1	V1	V2
F1	V2	V3
F1	V3	V1
F2	V1	V4
F2	V4	V2
F2	V2	V1

vertices(*id*	x	y)
V1	3.0	2.0
V2	1.0	2.0
V3	2.0	1.0
V4	2.0	3.0

There are some problems with even this improvement, which we can see from the example. Two of the triangles share an edge (`V2, V1`), and this edge is repeated in the data structure. Such redundancy is risky when updates are made, in case one of the edges gets changed but not the other, although there may be a semantic constraint that these two occurrences are always the same edge. (We do *not* talk about "wasted storage", because this is an implementation issue and not our concern at this level of abstraction.)

The data structure is good for isolated polygons and polylines (and vertices, in the case of the enumerated-sequence version), but not for general *subdivisions* of the plane, as maps usually are. Such subdivisions usually have many shared edges, to the extent that almost every edge belongs to two polygons and so gets repeated.

It also does not generalize to more than two dimensions. An isolated *polyhedron* is a planar subdivision wrapped up in three dimensions to be topologically equivalent to a sphere (or a torus, or some "multiply-connected" manifold), and representing it by this data structure has the same problems as representing a planar subdivision. Such objects are of less interest to GIS than to CAD, but we mention them because they have the same representational difficulties as maps, and it would be valuable to find a solution to both problems if there is no extra cost.

4.2 The Quad-Edge Representation

The idea needed to solve all these problems is to move from sequences of vertices to sequences of edges. The basis for this move is that a vertex is a cycle of edges; and a polygon, or face, is also a cycle of edges. The sequences in this new approach are all cycles, so we need no longer distinguish closed from open sequences. Even the endpoint of an isolated edge is a cycle of one edge. And a face can have one edge, and one vertex which is both endpoints of the edge: this would also be a cycle of one edge. (We are getting free of straight edges and polygons, which are both geometrical concepts, since straightness requires the concept of distance. We can really separate topology from geometry.) The cycles are conventionally taken counterclockwise, in the direction of increasing angle.

An important extra benefit of this way of thinking is that representing both vertices and faces of a subdivision captures both the topology and the dual topology of a map. The *dual* of a subdivision is a subdivision whose faces were vertices in the original, and whose vertices were faces in the original. Each edge of the original has an edge from the dual which crosses it. The dual of our two connected triangles has five edges (as do the triangles themselves) and three vertices. Two of these vertices correspond to the two triangular faces in the original, and so each have three dual edges emerging from them. The third vertex corresponds to the external "face", and has four dual edges, corresponding to the four edges bounding this face. This external face is usually included in topological thinking, for completeness. It contains the "point at infinity", which makes the plane topologically equivalent to a sphere. (For mapping, this is an easy concept, since the Earth is (almost) a sphere.) Figure 5 shows the original using solid lines and the dual using dashed lines.

Each edge generates four labels in this representation. Consider, for example, the edge labelled e in figure 5. This is a solid line. Its dual, the dashed line crossing it we could label e', but there are also the lines with opposite directions, and we will have too many primes. So we will use indexes from 0 to 3 to distinguish each label, and represent the four as (*edge, direction*) pairs: $(e, 0), (e, 1), (e, 2), (e, 3)$. The even indexes stand for the original edge and its oppositely-directed counterpart; the odd indexes are the dual edge and its opposite. These may be thought of as a cycle of counterclockwise rotations, from edge to dual to opposite edge to opposite dual. The name "quad-edge" is from the four directed edges thus generated

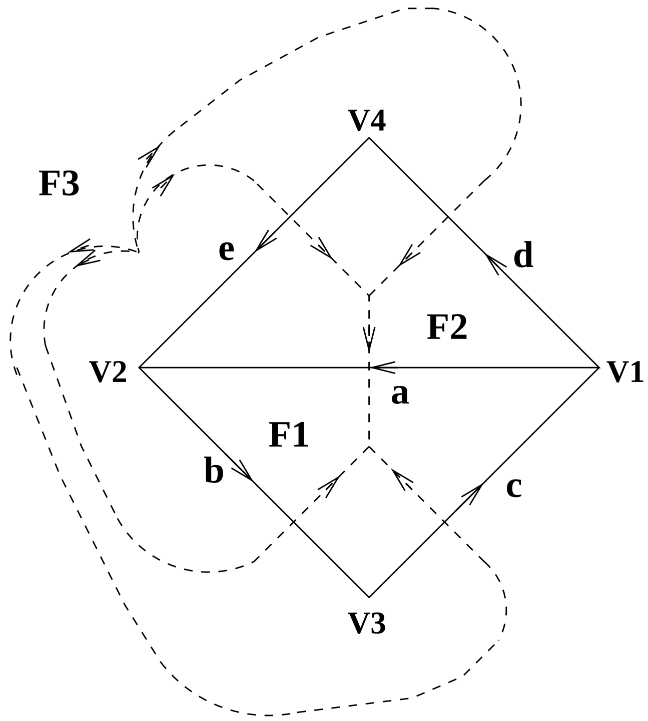


Figure 5: A subdivision of space and its dual

for each edge in the original.

The reason we need the oppositely directed edges and duals is so that we can always use outgoing edges to identify each vertex and face. Thus, vertex $V1$ is the cycle $(a, 0), (c, 2), (d, 0)$, and face $F2$ is the cycle $(a, 1), (d, 3), (e, 3)$. The data structure for this example could be written, as an enumerated sequence, as shown in figure 6. Note that both vertices and faces are represented in the same way, and so all the topology may be included in the single relation, *QuadEdgeES*.

Element pairs are a better way to represent these cycles, and encounter no ambiguities since no cycle repeats any edge. We can remove references to vertices and faces from the topology, since these are used above only to group the sequences. To identify faces and vertices, and to record the coordinates of the latter, we introduce the notions of the *origin* and *destination* vertices of an edge, and the *left* and *right* faces, and we represent these in an intermediate relation, *VertFace*. The direction is omitted from *VertFace*, and is taken to be 0; the entries for the three other directions are easily generated. (See figure 7.) Note that we have modified *Geom* to include information about the faces as well as coordinates of the vertices. The coordinates shown for the faces are for the junctures of the edges of the *Voronoi diagram* of the four vertices. This is the set of edges that divide the plane into four regions whose points are, respectively, nearest to each of the four vertices. In this example, the junctures for faces $F1$ and $F2$ are common, both at the centre. The juncture for face $F3$ is taken to be the point at infinity (or the point on the Earth diametrically opposite the centre).

The quad-edge data structure was proposed by Guibas and Stolfi [27], who show that it is capable of representing subdivisions not only of the extended plane or sphere, but of any two-dimensional manifold, no matter how multiply-connected (e.g., a torus is doubly-connected),

QuadEdgeES(*fv*	*seq*	*edge*	*dir*)
$V1$	1	a	0
	2	c	2
	3	d	0
$V2$	1	b	0
	2	a	2
	3	e	2
$V3$	1	b	2
	2	c	0
$V4$	1	d	2
	2	e	0
$F1$	1	a	3
	2	b	3
	3	c	3
$F2$	1	a	1
	2	d	3
	3	e	3
$F3$	1	e	1
	2	d	1
	3	c	1
	4	b	1

Geom(*vert*	x	y)
$V1$	3.0	2.0
$V2$	1.0	2.0
$V3$	2.0	1.0
$V4$	2.0	3.0

Figure 6: Quad-Edge Relations (Enumerated Sequence)

QuadEdge(*edge1*	*dir1*	*edge2*	*dir2*)
a	0	c	2
c	2	d	0
d	0	a	0
b	0	a	2
a	2	e	2
e	2	b	0
b	2	c	0
c	0	b	2
d	2	e	0
e	0	d	2
a	3	b	3
b	3	c	3
c	3	a	3
a	1	d	3
d	3	e	3
e	3	a	1
e	1	d	1
d	1	c	1
c	1	b	1
b	1	e	1

VertFace(*edge*	*org*	*dest*	*left*	*right*)
a	$V1$	$V2$	$F1$	$F2$
b	$V2$	$V3$	$F1$	$F3$
c	$V3$	$V1$	$F1$	$F3$
d	$V1$	$V4$	$F2$	$F3$
e	$V4$	$V2$	$F2$	$F3$

Geom(*vf*	x	y)
$V1$	3.0	2.0
$V2$	1.0	2.0
$V3$	2.0	1.0
$V4$	2.0	3.0
$F1$	2.0	2.0
$F2$	2.0	2.0
$F3$	∞	∞

Figure 7: Quad-Edge Relations (Element Pairs)

or even if inside and outside are the same (e.g., a Klein bottle). They give a thorough treatment, which we here adapt to secondary storage. We are indebted to Christopher Gold [25] for not only insisting on the importance of this structure for geomatics but for sending us copies of the paper and of his own implementation.

4.3 The Splice Operator

As we have said for databases in general, it is not sufficient to provide a representation only. What makes or breaks a data model or structure is the facility with which it can be manipulated. For this we need operators, such as provided by the relational algebra. In the case of the quad-edge structure, apart from operators to create or destroy edges, only one operator is needed, and it is particularly simple. This is the *splice* operator.

For any sequence or cycle, $splice(t, u)$ is the operator on two elements, t and u, that swaps their positions at the start (or at the end) of their element pairs. Strictly speaking there are two splice operators in general, one for the firsts of pairs and one for the seconds of pairs, but these will be combined for the quad-edge representation. To begin, we will distinguish them as $splice1(t, u)$ and $splice2(t, u)$.

Let's see what happens when we $splice1((b, 0), (a, 2))$ in the cycle

$$(b, 0), (a, 2), (e, 2)$$

(*edge1*	*dir1*	*edge2*	*dir2*)		(*edge1*	*dir1*	*edge2*	*dir2*)
b	0	a	2		a	2	a	2
a	2	e	2	$\Rightarrow$	b	0	e	2
e	2	b	0		e	2	b	0

The splice *disconnects* the cycle, making two: a unit cycle of $(a, 2)$, and a binary cycle of $(b, 0), (e, 2)$.

Because it is a swap, *splice*() is obviously idempotent: it is its own inverse. So doing $splice1((b, 0), (a, 2))$ a second time has the effect of inserting $(a, 2)$ *after* $(b, 0)$ in its cycle. (*Splice*() is also commutative: $splice(t, u) = splice(u, t)$.)

Splice2() has the effect of inserting *before*. Here is $splice2((b, 3), (a, 1))$ applied to the cycles

$$(a, 1), (d, 3), (e, 3) \text{ and } (a, 3), (b, 3), (c, 3)$$

(*edge1*	*dir1*	*edge2*	*dir2*)		(*edge1*	*dir1*	*edge2*	*dir2*)
a	3	b	3		a	3	a	1
b	3	c	3		a	1	d	3
c	3	a	3	$\Rightarrow$	d	3	e	3
a	1	d	3		e	3	b	3
d	3	e	3		b	3	c	3
e	3	a	1		c	3	a	3

The effect is to break each of the two cycles just before the operand in *splice2*(), then combine them into one cycle at these breaks. (Note that the rows have been reordered to reveal the single cycle.)

The corresponding effect of *splice1*(), applied to two cycles, is to break each of the two cycles just after the operand, then combine them into one cycle at these breaks. The above example, with a unary cycle, does not show this clearly.

For quad-edge, we now define $splice((e, d), (e', d'))$ to be (d, d' even)

$splice1((e, d), (e', d'))$

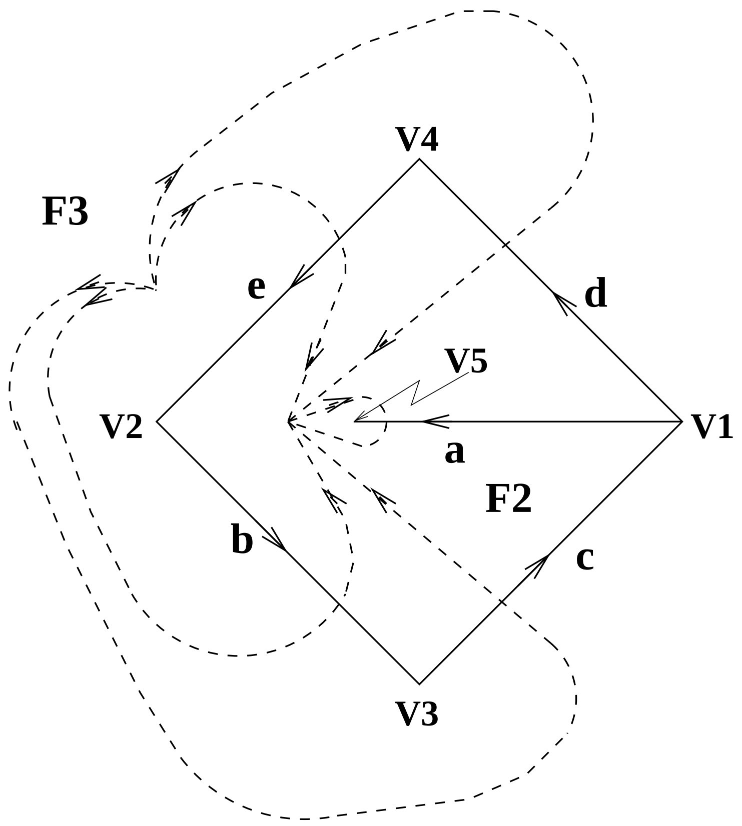


Figure 8: The effect of $\mathit{splice}((b,0),(a,2))$

$$\mathit{splice2}((e,d-1),(e',d'-1))$$

performed in either order, where the arithmetic is done modulo 4. The overall effect on the example subdivision is shown in figure 8. We see that $\mathit{splice}((b,0),(a,2))$ completely disconnects edge a from its common vertex with b before it (and e after it), and *automatically merges the faces on either side of a.*

Because the disconnection has created a new vertex and eliminated a face, we must make corresponding changes to relations *VertFace* and *Geom* in the data structure. The presence of a new vertex and the loss of a face appear with the unary cycle in *QuadEdge*, $((a,2),(a,2))$. In *VertFace*, the tuple

$$(a,V1,V2,F1,F2)$$

must be changed to

$$(a,V1,V5,F2,F2)$$

to reflect the new origin of $(a,2)$ (i.e., destination of $(a,0)$) and the merging of $F2$ with $F1$. Coordinates must be supplied for the new vertex, $V5$, which must be additional input to the disconnect procedure that includes $\mathit{splice}((b,0),(a,2))$. No other occurrence of $V2$ need be changed, but $F1$ must be replaced by $F2$ everywhere in *VertFace*: this happens only for tuples containing dual edges between $(b,3)$ and $(a,1)$. If the data in *Geom* for $F2$ must be changed, this must also be supplied, as well as new coordinates for `V5`.

A further disconnect operation, using $\mathit{splice}((d,0),(a,0))$, will remove the other end of a, creating a vertex $V6$ and leaving a square. The dual edge to a also gets disconnected, so that a and its dual are completely isolated and can be thought of as occupying an entirely

QuadEdge(*edge1*	*dir1*	*edge2*	*dir2*)
d	0	c	2
c	2	d	0
a	0	a	0
a	**2**	a	2
b	**0**	e	2
e	2	b	0
b	2	c	0
c	0	b	2
d	2	e	0
e	0	d	2
a	3	**a**	**1**
b	3	c	3
c	3	**d**	3
a	1	**a**	3
d	3	e	3
e	3	**b**	**3**
e	1	d	1
d	1	c	1
c	1	b	1
b	1	e	1

VertFace(*edge*	*org*	*dest*	*left*	*right*)
a	**V6**	**V5**	**F0**	**F0**
b	$V2$	$V3$	**F2**	$F3$
c	$V3$	$V1$	**F2**	$F3$
d	$V1$	$V4$	$F2$	$F3$
e	$V4$	$V2$	$F2$	$F3$

Geom(*vf*	*x*	*y*)
$V1$	3.0	2.0
$V2$	1.0	2.0
$V3$	2.0	1.0
$V4$	2.0	3.0
V5	**1.75**	**2.0**
V6	**2.25**	**2.0**
F0	∞	∞
$F2$	**1.5**	**2.0**
$F3$	∞	∞

Figure 9: Quad-Edge Relations with Edge a Disconnected

separate manifold. In this sense, we can imagine that a new face, $F0$, has been created as the only face of this new manifold. Figure 9 is the data structure showing the final result after

$$splice((b,0),(a,2))$$
$$splice((d,0),(a,0))$$

with the changes shown in bold.

The inverse process, inserting a new edge in a subdivision, starts with the edge in a separate manifold, created, let's say, by a *createEdge*() operation, and uses exactly the same two *splice*() operations to insert it after d at $V1$ and after b at $V2$.

To "swap" edge a, so that it connects $V3$ to $V4$ instead of $V1$ to $V2$, we disconnect a as above, then reconnect it using

$$splice((e,0),(a,2))$$
$$splice((c,0),(a,0)).$$

This is useful in constructing Delaunay triangulations.

To add a vertex to an edge of a polygon, say to convert a triangle to a quadrilateral, we must disconnect an endpoint vertex of the edge, then insert a new edge between that vertex and the newly created vertex. For example, we could insert a new edge, f, between a and $V2$, after disconnecting the first end of a, as above, by creating f, then following

$$splice((b,0),(a,2)),$$

with

$$splice((f,0),(a,2))$$
$$splice((b,0),(f,2)).$$

This is useful, among other applications, in constructing polygon overlays.

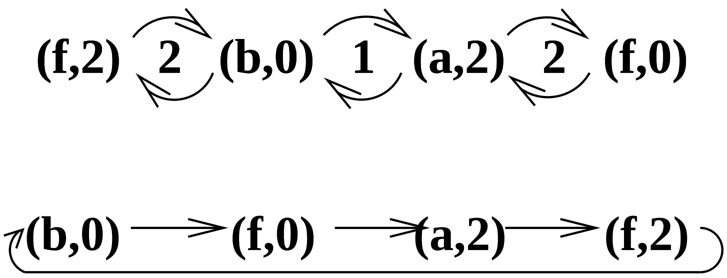


Figure 10: Combining Splices into Substitutions

4.3.1 Splice on Secondary Storage

The splice operation is microscopic, from the point of view of secondary storage: it affects only four records, by swapping either the first or the second elements of the two pairs. Yet, many constructions require several splices in sequence, and may even affect the bulk of the file of pairs.

We have said that it is better, on secondary storage, to do all required processing on data once retrieved, than to retrieve it repeatedly for each processing step. A sequence of splices is a case in point. For example, the above sequence,

$splice((b,0),(a,2))$
$splice((f,0),(a,2))$
$splice((b,0),(f,2))$

has the following effect on the vertex cycles.

(*e1*	*d1*	*e2*	*d2*)		(*e1*	*d1*	*e2*	*d2*)		(*e1*	*d1*	*e2*	*d2*)		(*e1*	*d1*	*e2*	*d2*)
b	0	a	2		a	2	a	2		f	0	a	2		f	0	a	2
a	2	e	2	$\Rightarrow$	b	0	e	2	$\Rightarrow$	b	0	e	2	$\Rightarrow$	f	2	e	2
f	0	f	0		f	0	f	0		a	2	f	0		a	2	f	0
f	2	f	2		f	2	f	2		f	2	f	2		b	0	f	2

(Refer back to figure 7 for the full starting data.)

This can be achieved in one step by combining the effects of all the splice operations before applying them to the file. Each splice is a swap: combining them does not always give swaps, but can result in simple substitutions (half swaps). Figure 10 shows the swaps specified by the three splices. Each swap is numbered according to the place of the corresponding splice in the sequence: the last two splices are independent, so can be performed in any order, so are given the same number. The lower diagram in figure 10 shows the substitutions that result,

$(b,0) \rightarrow (f,0),$
$(f,0) \rightarrow (a,2),$
$(a,2) \rightarrow (f,2).$
$(f,2) \rightarrow (b,0).$

To work with quad-edge data structures on secondary storage, it is usually advisable to combine all the required splice operations in this way before applying the result to the file. The combination can usually be done in RAM, and the relevant parts of the file need be accessed only once.

5 The Domain Algebra and Nested Relations

In this section, we introduce a set of operations complementary to the relational algebra (and to SQL), which allow arbitrary manipulation of the fields of a relation. These operators form a "domain algebra". (This is mistakenly named, because "domain" is a poor synonym for "field", or even "attribute" in the database sense.)

The domain algebra can be divided into scalar and aggregate operators. It is much more expressive than the limited arithmetic capabilities of SQL, and than SQL's weak collection of aggregate functions, which furthermore demands explicit syntax for each (COUNT, SUM, AVG, MAX, MIN). It can, fortunately, be added to SQL with no changes to existing SQL syntax.

The domain algebra is algebraic in that it operates on fields ("domains") to produce new fields. It is closed: only fields result. It is abstract: the relations the fields may belong to are not considered in the formalism. These two properties give the programmer using the domain algebra a great deal of intellectual freedom to avoid extraneous thinking.

5.1 The Scalar, or "Horizontal", Operations

Scalar operations apply within tuples. They are also called "horizontal" in reference to the table form of relations (which is not the only way to think of relations) in which each tuple is a row of the table.

Simple arithmetic, or boolean or string operations, within each tuple is the obvious capability provided by the scalar domain algebra. All the usual arithmetic and other operations are available, such as $+$, $\times$, unary and binary $-$, **and**, **or**, etc., and infix binary **min** and **max**, as well as the usual collection of functions (*sqrt*(), *abs*(), trigonometric, etc.). As an illustration, consider the Euclidean and Manhattan lengths of a set of edges,

let *eLen* **be** *sqrt* ((*Xorg*−*Xdest*)∧2 + (*Yorg*−*Ydest*)∧2);
let *mLen* **be** *abs*(*Xorg*−*Xdest*) + *abs*(*Yorg*−*Ydest*);

We can see what this looks like for the quadrilateral at the end of section 4.3. *EdgeSet* is a relation which has all the fields needed for both these calculations. We show it below, with the new fields written outside the parentheses, to indicate the values that might be calculated if the above domain algebra were somehow applied to *edgeSet*: we will not discuss how this might happen until section 5.3.

edgeSet(*edge*	*Xorg*	*Yorg*	*Xdest*	*Ydest*)	*eLen*	*mLen*
b	1.0	2.0	2.0	1.0	1.4142	2.0
c	2.0	1.0	3.0	2.0	1.4142	2.0
d	3.0	2.0	2.0	3.0	1.4142	2.0
e	2.0	3.0	1.0	2.0	1.4142	2.0

We note that the domain algebra **let**-statement defines a value for the new field for each tuple of *edgeSet*. There is no choice: since relations are not part of the domain algebra formalism, any relation to which the new field is eventually connected must have a value for that field for each tuple. Thus the domain algebra, like the relational algebra, abstracts over looping, and so works at the same high level of abstraction.

We also note that, in this particular example, both new fields happen to have the same values in each tuple. They are thus, accidentally, what is called a "constant field". Constant fields are one special capability of the scalar domain algebra.

let *One* **be** 1;

or, more subtly, the 2 in

let *eLen* **be** *sqrt* $((Xorg - Xdest) \wedge 2 + (Yorg - Ydest) \wedge 2)$;

A second special capability of the scalar domain algebra is *renaming*. We needed this in section 2.

let *seq1* **be** seq;
let *seq2* **be** seq;

Note that an existing field can be renamed in more than one way, but renaming two existing fields to the same new name at the same time is ambiguous. However, two existing fields may be renamed to the same new name at different times. (These restrictions are analogous to societies which permit serial polygamy while discouraging concurrent polygamy.)

5.2 The Aggregation, or "Vertical", Operations

Aggregation operations apply within fields and across tuples. They are also called "vertical" in reference to the table form of relations (which is not the only way to think of relations) in which each field is a column of the table.

There are two families of aggregation operators, *reduction* and *functional mapping*. These each include a simple form and a grouping form.

5.2.1 Reduction and Equivalence Reduction

Examples of reduction are to find the perimeter of the above quadrilateral, and to find its area, using Stokes' theorem (section 3.4).

let *perim* **be red** + **of** *eLen*;
let *area* **be red** + **of** $(Xorg \times Ydest - Yorg \times Xdest)$;

Here are these results associated with *edgeSet*

edgeSet(*edge*	*Xorg*	*Yorg*	*Xdest*	*Ydest*)	*perim*	*area*
b	1.0	2.0	2.0	1.0	5.6569	2.0
c	2.0	1.0	3.0	2.0	5.6569	2.0
d	3.0	2.0	2.0	3.0	5.6569	2.0
e	2.0	3.0	1.0	2.0	5.6569	2.0

Once again, the new fields are constant, this time not accidentally: the perimeter and the area apply to the whole polygon. Once again, we have no choice but to associate the constant value with each tuple. The apparent wastefulness is not an issue: it is only apparent, and the programmer can avoid it easily (see section 5.3).

The **red** clause takes an operator as its operand. In this case we want to sum so the operator is +. It could also be $\times$, **max**, **min**, **and**, **or**, etc. It could *not* be $-$: it must be both associative and commutative, because relations define no order on their tuples, and only operators that are both associative and commutative give results that are independent of the order of the operands. We call such operators *reduction operators*, or "*redops*". (Operators, such as absolute difference, that are commutative but not associative, we call *symmetric operators*, or "*symops*". Operators, such as string concatenation, that are associative but not commutative, play no special syntactic role.)

Special cases of reduction are thus sum, count, min, max, and average.

let *sum* **be red** + **of** *eLen*;
let *count* **be red** + **of** 1;
let *min* **be red min of** *eLen*;
let *max* **be red max of** *eLen*;

let *avg* **be** (**red** + **of** *eLen*)/(**red** + **of** 1);

or

let *avg* **be** *sum*/*count*;

The example of average is possible because of the closure and the abstraction of the domain algebra: if *sum* or *count*, or their anonymous equivalents, **red** + **of** *eLen* and **red** + **of** 1, were not themselves fields, independent of any relation, we would not be able to combine them, and the interpretation of the result would be confused.

Of course, we can now go beyond these basic aggregations.

let *stdDev* **be** *sqrt*((**red** + **of** (*eLen*−*avg*)∧2)/*count*);

The reduction calculations, above, are done on a single polygon. If we have many polygons, or a subdivision, we need to be able to calculate, say, perimeters and areas for each without writing loops. We introduce a *poly* field, which breaks the tuples of the relation into *equivalence classes*, each class with all the same values for *poly*.

let *perim* **be equiv** + **of** *eLen* **by** *poly*;

let *area* **be equiv** + **of** *Xorg*×*Ydest* − *Yorg*×*Xdest* **by** *poly*;

Here are these results associated with *edgeSet* extended to include *poly* and describing the two triangles in section 2.

edgeSet(*poly*	*edge*	*Xorg*	*Yorg*	*Xdest*	*Ydest*)	*perim*	*area*
*F*1	*a*	3.0	2.0	1.0	2.0	4.8284	1.0
*F*1	*b*	1.0	2.0	2.0	1.0	4.8284	1.0
*F*1	*c*	2.0	1.0	3.0	2.0	4.8284	1.0
*F*2	*d*	3.0	2.0	2.0	3.0	4.8284	1.0
*F*2	*e*	2.0	3.0	1.0	2.0	4.8284	1.0
*F*2	*a*	1.0	2.0	3.0	2.0	4.8284	1.0

The constancy of the new fields is now, again, accidental. We expect them to have the same value within each *poly*gon, but not necessarily the same from one polygon to the next. If we had included $F3$ in the relation, the same domain algebra would have yielded

edgeSet(*poly*	*edge*	*Xorg*	*Yorg*	*Xdest*	*Ydest*)	*perim*	*area*
*F*1	*a*	3.0	2.0	1.0	2.0	4.8284	1.0
*F*1	*b*	1.0	2.0	2.0	1.0	4.8284	1.0
*F*1	*c*	2.0	1.0	3.0	2.0	4.8284	1.0
*F*2	*d*	3.0	2.0	2.0	3.0	4.8284	1.0
*F*2	*e*	2.0	3.0	1.0	2.0	4.8284	1.0
*F*2	*a*	1.0	2.0	3.0	2.0	4.8284	1.0
*F*3	*e*	1.0	2.0	2.0	3.0	5.6569	-2.0
*F*3	*d*	2.0	3.0	3.0	2.0	5.6569	-2.0
*F*3	*c*	3.0	2.0	2.0	1.0	5.6569	-2.0
*F*3	*b*	2.0	1.0	1.0	2.0	5.6569	-2.0

Note that Stokes' theorem produces a negative area if the edges are traversed clockwise. This is useful to find the area of a region with holes in it, and to detect left versus right turns from one edge to another.

(For a triangle, Stokes' form of the area reduces to half the determinant

$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_1 - x_1y_3 = \begin{vmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{vmatrix} = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

which makes sense if we realize that diagonalizing the matrix makes the determinant a product of two orthogonal directions. Similar results apply to the volumes of tetrahedra, and of any simplex (d-dimensional polytope with $d+1$ vertices).)

"Group-by" sums, counts, averages, etc. can all be found using this *equivalence reduction*. The requirement that the operator be associative and commutative (a redop) holds for equivalence reduction also.

5.2.2 Functional Mapping and Partial Functional Mapping

In spatial data, ordering can be important, as in the case of the order induced by a *seq*uence field on the tuples of the relations in section 2. Here, the tuples are not ordered in themselves. They can appear in the relation in any order. But each tuple has a position in the strict ordering defined by the value of *seq*.

We can use this ordering to calculate a cumulative sum, say, of a histogram (a distribution function) to give the cumulative distribution function.

let *cumL* **be fun** + **of** *histL* **order** *value*;
let *cumR* **be fun** + **of** *histR* **order** *value*;

Note the **order** clause, which induces the order on the tuples.

Here are two histograms on the same range of values, for which the above domain algebra statements respectively compute the cumulative values. (Strictly, this example does not show distributions, which should be normalized to 1, but it has been contrived to have the same number, 60, of entries in each case, so dividing everything by 60 would give the distributions.)

histograms(*value*	*histL*	*histR*)	*cumL*	*cumR*	*proporL*	*proporR*
0	8	3	8	3	0.87	0.95
1	24	3	32	6	0.47	0.90
2	23	19	55	25	0.08	0.58
3	5	35	60	60	0.00	0.00

(The average values can be computed from the histograms.

let *avgL* **be red** + **of** *value*$\times$*histL*;
let *avgR* **be red** + **of** *value*$\times$*histR*;

The proportion of each set of data where the value exceeds a specified value can be computed from the cumulations.

let *proporL* **be** 1.0 - *cumL*/(**red max of** *cumL*);
let *proporR* **be** 1.0 - *cumR*/(**red max of** *cumR*);

Here, we have done the division that would not be needed if *cum* were true distributions.)

If the original histograms resulted from spatial data, we have derived and used *spatial cumulative distribution functions* (*SCDF*) [4]. For example, the two histograms might have come from the left and right halves of the following spatial observations (or from regions defined by more complicated polygons over a more complicated choropleth map).

```
 0 0 1 1 1 2 1 0 2 3 3 3 3 2 3
0 0 0 1 1 2 2 1 0 3 2 3 3 2 2
 0 0 1 1 2 2 2 0 1 3 3 3 2 3 2
0 1 1 2 2 2 2 1 1 2 3 3 3 3 3
 1 2 2 2 2 1 1 2 3 2 3 3 2 2 3
1 1 2 3 3 2 1 2 2 3 3 2 3 2 3
 1 1 2 3 3 2 1 1 3 2 2 3 3 3 3
1 1 2 2 3 2 1 2 2 3 3 2 3 3 3
```

Superimposed plots of the two cumulative distributions help to reveal differences between the regions.

Any operator may appear after the **fun** keyword, not just associative or commutative operators. For $-$, for instance, the functional mapping is defined so as to produce an alternating sum.

We can add a grouping facility, as in equivalence reduction, to functional mapping, giving *partial functional mapping.* This saves us writing repeated statements in the above example.

let *cum* **be par** + **of** *hist* **order** *value* **by** *region*;
let *avg* **be equiv** + **of** *value*×*hist* **by** *region*;
let *propor* **be** 1.0 - *cum*/(**equiv max of** *cum* **by** *region*);

It also allows us to do all the calculations for an indefinite number of regions with exactly the same three statements.

histograms(*region*	*value*	*hist*)	*cum*	*propor*
L	0	8	8	0.95
L	1	24	32	0.90
L	2	23	55	0.58
L	3	5	60	0.00
R	0	3	3	0.87
R	1	3	6	0.47
R	2	19	25	0.08
R	3	35	60	0.00

Two operators are useful in the context of functional (and partial functional) mapping, which have little significance on their own. These are the successor and predecessor operators, **succ** and **pred**, respectively. We can use them, for instance, to convert from the enumerated-sequence representation of a cycle to the element-pair representation.

let *xcoord'* **be par succ of** *xcoord* **order** *seq* **by** *id*;
let *ycoord'* **be par succ of** *ycoord* **order** *seq* **by** *id*;

Applied to the relation *polygons* in section 2, these give

polygons(*id*	*seq*	*xcoord*	*ycoord*)	*xcoord'*	*ycoord'*
F1	1	3.0	2.0	1.0	2.0
F1	2	1.0	2.0	2.0	1.0
F1	3	2.0	1.0	3.0	2.0
F2	1	3.0	2.0	2.0	3.0
F2	2	2.0	3.0	1.0	2.0
F2	3	1.0	2.0	3.0	2.0

where we note the cyclic effect of **par succ of** (**par pred of** is likewise cyclic). For sequences that are open, not cycles, the last tuple can be got rid of by a relational algebra selection.

5.3 Some More Relational Algebra

Although the domain algebra seems a slight extension of SQL capabilities, aggregation in particular, the significant advance over SQL is subtle: the complete separation of domain algebra from relations. Fortunately, this makes the domain algebra orthogonal to the relational algebra, and so it could be added to SQL with no changes to existing SQL syntaxes.

To do this requires us to revisit the relational algebra. The domain algebra is an advance because it allows the programmer to separate thinking about fields within relations from thinking about the relations. Eventually however, the two must be connected. The fields defined by domain algebra statements must eventually take their places in the context of relations.

All the results of the domain algebra shown so far have been *virtual*, not yet evaluated for any particular relation, although we have shown them as columns appended to particular relations for concreteness. The step we now address is *actualization.*

Actualization is easy, from the point of view of language syntax: any virtual field may be named anywhere actual fields have hitherto been used in relational expressions. For example, the successor coordinates in the last example can simply be named in the projection list

> **select** *id, xcoord, ycoord, xcoord', ycoord'*
> **from** *polygons*
>   *Virtual field actualization in SQL*

This creates a new relation (not named by SQL), with the fields *id, xcoord*, and *ycoord* from *polygons*, and the two newly actualized fields *xcoord'* and *ycoord'* from the domain algebra definitions. Those definitions referred only to fields *id, seq, xcoord*, and *ycoord*, which are already present in *polygons*, so the actualization is fully specified.

A virtual field name may be used anywhere in the relational algebra where actual fields of the relations have hitherto been used. The implementation of a relational algebra expression will check any field named against the relevant relation; if it is an actual field of the relation, this is used, or else if it is a virtual field defined by some domain algebra statement, the above check is done recursively on the fields used by the domain algebra to define it; if ultimately some needed field is neither actual in this relation nor virtual (i.e., defined by domain algebra), then the actualization fails, otherwise the field is evaluated by the relational algebra. This, of course, specifies a major change in the *implementation* of the relational algebra, although the *syntax* changes not at all.

This shows us that which virtual fields are actualized, and in what contexts, is entirely up to the programmer. Thus the apparent redundancies of constant values, especially resulting from reduction and equivalence reduction, are illusions: the programmer is free to avoid them or to perpetrate them.

Anonymous domain algebra expressions (i.e., any of the above statements omitting the **let .. be**) may also be used in most places actual domains appear in the relational algebra, with the exception being any usage which would place an anonymous field in the result relation.

An important consequence of the domain algebra is that it leads to operations on nested, or "non-first-normal-form", relations. Since the domain algebra is a formalism by which fields can be manipulated, it can use the operators of relational algebra as well as those of arithmetic, etc., and the fields can themselves be relations. To support this, we need a relational algebra notation which distinguishes clearly between expressions and statements, and in which join expressions in particular are explicit. We take a little space and time to examine notation which departs from SQL in ways that are small in appearance but significant in reality.

The unary operator syntax is that of the programming notation introduced in section 2.

> [<field list>] **where** <tuple conditional expression> **in** <relational expression>

where the **where**-clause may be omitted for pure projection and the [<field list>] may be omitted for pure selection.

The major advantage of this rearrangement of the SQL order is that it puts the <relational expression> last so that compound expressions are more easily read and more easily further articulated.

5.3.1 Joins

The binary operator syntax is kept extremely simple and the different operators are explicit.

<relational expression> *join* <relational expression>

or, if the join fields must be named explicitly because their names are not common to the two operands,

<relational expression> [<field list> *join* <field list>] <relational expression>

Now we elaborate on the kinds of join. In section 2, we had only **join**, the natural join. Here we note that a whole family of joins related to the natural join is possible. The natural join generalizes set intersection to relations. The other set-valued set operations are union, difference, and symmetric difference. The first of these generalizes to relations in the same way as intersection generalizes to natural join, giving what is usully known as the *outer join*. The others do not have conventionally known generalizations. We can call the whole suite **ijoin**, **ujoin**, **djoin**, and **sjoin**, respectively, to emphasize these generalizations, and to these we can add **ljoin** and **rjoin**, which generalize the trivial set operators that return, respectively, the left and right operand. For this tutorial, we use only the first three. We call them **join**, **union**, and **diff**, respectively, for familiarity. The latter two we use mainly as simple set operators, as far as we can hold back to the limitations of SQL.

The above family generalizes the *set*-valued operators on sets. Another family of operators on sets is *logic*-valued, such as test for set containment, intersection, etc. In his second paper on the relational model, Codd [16] introduced generalizations to relations of two of these, superset (sup) and intersection-not-empty ($\not\emptyset$), as *division* and *natural composition*, respectively. These did not make it into SQL. All twelve set comparisons may be so generalized to give another family of join operators. These are called σ-joins, to distinguish them from the above family, the μ-joins. In this tutorial, we use only one, the natural composition, **comp** in the syntax. Natural composition can also be thought of as a natural join followed by a projection which gets rid of the join field(s). It could be left to the programmer to write it in this way when needed, but it is a useful operator to implement (along with the other eleven σ-joins) because it frequently avoids a need for cumbersome field renaming.

In summary, for this tutorial, we use three binary operators, or joins: the natural (intersection) join, **join**, the outer (union) join, **union**, and the natural composition, **comp**. We use these symbols to replace the generic *join* in the binary operator syntax, above.

As an example of the use of **comp** and **union**, consider a road network consisting of the four segments given by edges with the following origin and destination vertices. (We will come to the *length* field shortly.)

Edges(*edge*	*org*	*dest*	*length*)
a	$V1$	$V2$	4.0
b	$V2$	$V3$	2.0
c	$V2$	$V4$	5.0
d	$V3$	$V4$	2.0

For the moment, we are interested only in the origin and destination vertices of this network.

OD <− [*org, dest*] **in** *Edges*;

The expression to find out what vertex can be reached from what in this network is on the right of the assignment arrow:

TC <− *OD* **union** *OD*[*dest* **comp** *org*]*OD*;

The natural composition is done first, and finds paths of length two.

$OD(org$	$dest)$	$OD[dest$ **comp** $org]OD$	
$V1$	$V2$	$(org$	$dest)$
$V2$	$V3$	$V1$	$V3$
$V2$	$V4$	$V1$	$V4$
$V3$	$V4$	$V2$	$V4$

The union then combines this result with *OD* to give *TC*.

This simple statement works for this example, because once all paths of length two are accounted for, there are no new connections to discover. (Although there is a path of length three, $V1$–$V2$–$V3$–$V4$, we already found a length-two connection from $V1$ to $V4$, namely $V1$–$V2$–$V4$.) In general, however, we would need a loop containing the slightly modified statement,

TC <− *OD* **union** *OD*[*dest* **comp** *org*] *TC*

Here, TC will accumulate (because of **union**) paths of length one (the original *OD*) and greater (because of **comp** of *OD* with the growing *TC*).

In general, this statement can be expressed as a *recursive view*, and a looping construct is not needed. Discussion of efficient execution of such views, which is under the control of the programmer, is beyond the scope of this tutorial [15].

TC is so named because it is the *transitive closure* of the graph contained in *OD*. The recursive view will compute the transitive closure of any graph, with or without cycles.

More than knowing what vertices can be reached from what, we would like to know what the *shortest path* is between pairs of vertices. This involves a combination of domain and relational algebra, and illustrates well the advantage of keeping these two formalisms independent of each other. The relational algebra part is very similar to the above, except that we must accumulate the total length of the paths. This accumulation can be calculated independently:

let *length'* **be** *length*;
let *length''* **be equiv min of** *length* + *length'* **by** *org, dest*;
let *length'''* **be** *length* **min** *length''*;

The second of these statements adds the lengths of two adjoining paths, and the first just renames the length from one of these paths to distinguish it in the sum. The second statement goes on to find the least of all the sums that connect the same *org*–*dest* pair. The third statement is needed in case a newly-found path happens to have a shorter length than an earlier path that happened to have fewer edges.

The relational algebra to actualize all this must just do the renaming and the calculations in the right places. We start with one last renaming, to close the loop and give the result the same fields as the input, *ODL*, which we must first create.

```
ODL <- [org, dest, length] in Edges;
let length be length''';
TCL <- [org, dest, length] in
         [org, dest, length''] in
         (ODL union [org, dest, length] '' in
           (ODL[dest comp org] [org, dest, lenght'] in TCL)
         )
```

Here, *TCL* is initially equal to *ODL*, and the code has the following effect.

TCL(*org* *dest* *length'*)

org	dest	length'
$V1$	$V2$	4.0
$V2$	$V3$	2.0
$V2$	$V4$	5.0
$V3$	$V4$	2.0

result of comp

(*org*	*dest*	*length*	*length'*)	*length''*
$V1$	$V3$	4.0	2.0	6.0
$V1$	$V4$	4.0	5.0	9.0 $V2$

result of union

(*org*	*dest*	*length*	*length''*	*length'''*)
$V1$	$V2$	4.0		4.0
$V2$	$V3$	2.0		2.0
$V2$	$V4$	5.0	4.0	4.0
$V3$	$V4$	2.0		2.0
$V1$	$V3$		6.0	6.0
$V1$	$V4$		9.0	9.0
$V4$	2.0	2.0	4.0	

After a loop, or execution of the corresponding recursive view, the cost of 9.0 from $V1$ to $V4$ gets reduced to 8.0, and there is no further change. This code will find the all-points shortest path for a network of any size, with or without cycles.

Note that the **union** operator here transcends simple set union. The two non-matching fields, *length* from one side, and *length''* from the other, are recorded separately, with nulls (blanks) where the join fields from one side or the other do not match. The sum that creates *length'''* treats these nulls as the identity element, 0.

5.3.2 Updates

The domain algebra and all the relational algebra operators described so far, except for the assignment operator, are *functional*, in the technical sense that the same operands always give the same results, and there are no side-effects. Functional programming is very elegant and avoids most of the opportunities a programmer has for error in non-functional languages. It can be used for databases, however, only by copying entirely every relation one wishes to "change", and this is prohibitively expensive. So we need non-functional **update** operations, which have the side-effect of changing a relation in place.

SQL offers three separate commands for this, one for each of insert, change, and delete operations, with the insert command unfortunately differing from the other two by unfortunately working with only one tuple. It is cleaner to make them all relational and to avoid the ambiguity of the word, "update" by using it only to refer to relations, not tuples. It is also cleaner to isolate the update operations from the conditions determining which parts of the relation are affected. The T-selector and joins already discussed are ample to provide this control.

Here are the three cases of the **update** statement.

update R **add** S;
update R **delete** S;
update R **using** S **change** <statements>;

S is always a relation, and the **using** S clause (which is optional) in the change command uses the natural join of S with R to select the parts of R that will change. The <statements> in this case are usually assignment statements changing values of fields. They may contain domain algebra expressions on the right hand side. Of course, S may be any relational expression in all three cases. It may also be preceeded by a join operator, if simple **join** is not enough for the task.

5.4 Relations as Domains: Nested Relations

With the domain algebra and a cleaned-up relational algebra, we get nested relations for free (syntactically, that is: the implementation is more difficult, although everything we are about to say can be built with non-nested relations). The linguistic step is to subsume the relational algebra into the domain algebra.

Nested relations are relations whose field values may themselves be relations. Thus, to work with them, we need a formalism to create new relation-valued fields from existing ones. The domain algebra, with relational operations incorporated into it, can do this.

Although nested relations add nothing whatsoever to the *functionality* of the "flat" relational and domain algebras, which is what we have discussed so far, they do at times simplify our *thinking*, and so have a rightful place in secondary storage programming. They also repair an aesthetic inegality of the data types in the programming language: numbers, strings, and other scalar data types have hitherto had privileges relations have not, namely the ability to be included as values in relational tuples. Nesting gives relations these privileges, too.

Because in many instances nested relations offer no advantages over the relational algebra and, particularly, the domain algebra, as we have seen them so far, we must be careful in presenting a motivating example. We consider the two triangles of section 2, but we add a *colour* for each triangle. We could just extend the relation we used then

polygons(*id*	*seq*	*xcoord*	*ycoord*	*colour*)
F1	1	3.0	2.0	red
F1	2	1.0	2.0	red
F1	3	2.0	1.0	red
F2	1	3.0	2.0	blue
F2	2	2.0	3.0	blue
F2	3	1.0	2.0	blue

but normal relational habits would lead us to decompose this into two relations to avoid the effects of redundancy (update inconsistencies, for instance).

polygons(*id*	*seq*	*xcoord*	*ycoord*)
F1	1	3.0	2.0
F1	2	1.0	2.0
F1	3	2.0	1.0
F2	1	3.0	2.0
F2	2	2.0	3.0
F2	3	1.0	2.0

colours(*id*	*colour*)
F1	red
F2	blue

This decomposition is not in itself a problem, but subsequent operations, which might need the colours associated with the coordinates, would need a join to put the relations back together again. We may prefer to think about this in the context of only one relation.

So we express the data as a nested relation.

polygons				
(*id*	*structure*			*colour*)
	(*seq*	*xcoord*	*ycoord*)	
F1	1	3.0	2.0	red
	2	1.0	2.0	
	3	2.0	1.0	
F2	1	3.0	2.0	blue
	2	2.0	3.0	
	3	1.0	2.0	

Polygons now has only two tuples, and three fields: *structure*, which is itself a relation, and *id*, and *colour*.

As an example calculation with this, we project the inner relation on the coordinates alone and derive a new relation containing only this projection and the colour. This latter is also a projection. The important mental simplification offered us by the prescription that the domain algebra now includes the relational algebra is to isolate the two levels of the relation, and the two projections, from each other. Since the inner projection is on a field, albeit a field which is a relation, we do it in the domain algebra.

let *coords* **be** [*xccord, yccord*] **in** *structure*;

Then we use the ordinary relational algebra (but with a relation as one of the fields) to do the top-level projection.

colourCoord <− [*coords, colour*] **in** *polygons*;

colourCoord		
(*coords*		*colour*)
(*xcoord*	*ycoord*)	
3.0	2.0	red
1.0	2.0	
2.0	1.0	
3.0	2.0	blue
2.0	3.0	
1.0	2.0	

This result also has two tuples, and a relation-valued field, *coords*.

We can also use the domain algebra at both levels. Here is a count of the number of vertices in each polygon.

let *count* **be red** + **of** 1;
let *size* **be** [*count*] **in** *structure*;
polySizes <− [*id, size, colour*] **in** *polygons*;

This still has two tuples, and *size* is an inner relation even though it is a singleton in each tuple. To "raise" the level of its field, *count* is a simple matter, involving no new syntax. We just do not name the field; its anonymity will force the system to raise it. We rewrite the code.

let *size* **be** [**red** + **of** 1] **in** *structure*;
polySizes <− [*id, size, colour*] **in** *polygons*;

Now *size* has no field, and so can no longer be a nested relation. (It is important in doing this raising operation that the inner relation being raised is a singleton.)

polySizes(*id*	*size*	*colour*)
F1	3	red
F2	3	blue

Finally in this brief coverage of nested relations, we look at an aggregation which will be useful to us later. This finds the union of the coordinates in *colourCoord* and projects on them only.

let *unionCoord* **be red union of** *coords*;
aggregate <− [*unionCoord*] **in** *polygons*;

And, since the result, *aggregate*, is a singleton relation (with only the one tuple, containing the union of all the coordinates), we can rewrite this, raising the inner relation to an ordinary set of fields.

aggregate <− [**red union of** *coords*] **in** *polygons*;

aggregate(*xcoord*	*ycoord*)
3.0	2.0
1.0	2.0
2.0	1.0
2.0	3.0

5.5 Relational Programming

It is beyond the scope of this tutorial to take further the operations on relations and on fields that we have discussed. It is clear, though, that these operations are only the start of a programming language for secondary storage. They provide a formalism, much as arithmetic provides a formalism for working with numbers, to which must be added programming language considerations of procedural and data abstraction, scoping, type systems, etc. These can all be done by a careful program of examining the fundamental ideas of both fields, programming languages and databases, and putting them together. We do not discuss this here.

We only warn the reader of how not to do it, namely by the copout of getting only so far with the database language, then embedding calls to it in some other, entirely incompatible, programming language. This imposes on the programmer a conceptual mismatch which defeats the purpose of a good language. This purpose is to aid thinking, not hinder it. In the worst case, this mismatch can destroy the advantage of the database language itself for secondary storage, which is to abstract over looping and away from individual records and field values. SQL's dismally retrograde notion of a "cursor" is such a throwback.

Postscript. The limitations of SQL are by now abundantly clear, and we avoid SQL notation in the remainder of this tutorial. Support for our stance comes even from the heart of the relational community [19]:

> SQL involves so many inconsistencies, exceptions, and special cases that ... it [is] very difficult to see the forest for the trees.
>
> ... the official standard is not particularly easy to read. In places, in fact, it is well-nigh impenetrable.
>
> The current draft (encompassing both SQL3 and SQL4) is over 1500 pages long, while SQL/92 was "only" about one third that size.

Unfortunately, many who recognise these defects in SQL have also rejected all relational concepts and operations, thus throwing out the groundwork for a clean and elegant treatment of all secondary storage problems, including G.I.S.

Programming for secondary storage is far too important to be left entirely to committees.

6 Spatial Algorithms for Secondary Storage

Always program in the highest-level language, even if no implementation is on hand—Christopher Strachey

Since we cannot expound secondary storage techniques for all possible spatial algorithms, we pick two applications which are significant and which illustrate a variety of the methods available. *Polygon overlay* generates all possible polygons that result when two polygonal

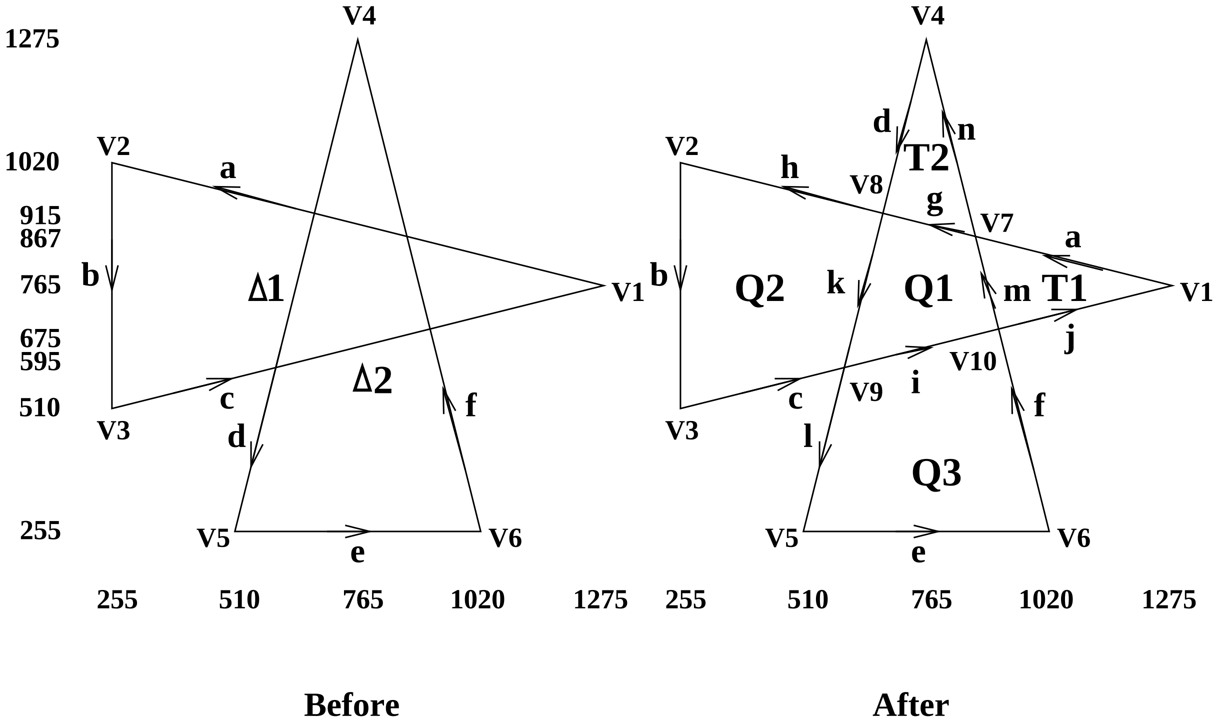


Figure 11: Source of the Overlay Example

subdivisions are superimposed. The *Delaunay triangulation* and its dual, the *Voronoi diagram*, identify neighbours of vertices, polygons whose interiors are nearer the given vertices than are any other points.

The G.I.S. user would expect to find the two constuctions that we discuss in detail below already built and available. We are not advocating that hey now be required to code them from scratch. We are illustrating the spatial capabilities of the database programming language. The G.I.S. user should find these algorithms, already coded, in a library. This library must be compatible with any programming we may want to do for non-spatial values and so the interfaces must use the general secondary-storage language. It is best if the construction also uses this language.

In the following discussion, we work with simple examples. Once the program is written for the simple example (assuming all exceptional cases, such as vertical lines or collinear vertices, have been accounted for), it applies to all other cases. This is an advantage of abstracting over loops. We do not give full program code below, although the reader will see that the programs are short.

6.1 Polygon Overlay

The simple example for polygon overlay is the two triangles in figure 11 (Before). This figure shows coordinates along the vertical axis. There are also some coordinates along the horizontal axis: the figure is x–y symmetrical, so the remaining horizontal coordinates may be inferred.

$TwoTriangles(edge1$	$dir1$	$edge2$	$dir2)$
a	0	c	2
c	2	a	0
b	0	a	2
a	2	b	0
c	0	b	2
b	2	c	0
d	0	f	2
f	2	d	0
e	0	d	2
d	2	e	0
f	0	e	2
e	2	f	0
a	3	b	3
b	3	c	3
c	3	a	3
d	3	e	3
e	3	f	3
f	3	d	3
c	1	b	1
b	1	a	1
a	1	c	1
f	1	e	1
e	1	d	1
d	1	f	1

$VertFace(edge$	org	$dest$	$left$	$right)$
a	$V1$	$V2$	$\Delta 1$	$U1$
b	$V2$	$V3$	$\Delta 1$	$U1$
c	$V3$	$V1$	$\Delta 1$	$U1$
d	$V4$	$V5$	$\Delta 2$	$U2$
e	$V5$	$V6$	$\Delta 2$	$U2$
f	$V6$	$V4$	$\Delta 2$	$U2$

$Geom(vf$	x	$y)$
$V1$	1275	765
$V2$	255	1020
$V3$	255	510
$V4$	675	1275
$V5$	510	255
$V6$	1020	255

$\Delta 1$
$\Delta 2$
$U1$
$U2$

Figure 12: Data Structure for the Overlay Example

We must find which edges intersect and where, generate vertices and edges corresponding to these intersections, and then find the set of polygons corresponding to the overlay. To save space, we do not go on to find the subset of these polygons corresponding to G.I.S. procedures such as erase, identity, intersect, or union, nor do we discuss edge elimination between adjacent polygons, or operations such as update. These follow in the same manner.

The data structure for the two polygons is in figure 12. We see that the two triangles are initially considered to be on separate manifolds: thus we have two triangular faces, $\Delta 1$ and $\Delta 2$, and two complementary faces, $U1$ and $U2$. For this example, we record no coordinates or other data for these four faces.

6.1.1 Intersecting Edges

To find which edges intersect and where, the easiest approach is to take the Cartesian product (a special case of **join**) of the edges, a, b, and c, from $\Delta 1$ with the edges, d, e, and f, from $\Delta 2$. We can use a construction on two edges to see if they intersect, and then do linear interpolation between the endpoints to find out where.

The intersection test for two finite edges described by vertices o_1, d_1 and o_2, d_2, respectively, checks the two pairs of triangles, o_1, d_1, o_2 versus o_1, d_1, d_2, and o_2, d_1, o_1 versus o_2, d_2, d_1. If the triangles in a pair have opposite directions (e.g., clockwise versus counter-clockwise), and if this is true for both pairs, then the edges intersect somewhere between their endpoints. Since the area of a triangle is positive if the vertices are processed counterclockwise, and negative if clockwise, this test just compares signs of areas:

$$area(o_1, d_1, o_2) \times area(o_1, d_1, d_2) \leq 0$$
and
$$area(o_2, d_2, o_1) \times area(o_2, d_2, d_1) \leq 0$$

We showed in section 5.2.1 how to find areas. If any two of the four areas equal zero, the segments are collinear; to tell if they coincide, a further test is needed to detect if an endpoint of one segment lies between the endpoints of the other.

The linear interpolation finds equations for the edges,

$$y - slope \times x = constant$$

and uses the slope for, say, edge o_1, d_1 to find the values of the constant for parallel lines passing through o_2 and through d_2. (This constant is the y-intercept of the extrapolated edge.) The interpolation fraction can be found using these two constants and the original constant for edge o_1, d_1. This fraction gives the interpolated coordinates of the intersection point. Here is the domain algebra.

let s **be** $(yd1 - yo1)/(xd1 - xo1)$; //slope
let $c1$ **be** $yo1 - s \times xo1$; //constant o1–d1
let $co2$ **be** $yo2 - s \times xo2$; //constant o2
let $cd2$ **be** $yd2 - s \times xd2$; //constant d2
let $f2$ **be** $(c1 - co2)/(cd2 - co2)$; //fraction
let x **be** $xo2 + (xd2 - xo2) \times f2$; //x-intersect
let y **be** $yo2 + (yd2 - yo2) \times f2$; //y-intersect

We leave it as an exercise to the reader to do the join, intersection test, interpolation, and all the necessary renaming. The result we need is

$e1$	$xo1$	$yo1$	$xd1$	$yd1$	s	$c1$	$e2$	$xo2$	$yo2$	$xd2$	$yd2$	$co2$	$cd2$	$f2$	x	y
a	1275	765	255	1020	$-1/4$	17	d	765	1275	510	255	23	6	6/17	675	915
a	1275	765	255	1020	$-1/4$	17	f	1020	255	765	1275	8	23	3/5	867	867
c	255	510	1275	765	1/4	7	d	765	1275	510	255	17	2	2/3	595	595
c	255	510	1275	765	1/4	7	f	1020	255	765	1275	0	17	7/17	915	675

(To keep the entries simple, we have used smallest integers for $c1, co2$, and $cd2$, which are used only in ratios.)

(The calculation is complicated by a case not shown here, in which one of the intersecting edges is a vertical line and has infinite slope. The calculation must be shifted to the other edge, which necessarily has finite slope.)

We started this calculation with a Cartesian product of every edge on one side by every edge on the other. This has quadratic complexity, and is not asymptotically optimal (although a better algorithm might not beat it in practice on secondary storage unless the language implementation is very sophisticated). We now discuss briefly the *plane sweep* algorithm for detecting edge intersections by sorting the endpoints of the edges on x-coordinate within y-coordinate, hence incurring only $\mathcal{O}(n \log n)$ complexity.

We follow the treatment of de Berg et al. [20] (p.25), using an "event queue", Q, which stores endpoint coordinates of the edges in decreasing order of y, and a temporary workspace, T, which stores records from Q in increasing order of x until two endpoints of the same edge can be matched up and tested for crossing any other edge.

We must address the issue of "store .. in decreasing order of y" in the relational context. We cannot explicitly sort a relation, but we can use functional mapping from the domain algebra to induce the ordering we need. (Descending order can be induced by using a minus sign, e.g., **order** $-y$.) We also use **red max** and **equiv max** to extract the current largest value. To resolve different points with the same y value, we refine the ordering to include increasing values of x within decreasing values of y. The code begins

```
while [] in Q                          //Q is not empty
{  event <-  where (y=red max of y)
             and (x=equiv min of x by y)
             in Q;
   update Q delete event;
```

We here introduce the *empty projection list*, []. This produces a *nullary* relation, one with no fields. Such a relation is interpreted as a boolean value, using the rather abstract claim that a nullary relation can have only two states, empty, and not empty, and making the convention that the former is interpreted as false and the latter as true. The form, [] **in** Q, can be pronounced "something in Q".

$Q(e, x, y, sie, x', y')$ holds the name, e of the edge, its starting vertex, (x, y), its ending vertex, (x', y') ($y \geq y'$, and for horizontal lines, $x < x'$), and the flag, *sie*, which is `s` if (x, y) starts the edge, `e` if (x, y) ends the edge (in which case, (x', y') are null), and `i` if (x, y) is an intersection point discovered in the course of the algorithm.

Here is Q before the first step, in the case of our two triangles. These embody some special cases, such as vertical and horizontal edges and shared endpoints.

$Q(e$	x	y	*sie*	x'	$y')$
d	765	1275	s	510	255
f	765	1275	s	1020	255
a	255	1020	s	1275	765
b	255	1020	s	255	510
a	1275	765	e		
c	1275	765	s	255	510
b	255	510	e		
c	255	510	e		
d	510	255	e		
e	510	255	s	1020	255
e	1020	255	e		
f	1020	255	e		

The loop continues with just about one statement per sentence from the algorithm given on p.26 of [20]:

```
p <-[x,y] in event;                                  // Not used, but connects to [20]
U <-[e,x,y,x',y'] where sie="s" in event;            // segments with upper end at p
C <-[e,x,y,x',y'] where sie="i" in event;            // .. with p as intersection point
L <-[e,x,y,x',y'] in (event[x,y comp x',y'] [x',y'] in T);   // segments with lower end at p
UC <-U union C;
UCL <- UC union L;
let ct be red + of 1;
OUT <+[e,x,y] where ct>1 in UCL;                     // Newly found intersection point
update T delete L;                                   // remove segments ending
update T add U;                                      // insert segments starting
```

Here are Q, *event*, U, C, and L after the first iteration.

$Q(e$	x	y	*sie*	x'	$y')$
a	255	1020	s	1275	765
b	255	1020	s	255	510
a	1275	765	e		
c	1275	765	s	255	510
b	255	510	e		
c	255	510	e		
d	510	255	e		
e	510	255	s	1020	255
e	1020	255	e		
f	1020	255	e		

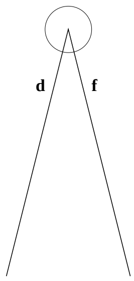
event$(e$	x	y	*sie*	x'	$y')$
d	765	1275	s	510	255
f	765	1275	s	1020	255

$U(e$	x	y	x'	$y')$
d	765	1275	510	255
f	765	1275	1020	255

$C(e$	x	y	x'	$y')$

$L(e$	x	y	x'	$y')$

This iteration reports (in OUT) `d` and `f` as edges intersecting at point (`765, 1275`). We knew this already, but the algorithm is formulated to report all intersections of every pair of edges; a later pass will filter out the significant intersections between the two different edge sets we might be interested in.



$OUT(e$	x	$y)$
d	765	1275
f	765	1275

Finally, T is

$T(e$	x	y	x'	$y')$
d	765	1275	510	255
f	765	1275	1020	255

No new intersection point must be computed in this iteration, so we defer discussion of that part of the code, and return to the top of the loop. Here are the results of the second iteration.

$Q(e$	x	y	*sie*	x'	$y')$
a	1275	765	e		
c	1275	765	s	255	510
b	255	510	e		
c	255	510	e		
d	510	255	e		
e	510	255	s	1020	255
e	1020	255	e		
f	1020	255	e		

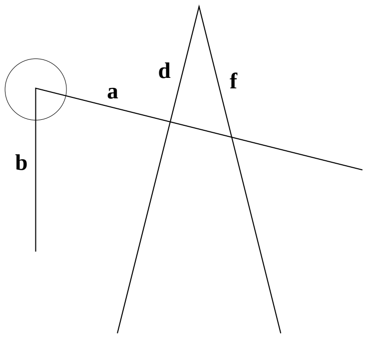
event$(e$	x	y	*sie*	x'	$y')$
a	255	1020	s	1275	765
b	255	1020	s	255	510

$U(e$	x	y	x'	$y')$
a	255	1020	1275	765
b	255	1020	255	510

$C(e$	x	y	x'	$y')$

$L(e$	x	y	x'	$y')$

Again, in OUT there is an intersection which is not new.



OUT(e	x	y)
:	:	:
a	255	1020
b	255	1020

This time, T is

T(e	x	y	x'	y')
b	255	1020	255	510
a	255	1020	1275	765
d	765	1275	510	255
f	765	1275	1020	255

And we will find that the tuples for a and d, which are adjacent in the horizontal ordering on x and x', intersect at a new point, so we now discuss the code to find this new point.

The code to detect intersecting edges and to find the intersection point was essentially given above when we looked at the Cartesian product method, and we do not repeat it. A simplistic formulation of the algorithm just checks the predecessor and the successor of every record in T, ordered on x' within x. Here is the domain algebra statement giving the predecessor of the x coordinate.

let *px* **be fun pred of** x **order** x, x';

The other three predecessors are similarly found, then the intersection coordinates, *xp* and *yp*, are found (null if the edge does not intersect with its predecessor). The data that will be added to Q is finally renamed and assigned to *newQ*.

let x **be** *xp*;
let y **be** *yp*;
let *sie* **be** `"i"`;
newQ <−[e, x, y,*sie*, x', y'] **in** [*e*,*xp*,*yp*,*sie*,x', y'] **in** T;

The same process checks the succesor of each record in T for intersection at (*xs*,*ys*), concluding

newQ <+ [*e*,*x*,*y*,*sie*, x', y'] **in** [*e*,*xs*,*ys*,*sie*,x', y'] **in** T;

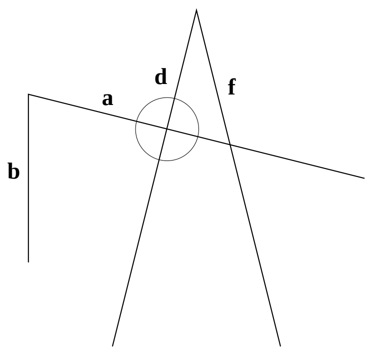
As a result, *newQ* contains modified tuples for edges a and d, with x and y now the coordinates of the intersection point, instead of the upper point.

newQ(e	x	y	*sie*	x'	y')
a	675	915	`i`	1275	765
d	675	915	`i`	510	255

We terminate the loop by updating Q to add the new intersection point(s).

update Q **add** *newQ*;
}

Thus, when the loop enters the third iteration, Q has the two new tuples from *newQ*. After this iteration, *OUT* reports the first new intersection.



$OUT(e$	x	$y)$
:	:	:
a	675	915
d	675	915

At the end of the loop, *OUT* will contain all intersections. Here are the new ones.

$OUT(e$	x	$y)$
a	675	915
d	675	915
a	867	867
f	867	867
f	915	675
c	915	675
d	595	595
c	595	595

We should note that the loop contains four **update** statements, which each delete or add only one or two tuples. This can render the loop, which is clearly over all tuples, very expensive for secondary storage: on secondary storage, we should avoid repeated accesses to the same block. Hence our point, above, that the quadratic method which takes the Cartesian product might be better. However, a good *implementation* will sort the relations as the domain algebra indicates.

The above code, which is almost as short as the abstract specification of the algorithm it implements, is still a slight compromise between efficiency for secondary storage and optimal code for the algorithm as specified for RAM. Instead of processing every tuple in T, we need only compare the first and last records of *UC* with their predecessors and successors, respectively, in T (or, if *UC* is empty, then the predecessor and successor records in T of C **union** L, which was deleted from T). To do this requires more code and possibly more accesses to secondary storage. It is easily done, though, if we want: use the original Q to give search ranges for any neighbouring record in T, construct an index field for T to give the ordering, and use the relational algebra to process singleton relations, i.e., individual records.

6.1.2 Generate Vertices, Edges, and Splices

We have now found the following edge intersections.

$e1$	$e2$	x	y	$edgeGeom(e$	xo	yo	xd	$yd)$
a	d	675	915	a	1275	675	255	1020
a	f	867	867	c	255	510	1275	675
c	d	595	595	d	675	1275	510	255
c	f	915	675	f	1020	255	675	1275

This also shows the geometry for the relevant old edges, for reference.

Since there are four tuples, we must generate four new vertices, say, $V8, V7, V9$, and $V10$, respectively (the labels, and their order in particular, do not matter, but we keep the counterclockwise order shown in figure 11). Since each edge appears twice in the result, we must generate eight new edges, say, g, h, i, j, k, l, m, and n. Finally, we must decide how to associate these new edges with the old, and with the new vertices. We replace each old edge by a triplet: $a \rightarrow (a, g, h), c \rightarrow (c, i, j), d \rightarrow (d, k, l), f \rightarrow (f, m, n)$. Then we can use the geometry to string the vertices into these triplets: $(a, g, h) \rightarrow (a, V7, g, V8, h), (c, i, j) \rightarrow (c, V9, i, V10, j)$, $(d, k, l) \rightarrow (d, V8, k, V9, l), (f, m, n) \rightarrow (f, V10, m, V7, n)$. (This latter step notices, for instance, that the x-coordinates related to a go, from xo to xd, 1275, 867, 765, 255, and so the internal vertices are ordered from origin to destination of a as $V7$ then $V8$.)

All this gives us the data structure shown in figure 13 for the new, still disconnected, edges. (We have put them all on a single, new manifold, with face $U3$, which we can get away with since disconnected edges form no new faces.)

Our final activity in this section is to generate all the splices needed to construct the overlay of the two triangles, now that we have the new vertices and edges. First, we must disconnect the old edges, a, c, d, and f, that have been broken up, at their destination vertices, since we have decided to add the two new edges for each before these destinations. The splice pairs are

$$((a, 2), (b, 0)), ((c, 2), (a, 0)), ((d, 2), (e, 0)), \text{ and } ((f, 2), (d, 0)).$$

Next, we must connect the new edges with these destinations to the edges we just disconnected from. Splice pairs:

$$((h, 2), (b, 0)), ((j, 2), (a, 0)), ((l, 2), (e, 0)), \text{ and } ((n, 2), (d, 0)).$$

Finally, we must do the succession of splices that connect the edges around the four new vertices. We can add edges one at a time to a starting edge: a total of three splices per vertex.

Because each *splice*(p, q) adds the vertex cycle containing q counterclockwise *after* p in its vertex cycle (and the vertex cycle containing p counterclockwise *after* q in its vertex cycle), we must be sure to perform these connections in counterclockwise order, so we need to know what this is at each vertex. An edge, q, is counterclockwise from edge p at the same vertex if the angle from p to q is positive.

$V7$, for instance, is the destination of a and m, and the origin of g and n. So we must arrange $(a, 2), (g, 0), (m, 2)$, and $(n, 0)$ in counterclockwise order before connecting them. That is, we must arrange the origins of a and m, and the destinations of g and n, in counterclockwise order. If we start with the original edge, a, then its new continuation, g, then follow with m then n, we have the sequence

$$a.org,\ g.dest,\ m.org,\ n.dest,$$

where we have abbreviated vertices from *VertFace* in figures 12 and 13 using the *org* and *dest* fields. Naming the vertices directly, this is the sequence

$$V1, V8, V10, V4.$$

NewEdges(*edge1*	*dir1*	*edge2*	*dir2*)
g	0	g	0
g	2	g	2
h	0	h	0
h	2	h	2
i	0	i	0
i	2	i	2
j	0	j	0
j	2	j	2
k	0	k	0
k	2	k	2
l	0	l	0
l	2	l	2
m	0	m	0
m	2	m	2
n	0	n	0
n	2	n	2
g	1	g	3
g	3	g	1
h	1	h	3
h	3	h	1
i	1	i	3
i	3	i	1
j	1	j	3
j	3	j	1
k	1	k	3
k	3	k	1
l	1	l	3
l	3	l	1
m	1	m	3
m	3	m	1
n	1	n	3
n	3	n	1

VertFace(*edge*	*org*	*dest*	*left*	*right*)
g	$V7$	$V8$	$U3$	$U3$
h	$V8$	$V2$	$U3$	$U3$
i	$V9$	$V10$	$U3$	$U3$
j	$V10$	$V1$	$U3$	$U3$
k	$V8$	$V9$	$U3$	$U3$
l	$V9$	$V5$	$U3$	$U3$
m	$V10$	$V7$	$U3$	$U3$
n	$V7$	$V4$	$U3$	$U3$

Geom(*vf*	x	y)
$V7$	867	867
$V8$	675	915
$V9$	595	595
$V10$	915	675
$U3$		

Figure 13: Data Structure for the New Edges and Vertices

To make this a counterclockwise sequence of vertices,

$$V1, V4, V8, V10.$$

we need only move $V4$ from last to second position.

This same process applies to each of the crossing lines, because of the counterclockwise direction of the original edges, a, b, c and d, e, f around the two triangles. Thus

$$V1, V8, V10, V4 \rightarrow V1, V4, V8, V10$$

$$V4, V9, V7, V2 \rightarrow V4, V2, V9, V7$$
$$V3, V10, V8, V5 \rightarrow V3, V5, V10, V8$$
$$V6, V7, V9, V1 \rightarrow V6, V1, V7, V9$$

In terms of edges, the final sequences are

$$(a, 2), (n, 0), (g, 0), (m, 2)$$

$$(d, 2), (h, 0), (k, 0), (g, 2)$$
$$(c, 2), (l, 0), (i, 0), (k, 2)$$
$$(f, 2), (j, 0), (m, 0), (i, 2)$$

and so the splices we need are

$$((a, 2), (n, 0)), ((n, 0), (g, 0)), ((g, 0), (m, 2))$$

$$((d, 2), (h, 0)), ((h, 0), (k, 0)), ((k, 0), (g, 2))$$
$$((c, 2), (l, 0)), ((l, 0), (i, 0)), ((i, 0), (k, 2))$$
$$((f, 2), (j, 0)), ((j, 0), (m, 0)), ((m, 0), (i, 2))$$

6.1.3 Update the Data Structure

Combining these twelve splices with the eight, above, that precede them, we get twenty-four substitutions (before), in four cycles of six, for the vertices,

$$\begin{array}{c|c|c|c}
(a,2) \rightarrow (h,2) & (c,2) \rightarrow (j,2) & (d,0) \rightarrow (l,2) & (f,2) \rightarrow (n,2) \\
(h,2) \rightarrow (b,0) & (j,2) \rightarrow (a,0) & (k,2) \rightarrow (e,0) & (n,2) \rightarrow (d,0) \\
(b,0) \rightarrow (m,2) & (a,0) \rightarrow (k,2) & (e,0) \rightarrow (g,2) & (d,0) \rightarrow (i,2) \\
(m,2) \rightarrow (g,0) & (k,2) \rightarrow (c,0) & (g,2) \rightarrow (k,0) & (i,2) \rightarrow (m,0) \\
(g,0) \rightarrow (n,0) & (c,0) \rightarrow (l,0) & (k,0) \rightarrow (h,0) & (m,0) \rightarrow (j,0) \\
(n,0) \rightarrow (a,2) & (l,0) \rightarrow (c,2) & (h,0) \rightarrow (d,2) & (j,0) \rightarrow (f,2)
\end{array}$$

and a further twenty-four (after), for the faces

$$\begin{array}{c|c|c|c}
(a,1) \rightarrow (h,1) & (c,1) \rightarrow (j,1) & (d,3) \rightarrow (l,1) & (f,1) \rightarrow (n,1) \\
(h,1) \rightarrow (b,3) & (j,1) \rightarrow (a,3) & (k,1) \rightarrow (e,3) & (n,1) \rightarrow (d,3) \\
(b,3) \rightarrow (m,1) & (a,3) \rightarrow (k,1) & (e,3) \rightarrow (g,1) & (d,3) \rightarrow (i,1) \\
(m,1) \rightarrow (g,3) & (k,1) \rightarrow (c,3) & (g,1) \rightarrow (k,3) & (i,1) \rightarrow (m,3) \\
(g,3) \rightarrow (n,3) & (c,3) \rightarrow (l,3) & (k,3) \rightarrow (h,3) & (m,3) \rightarrow (j,3) \\
(n,3) \rightarrow (a,1) & (l,3) \rightarrow (c,1) & (h,3) \rightarrow (d,1) & (j,3) \rightarrow (f,1)
\end{array}$$

The first twenty-four will be produced as a relation,

$$\begin{array}{rccc} subst(e & d & e' & d') \\ a & 2 & h & 2 \\ h & 2 & b & 0 \\ : & : & : & : \end{array}$$

and the second twenty-four can be derived from this by reducing d and d' by 1 modulo 4,

$$\begin{array}{cccc} a & 1 & h & 1 \\ h & 1 & b & 3 \\ : & : & : & : \end{array}$$

Applying these substitutions to figures 12 and 13, we get the final data structure for the overlaid triangles shown in figure 14. The faces are labelled for ease of identification: Qi for the three quadrilateral faces, and Ti for the two new triangles.

The relational code for generating the vertex and face topology in *Overlaid*Δ is based on join of the source relations, *TwoTriangles* (figure 12) and *NewEdges* (figure 13), with *subst* and with *subst* modified by domain algebra. Since not every tuple is changed (although most of them are for an overlay: in the example, only 8 of 56 tuples are not changed by the substitutions), it is (perhaps) best to do an update.

First, we separate the vertex topology from the face topology in the combined source relations

VertTopol <− **where** $d1$ **mod** $2 = 0$ **in** *TwoTriangles*
union
where $d1$ **mod** $2 = 0$ **in** *NewEdges*;
FaceTopol <− **where** $d1$ **mod** $2 = 1$ **in** *TwoTriangles*
union
where $d1$ **mod** $2 = 1$ **in** *NewEdges*;

Next, we apply *subst* to the vertex topology, inserting *before*

update *VertTopol* **using** $[e1, d1$ **join** $e, d]$ *subst* **change** $e1 <\!-e'; d1 <\!-d'$;

and we apply the modified *subst* to the face topology, inserting *after*

let D **be if** $d = 0$ **then** 3 **else** $d - 1$;
let D' **be if** $d' = 0$ **then** 3 **else** $d' - 1$;
update *FaceTopol* **using** $[e2, d2$ **join** $e, D]$ *subst* **change** $e2 <\!-e'; d2 <\!-D'$;

The result is the union of these updated topologies.

OverlaidΔ <− *VertTopol* **union** *FaceTopol*;

We have worked this simple example of two triangles thoroughly. The relational code follows our workings, and is then applicable to any overlay of two subdivisions with any number of polygons or arcs.

Postscript. Considered as an operator, overlay takes two subdivisions (usually sets of connected polygons) and produces a subdivision. The operator is commutative and associative: a redop. Thus it can be coded as an abstract data type on nested relations, and used for aggregation in domain algebra **red** and **equiv** expressions.

The *spatialUnion* operation, which is overlay followed by removal of the internal edges, is also a redop and can also be aggregated by **red** and **equiv** expressions. So are *intersection, symmetric difference,* etc.

6.2 Map Overlay

Geography differs from geometry in that it associates non-spatial fields ("attributes" in GIS terminology) with spatial figures. For example, the triangles in section 5.4 might be coloured.

$Overlaid\Delta(edge1$	$dir1$	$edge2$	$dir2)$
a	0	j	2
j	2	a	0
b	0	h	2
h	2	b	0
c	0	b	2
b	2	c	0
d	0	n	2
n	2	d	0
e	0	l	2
l	2	e	0
f	0	e	2
e	2	f	0
g	0	m	2
m	2	a	2
a	2	n	0
n	0	g	0
h	0	k	0
k	0	g	2
g	2	d	2
d	2	h	0
i	0	k	2
k	2	c	2
c	2	l	0
l	0	i	0
j	0	m	0
m	0	i	2
i	2	f	2
f	2	j	0
a	1	j	1
j	1	f	1
f	1	e	1
e	1	l	1
l	1	c	1
c	1	b	1
b	1	h	1
h	1	d	1
n	1	a	1
g	1	n	3
n	3	d	3
d	3	g	1
i	1	l	3
l	3	e	3
e	3	f	3
f	3	i	1
k	1	h	3
h	3	b	3
b	3	c	3
c	3	k	1
m	1	j	3
j	3	a	3
a	3	m	1
g	3	k	3
k	3	i	3
i	3	m	3
m	3	g	3

$VertFace(edge$	org	$dest$	$left$	$right)$
a	$V1$	$V7$	$T1$	$U1$
b	$V2$	$V3$	$Q1$	$U1$
c	$V3$	$V9$	$Q1$	$U1$
d	$V4$	$V8$	$T2$	$U1$
e	$V5$	$V6$	$Q2$	$U1$
f	$V6$	$V10$	$Q2$	$U1$
g	$V7$	$V8$	$Q3$	$T2$
h	$V8$	$V2$	$Q1$	$U1$
i	$V9$	$V10$	$Q3$	$Q2$
j	$V10$	$V1$	$T1$	$U1$
k	$V8$	$V9$	$Q3$	$Q1$
l	$V9$	$V5$	$Q2$	$U1$
m	$V10$	$V7$	$Q3$	$T1$
n	$V7$	$V4$	$T2$	$U1$

$Geom(vf$	x	$y)$
$V1$	1275	765
$V2$	255	1020
$V3$	255	510
$V4$	765	1275
$V5$	510	255
$V6$	1020	255
$V7$	867	867
$V8$	675	915
$V9$	595	595
$V10$	915	675

$T1$			
$T2$			
$Q1$			
$Q2$			
$Q3$			
$U1$ d	1	n	1

Figure 14: Final, Overlaid, Data Structure

Although nested relations can be useful, in this section we do not use them, but simply add fields to the *Geom* relation in the quad-edge representation.

Maps are also represented in *layers*, each specializing in a different aspect of the geography, such as topography, waterways, roads, political boundaries, agricultural products, climate, etc. The map overlay example we discuss in this section considers the horizontal triangle ($\Delta 1$) (see figure 11) of the previous section to be one element of a triangulated irregular network (TIN) in a layer giving the heights of land, and the vertical triangle ($\Delta 2$) to be an element of a climate layer giving temperatures. We are interested in overlaying these to study correlations between altitude and temperature.

We complicate the problem, for interest, by associating heights with the *vertices* of the subdivision in the topographic layer, and temperatures with the *faces* in the thermal layer. The *Geom* relations of the quad-edge representation initially are

GeomTopog(vf	*x*	*y*	*h)*	*GeomTherm(vf*	*x*	*y*	*t)*
$V1$	1275	765	50	$V4$	765	1275	
$V2$	255	1020	305	$V5$	510	255	
$V3$	255	510	254	$V6$	1020	255	
$\Delta 1$	595	765		$\Delta 2$	765	595	20
$U1$	∞	∞		$U2$	∞	∞	

The coordinates for the faces, $\Delta 1$ and $\Delta 2$, are those of the centroids. These are found using Stokes' theorem (section 3.4). We will also need Stokes' theorem to find the *gradients*, so we outline the results here. First, we must combine *Geom1* with the corresponding *VertFace* relation from the quad-edge representation to give *edgeSet*.

edgeSet(edge	*Xorg*	*Yorg*	*Horg*	*Xdest*	*Ydest*	*Hdest)*
a	1275	765	50	255	1020	305
b	255	1020	305	255	510	254
c	255	510	254	255	765	50

Then the four quantities we shall need (plus the area, which we repeat from section 3.4) are

quantity	*integral*	*domain algebra*
area, A	$\iint_A dx\,dy$	**red + of** *Xorg*× *Ydest* - *Yorg*× *Xdest*
x-centroid, $\overline{x}$	$\iint_A x\,dx\,dy$	**red + of** (*Xdest*∧2 + *Xdest*× *Xorg* + *Xorg*∧2)×(*Ydest* − *Yorg*)/*area*
y-centroid, $\overline{y}$	$\iint_A y\,dx\,dy$	**red + of** (*Ydest*∧2 + *Ydest*× *Yorg* + *Yorgwedge*2)×(*Xdest* − *Xorg*)/*area*
x-gradient, $\overline{h_x}$	$\iint_A \partial h/\partial x\,dx\,dy$	**red + of** (*Hdest* + *Horg*)× (*Ydest* − *Yorg*)/2/*area*
y-gradient, $\overline{h_y}$	$\iint_A \partial h/\partial y\,dx\,dy$	**red + of** (*Hdest* + *Horg*)× (*Xdest* − *Xorg*)/2/*area*

Using these, we calculate the two components of the mean gradient for $\Delta 1$, which we can subsequently use to find the mean height of $\Delta 1$ and of any polygon with the same gradient. The mean height is the height of the centroid of the polygon, as computed using the mean gradient.

$$h(x,y) = h(x_1,y_1) + \overline{h_x} \times (\overline{x} - x_1) + \overline{h_y} \times (\overline{y} - y_1)$$

for any point, (x_1, y_1), whose height is related to the height of the polygon, e.g., one of the vertices.

The relevant parts of the *Geom* relations are

GeomTopog(vf	*x*	*y*	$\overline{h_x}$	$\overline{h_y}$	*h)*	*GeomTherm(vf*	*x*	*y*	*t)*
$\Delta 1$	595	765	$-459/8/255$	$51/2/255$	203	$\Delta 2$	765	595	20

These preliminaries convert the topographic layer to one with the height field associated with the faces. We can now execute the overlay of the previous section to get combined faces with two fields each, h and t. (Actually, we keep the two components of the gradient as well, because they are used to find the mean heights of the faces.)

$Geom(vf$	x	y	$\overline{h_x}$	$\overline{h_y}$	h	$t)$
T1	1016.12	723.61	−459/8/255	51/2/255	104.1	20.6
T2	723.61	1016.12				20
Q1	752.33	752.33	−459/8/255	51/2/255	166.33	20
Q2	431.56	769.3	−459/8/255	51/2/255	240.2	19.3
Q3	769.3	431.56				20

We see that the thermal value, 20, is associated with the mean height, 166.33. In the above we have cheated by introducing two new thermal values, 20.6 and 19.8: to give ourselves more than one thermal value to correlate with the three mean heights, without complicating the geometry of our simple example, we have supposed that a temperature of 20.6 was measured to the right of $\Delta 2$, and that a temperature of 19.3 was measured to the left. Now we can go on to show a negative correlation of temperature with mean height if we want, a drop of approximately one Fahrenheit degree for 200 feet of altitude.

6.3 Delaunay and Voronoi Diagrams

A set of vertices has a Voronoi diagram which is a subdivision of the plane into polygons, each containing all points nearer to one vertex than to any other. Figure 15 shows this diagram (dashed lines) for the four vertices discussed in section 4.2. (We will change the coordinates to suitable integers.) The solid lines in this figure are the *triangulation* of the four vertices, called the Delaunay triangulation, that is dual to the Voronoi diagram. The faces shown pertain to the triangulation: the Voronoi diagram has four faces corresponding to the four vertices.

We will find the Delaunay triangulation first, then proceed to its dual, the Voronoi diagram. We follow the dynamic algorithm given by Guibas and Stolfi [27], which supposes that the Delaunay triangulation has been found for a set of points, to which one more point is to be added and the new triangulation found. This algorithm has two parts: finding the triangular face that contains the new point (a special point-in-polygon search); and then adjusting this face so the new point is included in the triangulation. (The overall cost of adding n points this way is $\mathcal{O}(n^2)$, since the point-in-polygon search is linear and must be repeated n times. This is more expensive than the divide-and-conquer algorithm for the static problem in which all n points are initially known.)

In figure 15, we see that the horizontal edge connecting $V1$ with $V2$ could equally be replaced with a vertical edge connecting $V3$ and $V4$: the Delaunay triangulation is not, in this example, unique. If $V3$ and $V4$ were a little closer together, the vertical edge, not the horizontal one, would be in the Delaunay triangulation, which is defined to consist of triangles with the largest possible internal angles. A Delaunay edge connects two vertices if the smallest circle containing those vertices contains no other vertex.

The process of changing this edge from horizontal to vertical we call *swapping*.

When we insert a new vertex, $V5$, in the diagram, we connect it to each of the vertices, $V1, V2$, and $V3$, of the triangle it lands in. Then we must check the opposite edge in each of these new triangles to see if it should be swapped in order to meet the smallest-circle test. Of these three opposite edges, a, b, and c, only a must be swapped, as we see in the *After* version in figure 15.

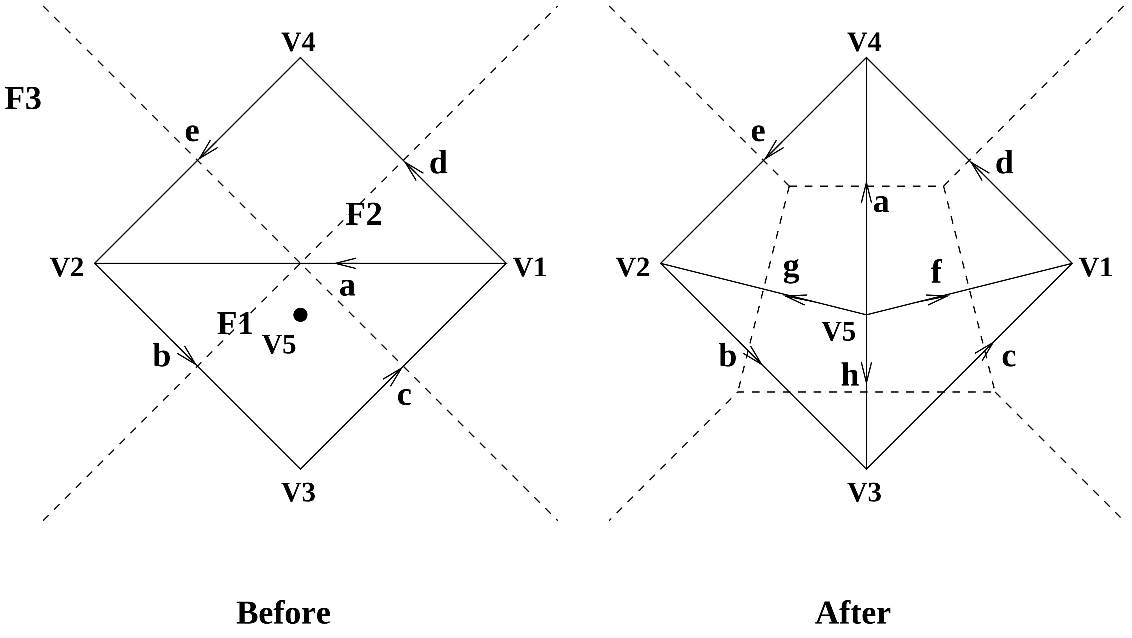


Figure 15: Voronoi and Delaunay Diagrams: Inserting V5

We also see that the Voronoi edges are the right bisectors of the Delaunay edges, as must be the case to satisfy the definition of the Voronoi diagram.

6.3.1 Point-in-Triangle

Finding which triangle contains a vertex is a special case of finding which convex polygon contains the point, and the more general problem is as easy to code. If every angle subtended at the vertex by the edges is non-negative, the vertex is in the polygon or on its boundary, otherwise it is not. (This is equivalent to the assertion that the vertex is always to the left if we negotiate the convex polygon counterclockwise: the mathematics is identical.)

The sign of the angle is the sign of the determinant, $x_1y_2+x_2y_p+x_py_1-x_1y_p-x_2y_1-x_py_2$, formed by the edge, $((x_1,y_1),(x_2,y_2))$, and the vertex, (x_p,y_p), where the edge direction is counterclockwise around the polygon. (This determinant is twice the area of the triangle formed by these three vertices, positive if the vertex order is counterclockwise.)

The relation for the example can be obtained by relational algebra from the data structure for the original diagram.

pip(*face*	x_1	y_1	x_2	y_2	x_p	y_p)	*det*
$F1$	18	10	2	10	10	8	32
$F1$	2	10	10	2	10	8	48
$F1$	10	2	18	10	10	8	48
$F2$	18	10	10	18	10	8	80
$F2$	10	18	2	10	10	8	80
$F2$	2	10	18	10	10	8	-32

The domain algebra to produce *det* is

let *det* **be** $x_1 \times y_2 + x_2 \times y_p + x_p \times y_1 - x_1 \times y_p - x_2 \times y_1 - x_p \times y_2$

and then we find the face containing the point

face **where** (**equiv and of** *det*$\geq$0 **by** *face*) **in** *pip*

(The general point-in-polygon is more complicated. A non-convex polygon containing the vertex can nonetheless have some negative values of *det*, so we must compute the angle as well, and take the total of the signed angles. If this total is 2π, the vertex is in the polygon. Other algorithms avoid the expense of computing inverse trigonometric functions, but for secondary storage applications, this expense is relatively unimportant.)

6.3.2 Delaunay Edges and Swapping

Now that we know that vertex $V5$ is in the lower triangle, $F1$, we can construct the three edges and connect them from $V5$ to the three vertices, $V1, V2$, and $V3$. We generate edges f, g, and h, originating at V5 and destined to $V1, V2$, and $V3$, respectively. At $V1$, $(f, 2)$ is counterclockwise after $(a, 0)$, so the splice needed is $((a, 0), (f, 2))$. At $V2$ we splice $((b, 0), (g, 2))$, and at $V3$ we splice $((c, 0), (h, 2))$.

For the new edges to meet at $V5$, we splice them in counterclockwise order, $((f, 0), (g, 0))$ and $((g, 0), (h, 0))$.

The first three of these five splices are independent, and so each generates two substitutions. For example, $((a, 0), (f, 2))$ gives $((a, 0) \to (f, 2))$ and $((f, 2) \to (a, 0))$. The last two splices are cyclic, and produce a cycle of three substitutions, $((f, 0) \to (g, 0))$, $((g, 0) \to (h, 0))$, and $((h, 0) \to (f, 0))$.

We do not apply these substitutions yet. More work must be done first. We must check each quadrilateral that could be produced by removing one of the original edges, a, b, or c to see if this edge should be swapped from its present pair of vertices to the other two vertices of the quadrilateral. Only a is a candidate in this example. Here is the InCircle test to see whether a should continue to connect $V1$ and $V2$, or should be swapped to connect $V4$ and $V5$.

The predicate, *InCircle*$(V1, V4, V2, V5)$, for vertices $V1, V4, V2$, and $V5$, with the first three forming a counterclockwise triangle, is *true* iff $V5$ is inside the circle on $V1, V4$, and $V2$. Guibas and Stolfi [27] use the positiveness of the determinant

$$z_1 \begin{vmatrix} x_4 & y_4 & 1 \\ x_2 & y_2 & 1 \\ x_5 & y_5 & 1 \end{vmatrix} - z_4 \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_5 & y_5 & 1 \end{vmatrix} + z_2 \begin{vmatrix} x_1 & y_1 & 1 \\ x_4 & y_4 & 1 \\ x_5 & y_5 & 1 \end{vmatrix} - z_5 \begin{vmatrix} x_1 & y_1 & 1 \\ x_4 & y_4 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}$$

as this test, where $z_i = x_i^2 + y_i^2$.

If *InCircle*$(V1, V4, V2, V5)$ is true, then the opposite vertices $V4$ and $V5$ are closer to each other than the opposite vertices $V1$ and $V2$, and so $V4$ and $V5$ form the Delaunay edge. If *InCircle*$(V1, V4, V2, V5)$ is false, $V1$ and $V2$ form the Delaunay edge. In this example, the determinant is 480, so *InCircle*$(V1, V4, V2, V5)$ is true.

Accordingly, we must swap a by disconnecting it from $V1$ and $V2$ (splices $((d, 0), (a, 0))$ and $((g, 2), (a, 2))$), and then connecting it to $V4$ and $V5$ (splices $((e, 0), (a, 2))$ and $((f, 0), (a, 0))$).

Combining these four splices with the previous five gives twelve substitutions: a cycle of six, a cycle of four, and a cycle of two.

$$\begin{array}{l|l|l}
((a,0) \to (f,2)) & ((g,2) \to (b,0)) & ((c,0) \to (h,2)) \\
((f,2) \to (d,0)) & ((b,0) \to (e,0)) & ((h,2) \to (c,0)) \\
((d,0) \to (f,0)) & ((e,0) \to (a,2)) & \\
((f,0) \to (h,0)) & ((a,2) \to (g,2)) & \\
((h,0) \to (g,0)) & & \\
((g,0) \to (a,0)) & &
\end{array}$$

Applying these substitutions in the way we did for polygon overlay in section 6.1.3, we get the topology of *After* in figure 15.

6.3.3 Voronoi Diagram

Since this topology includes the dual, both the Delaunay triangles (solid lines in figure 15) to the Voronoi polygons (dashed lines) are already present. All we must do is change the direction values, so that the vertices of the Voronoi diagram (faces of the Delaunay diagram) have even numbered directions and the faces have odd numbers. We do this by subtracting 1 from the odd-numbered directions, and adding 1 to the even.

$$\textbf{let } \mathit{dir1}' \textbf{ be if } \mathit{dir1} \textbf{ mod } 2 = 0 \textbf{ then } \mathit{dir1} + 1 \textbf{ else } (\mathit{dir1} - 1)\textbf{mod } 4;$$

(We do this for both *dir1* and *dir2*, then we rename both back again.) Figure 16 shows the Delaunay (before) and the Voronoi (after) topologies. Figure 16 also shows the relationships between edges, vertices and faces, *VertFace*, for both diagrams. We can see that the exchange of vertices and edges is trivial.

The remaining task is to calculate the coordinates of the Voronoi vertices (formerly the faces of the Delaunay diagram; we shall continue to label them Fi, for consistency with figure 15). Each vertex, except the vertex at infinity, is on the right bisector of all three edges of the triangle it belongs to. (That is, it is the circumcentre of the three vertices of the triangle.) We need to compute the slope, σ, and y-intercept, κ, for the right bisector of each edge connecting to the internal vertex. Then we must solve these equations in pairs to get the coordinates of their intersections.

$$\textbf{let } \sigma \textbf{ be } (x - x')/(y' - y);$$
$$\textbf{let } \kappa \textbf{ be } (y + y')/2 - (x' - x) \times \sigma/2;$$

e	x	x'	y	y'	σ	κ
a	10	10	8	18	0	13
g	10	2	8	10	4	-15
h	10	10	8	2	0	5
f	10	18	8	10	-4	65

$$\textbf{let } x \textbf{ be } (\kappa' - \kappa)/(\sigma - \sigma');$$
$$\textbf{let } y \textbf{ be } x \times \sigma + \kappa;$$

e	e'	σ	σ'	κ	κ'	x	y
a	g	0	4	13	-15	7	13
g	h	4	0	-15	5	5	5
h	f	0	-4	5	65	15	5
f	a	-4	0	65	13	13	13

In general, solving these equations at each internal vertex will give us three pairs of equations for each circumcentre. Ideally, all three will give the same result, and relational duplicate elimination will return just one tuple. In practice, roundoff errors may cause the results to differ, in which case we can use the average (and we can write code to find to how many significant figures they agree and so tell us the precision of the answers).

With these results, we have the third, geometrical, component of the data structure, which is common to both Delaunay and Voronoi.

$Delaunay\Delta($*edge1*	*dir1*	*edge2*	*dir2*$)$
f	0	a	0
a	0	g	0
g	0	h	0
h	0	f	0
f	2	c	2
c	2	d	0
d	0	f	2
e	0	a	2
a	2	d	2
d	2	e	0
g	2	e	2
e	2	b	0
b	0	g	2
b	2	c	0
c	0	h	2
h	2	b	2
e	1	d	1
d	1	c	1
c	1	b	1
b	1	a	1
a	3	e	3
e	3	g	1
g	1	a	3
b	3	h	1
h	1	g	3
g	3	b	3
c	3	f	1
f	1	h	3
h	3	c	3
a	1	f	3
f	3	d	3
d	3	a	1

$Voronoi\Delta($*edge1*	*dir1*	*edge2*	*dir2*$)$
f	1	a	3
a	1	g	3
g	1	h	3
h	1	f	3
f	3	c	1
c	3	d	3
d	1	f	1
e	1	a	1
a	3	d	1
d	3	e	3
g	3	e	1
e	3	b	3
b	1	g	1
b	3	c	3
c	1	h	1
h	3	b	1
e	0	d	0
d	0	c	0
c	0	b	0
b	0	a	0
a	2	e	2
e	2	g	0
g	0	a	2
b	2	h	0
h	0	g	2
g	2	b	2
c	2	f	0
f	0	h	2
h	2	c	2
a	0	f	2
f	2	d	2
d	2	a	0

$DVertFace($*edge*	*org*	*dest*	*left*	*right*$)$
a	$V5$	$V4$	$F6$	$F5$
b	$V2$	$V3$	$F3$	$U1$
c	$V3$	$V1$	$F4$	$U1$
d	$V1$	$V4$	$F5$	$U1$
e	$V4$	$V2$	$F6$	$U1$
f	$V5$	$V1$	$F5$	$F4$
g	$V5$	$V2$	$F3$	$F6$
h	$V5$	$V3$	$F4$	$F3$

$VVertFace($*edge*	*left*	*right*	*dest*	*org*$)$
a	$V5$	$V4$	$F6$	$F5$
b	$V5$	$V2$	$F3$	$U1$
c	$V5$	$V3$	$F4$	$U1$
d	$V5$	$V1$	$F5$	$U1$
e	$V5$	$V4$	$F6$	$U1$
f	$V5$	$V4$	$F5$	$F4$
g	$V5$	$V4$	$F3$	$F6$
h	$V5$	$V4$	$F4$	$F3$

Figure 16: The Dual Topology of Delaunay and Voronoi

$Geom(vf$	x	$y)$
$V1$	18	10
$V2$	2	10
$V3$	10	2
$V4$	10	18
$V5$	10	8
$F3$	5	5
$F4$	15	5
$F5$	13	13
$F6$	7	13
$U1$	∞	∞

6.4 Polygon Skeleton by Divide-and-Conquer

The Delaunay triangulation algorithm of the previous section can run at best in time linear in the number of vertices already present when the new vertex is inserted. (This linear cost is incurred by finding which Delaunay triangle the new vertex is in; subsequently finding the new Delaunay edge is independent of the size of the diagram.) Thus, to build an entire Delaunay triangulation of N points will cost $\mathcal{O}(N^2)$ operations using this method.

Such a circumstance is a potential candidate for a generic technique in computer science called "divide-and-conquer", which is good for this tutorial to introduce. The idea is that if a problem costs worse than a linear number of operations to solve, and if the solution to two (or more) small parts of the problem can be combined into the solution of the merged problem in linear time, then an improvement on the super-linear solution is to divide it into two, solve each of the two, and merge the solutions. And how do we solve each of the two smaller problems? Why, by dividing them each into two and merging, and so on, recursively.

A simple illustration of this procedure is sorting, which has exactly the same divide-and-conquer analysis that Voronoi diagrams will turn out to have. A super-linear sort algorithm, which is quadratic in cost, is the sort in which we make a pass of the data to find the smallest element (at linear cost), then make a second pass of all but this element to find the next-smallest, and so on, for one pass for each element. The cost of this is plainly $\mathcal{O}(N^2)$, and indeed $N^2/2$, just like the Delaunay triangulation. However, two sorted sets of data can be merged into a single, combined sorted set in linear time, so divide-and-conquer decomposes sorting into a tree of merges, with the root of the tree being the result of merging its two descendents, each of which is a merge of its two descendents, and so on until a level is reached in which the sets of data are so small that sorting each one is trivial (for instance, each is just one element). So the cost is in the $\log_2(N)$ levels of merge of the N elements, with each level of merge costing N. The total cost is $\mathcal{O}(N \log N)$ instead of $\mathcal{O}(N^2)$.

In this section, we will do the same thing for Voronoi diagrams. The details are rather more complicated than sorting, and so provide a test of the programming operations for secondary storage that we have been illustrating. We could apply divide-and-conquer directly to Delaunay triangles by splitting the problem on either side of some x-value and zipping the two solutions together by building a sequence of new Delaunay edges for increasing y-values [27]. Instead, we will tackle a somewhat more difficult, related problem, which has useful applications in mapmaking, particularly in label placement.

This is the problem of finding the *skeleton*, or *medial axis* (or "anti-crust") of a polygon—or, for that matter, of a whole set of polygons. It can also be called a Voronoi diagram problem, because the skeleton is the set of points in the interior of the polygon which are equidistant from the nearest parts of the polygon boundary. Sometimes the right bisector of the line connecting two points is part of the skeleton, just as in the point-point Voronoi diagrams we have been constructing. In addition, however, the skeleton can include lines which are equidistant from two edges in the polygon boundary—the bisector of the angle between those two edges. Or it can include lines which are equal distances from a vertex and from an edge of the polygon boundary, and such lines are no longer straight.

Because of the added involvement of edges in this variant of the equi-distance problem, the dual to the Voronoi diagram is not a simple Delaunay triangulation, so we will build the Voronoi diagram directly.[2]

Polygons come in various classes, and we should clarify which classes of polygon arise in mapmaking. General polygons include simple polygons, which in turn include convex polygons. The edges of convex polygons all meet at vertices with interior angles less than two right angles. Simple polygons allow "reflex" vertices, at which the interior angle exceeds two right angles: such vertices give rise to the point-point and point-edge components of the skeleton mentioned above. Maps can have both convex and simple polygons. They can also contain the kind of non-simple polygon that has "holes" in it, for example, a lake with islands, or a contour containing higher (or lower) contours. (The second kind of non-simple polygon, in which the edges of the same polygon may cross, is less usual in maps.)

To get started, we can inspect the skeleton of a convex polygon, which is composed entirely of parts of the bisectors of angles between various edges of the polygon. From figure 17 we can see that a convex polygon of N edges (or vertices) can have $N-2$ skeleton vertices. Each of these six vertices is numbered in the figure, and the corresponding number is also associated with the polygon edge eliminated by the vertex. The first vertex is the centre of the smallest inscribed circle touching at least three edges, the second is of the next-smallest, and so on, until the last vertex accounts for the three remaining edges (which need not be adjacent), hence $N-2$.

(To stress that the skeleton of a polygon is a generalized Voronoi diagram, and that the algorithm we will discuss is based on the algorithm for point-point Voronoi diagrams, the figures use "Voronoi" to label the skeleton edges.)

In figure 18, one of the vertices has been made reflex. We see that the reflex vertex causes an additional skeleton vertex, and so a number has been assigned to the reflex vertex, and the corresponding number to the skeleton vertex that eliminates it. Thus, a polygon with N edges and r reflex vertices has up to $N+r-2$ skeleton vertices. Euler's formula, ($v+f=e+2$) relating numbers of vertices, v, faces, f, and edges, e, tells us therefore that the skeleton can have up to $2(N+r)-3$ edges. (Smaller numbers occur if two or more skeleton vertices coincide, which they do in the case that the circle centred there touches four or more polygon edges.)

The general algorithm to find skeletons will thus consider both edges and reflex vertices to be elements of the polygon. Since the number of skeleton vertices is linear in N and r, we could hope for an algorithm of linear complexity. Such an algorithm exists for simple polygons ([12]). For polygons with holes, however, we can presently do no better than the $\mathcal{O}(N \log N)$ divide-and-conquer algorithm that is the topic of this section.

We do not need closed polygons to make such constructions, and the merge part of the algorithm combines pairs of open "polygons". We may limit our attention to the skeleton of one side only of such a polyline, namely the side that will be the inside of the final polygon. A terminal vertex of such a polyline is considered reflex. Its contribution to the skeleton is the line orthogonal to the edge it terminates, just as the separators of vertex 4 in figure 18 are orthogonal to the two edges it connects.

[2] An *approximation* to the medial axis of a polygon can be found by triangulating the polygon (a Delaunay triangulation of its vertices: $\mathcal{O}(N \log N)$) then using the centres of circles inscribed in each triangle as skeleton vertices.

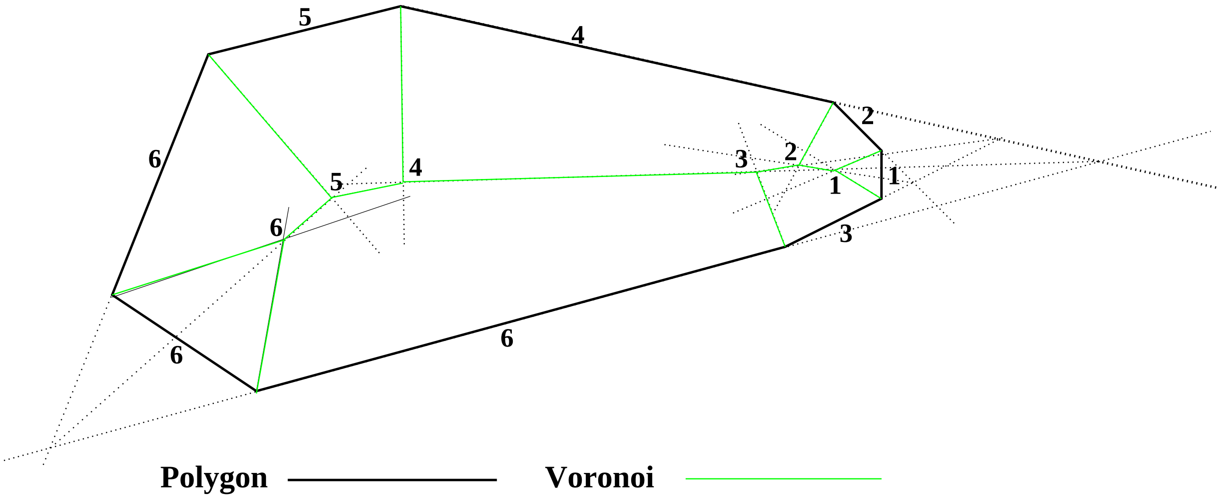


Figure 17: Skeleton of a Convex Polygon

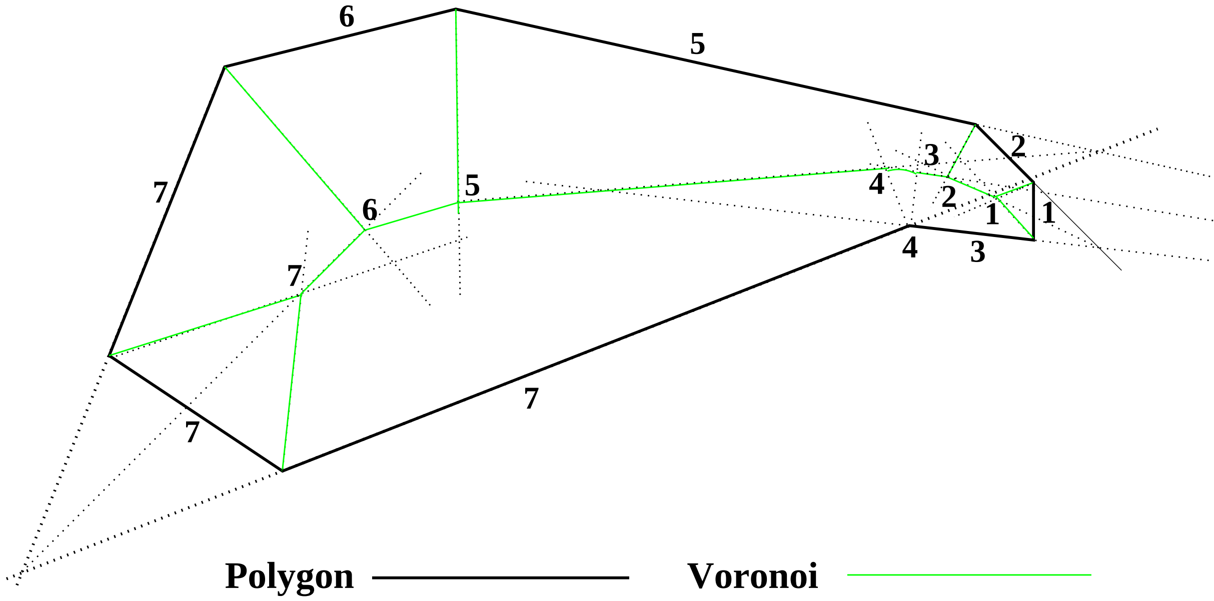


Figure 18: Skeleton of a Reflex Polygon

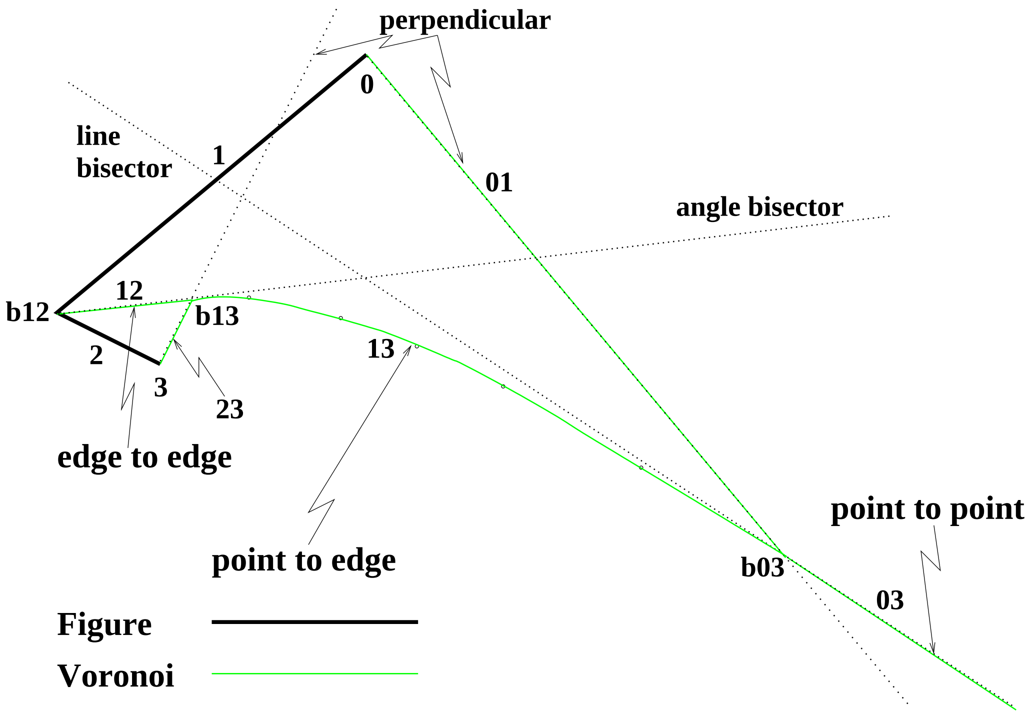


Figure 19: Skeleton of a Left Side

Figures 19 and 20 show the left and right sides, respectively, of the merge that we will now demonstrate. (They are related in structure so that we can easily imagine what the skeleton of the other side of each looks like.) These two figures will be merged to form a further portion of a polygon, which is still not a complete polygon.

In the next two sections we will, respectively, introduce the operations needed in skeleton construction by building the skeleton of the left portion (figure 19), and then go on to illustrate the merge process.

6.4.1 Preamble to the Merge: the Basic Calculations

The data structure for the two-edge part of the polygon shown in figure 19 is

QuadEdge

(*edge1*	*dir1*	*edge1*	*dir1*)
1	0	1	0
1	2	2	0
2	0	1	2
2	2	2	2
2	1	1	1
1	1	1	3
1	3	2	3
2	3	2	1

VertFace

(*edge*	*org*	*dest*	*left*	*right*)
1	0	b12	U	U
2	b12	3	U	U

Geom

(*vf*	*x*	*y*)
0	4800	6600
b12	1200	3600
3	2400	3000

Here, the edges (1, 2) and the reflex vertices (0,3), are listed as labelled in the figure. The unlabelled vertex connecting edges 1 and 2 is listed as `b12`.

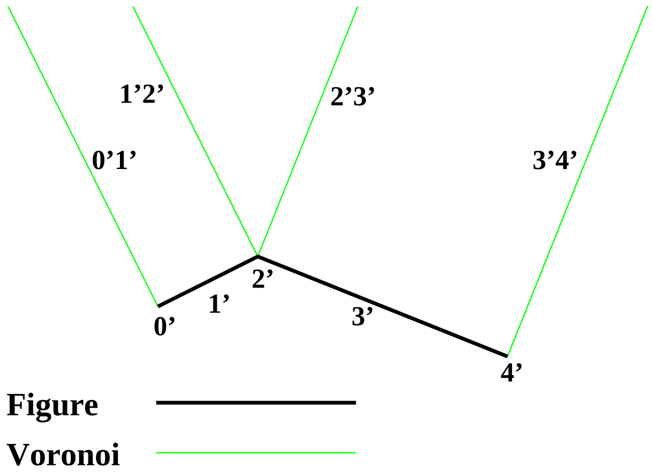


Figure 20: Skeleton of a Right Side

Without showing the code, we extract from this another representation of the elements needed for the skeleton calculation, the reflex vertices and the edges.

Reflex			*Edges*							
$(el$	r_x	$r_y)$	$(el$	b_x	b_y	e_x	e_y	d_x	$d_y)$
0	4800	6600	1	4800	6600	1200	3600	-6	-5
3	2400	2000	2	1200	3000	2400	3000	2	-1

Here, we use special field names for the coordinates of the reflex vertices, (r_x, r_y), and of the beginnings, (b_x, b_y), ends, (e_x, e_y), and directions, (d_x, d_y), of edges. The latter are needed because the Voronoi diagram has semi-infinite edges, with only one end point. (For finite edges, the direction can be derived from the beginning and end vertices:

$$d_x = e_x - b_x; d_y = e_y - b_y.)$$

Note that (d_x, d_y) could be replaced by a single *slope*, d_y/d_x, but that this gives problems for vertical lines. Because we use two numbers for direction instead of the slope, we have some leeway in signs and in relative sizes. We choose the signs of the direction so that the point $(b_x + d_x, b_y + d_y)$ is on the line (i.e., "after" the beginning vertex). Also, for the example, we choose the values of d_x and d_y to have a greatest common divisor of 1, but this is not an essential step.

(We will also need a further relation in the quad-edge data structure for semi-infinite edges.)

We will need five kinds of calculation for the merge step in the next section, and we can practice them here by partially constructing the Voronoi diagram on the right of this left-hand side. (That is, to the right as we look at figure 19. The edges in the quadedge representation and in *Edges* are directed downwards, so that the Voronoi diagram is on their left.)

The **first calculation** is to find the semi-infinite edges starting at the reflex vertices and orthogonal to the edges they terminate. The beginning vertices of these edges are the reflex vertices themselves, and their directions are found from the directions of the orthogonal edges by negating one component and then swapping. In fact, in both cases, we rotate the direction components using

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$

because, for `0`, we rotate `1` clockwise from its beginning vertex, while, for `3`, we rotate `2` counterclockwise from its end vertex. So we get the following two semi-infinite edges.

Edges

(el	b_x	b_y	e_x	e_y	d_x	d_y)
:	:	:	:	:	:	:
01	4800	6600			5	−6
23	2400	3000			1	2

The **second calculation** is to bisect the angle between `1` and `2`, i.e., at vertex `b12`. Since the bisector, which will also be a semi-infinite edge, starts at the common vertex, `(2, 6)`, we reverse the representation of edge `2`, which for this calculation means negating both d_x and d_y. The following is true about the sine, s, and the cosine, c, of the angle of the line bisecting the angle between two edges with angles of sines s_1 and s_2, respectively, and cosines c_1 and c_2, respectively.

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{pmatrix} \begin{pmatrix} c_2 & -s_2 \\ s_2 & c_2 \end{pmatrix}$$

This tells us that

$$\begin{aligned} 1 - 2s^2 = 2c^2 - 1 = c^2 - s^2 &= c_1c_2 - s_1s_2 \\ 2cs &= s_1c_2 + c_1s_2 \end{aligned}$$

and we can solve the first for both c^2 and s^2, and use the second to find the relative sign of s and c. The absolute signs of the direction components for the bisector are given by requiring that the bisector lie to the left of the two edges, in terms of their original directions. That is, going from a point on `1` to a point on `2` to a point on the bisector should be a counterclockwise motion. These points can be given by the displacements from the common vertex, $\mathbf{v}$, by the respective directions, normalized so they lie on a unit circle centred at $\mathbf{v}$ and so are not collinear. Since c_i and s_i are just the normalized values of d_x and d_y, respectively, for each line, i = `1` and i = `2`, the calculation requires only the ability to extract square roots and to find a determinant, once the different values of d_x and d_y have been combined for edge `1` (negated) and edge `2`. As a result, we add the following semi-infinite edge.

Edges

(el	b_x	b_y	e_x	e_y	d_x	d_y)
:	:	:	:	:	:	:
12	1200	3600			0.9933	0.1153

The **third calculation** finds the intersection of two semi-infinite edges, for example the bisector, `12`, we just found, and the orthogonal, `23`, at `3`. Instead of the method used in section 6.1.1 for finite edges, we first find the intersection point as if the edges were infinite straight lines, then we check if this intersection point lands on the portion of interest. For this, we express the equation of each line in terms of its beginning point and direction.

$$0 = -d_y(x - b_x) + d_x(y - b_y) = -d_yx + d_xy + c$$

where $c = d_yb_x - d_xb_y$ (which is a 2×2 determinant). The solution of two such equations uses the three further 2×2 determinants contained in

$$\begin{matrix} -\ {}_1d_y & {}_1d_x & {}_1c \\ -\ {}_2d_y & {}_2d_x & {}_2c \end{matrix}$$

where prefix subscripts distinquish the two lines from each other. The point common to the two lines is

$$c_x = \frac{{}_1d_x\ {}_2c - {}_2\ d_x\ {}_1c}{-\ {}_1d_y\ {}_2d_x + {}_2\ d_y\ {}_1d_x}, c_y = \frac{-\ {}_1c\ {}_2d_y + {}_2\ c\ {}_1d_y}{-\ {}_1d_y\ {}_2d_x + {}_2\ d_y\ {}_1d_x}$$

It is then easy to check whether (c_x, c_y) is on the interesting portion of either semi-infinite edge: for each line, $(c_x - b_x, c_y - b_y)$ will have the same signs as (d_x, d_y). For the example of lines `12` and `23`, the result is $(c_x, c_y) = (2792, 3780)$, and we can update the entries in *Edges* with this as a new end vertex for `12` and `23`.

Edges

$(el$	b_x	b_y	e_x	e_y	d_x	$d_y)$
:	:	:	:	:	:	:
12	1200	3600	2792	3780	0.9933	0.1153
23	2400	3000	2792	3780	1	2

(This calculation of intersection points will also work for finite edges. We must only conclude with one further test, that each component of $(c_x - b_x, c_y - b_y)$ is less than the corresponding component of $(e_x - b_x, e_y - b_y)$.)

The Voronoi diagram of figure 19 now has one triangle, bounded by edges `2, 12` and `23`. This contains all points that are closer to `2` than to any other element. We call the new vertex `b13`.

The **fourth calculation**, which is needed to find the the Voronoi edge `13`, separating edge `1` from vertex `3`, involves the curved line that is equidistant from a straight edge and a vertex. We normalize the problem by rotating the coordinate axes so that the straight edge is parallel to the new X-axis, and translating them so that the vertex is on the new Y-axis, with the new origin halfway between the two. We will call the straight edge the *directrix* and the vertex the *focus*. The Y-separation between the two we call the *semi-latus rectum*, or $2/A$, with the parameter, A, that we will soon see the use for. In the new coordinate system, a point, (X, Y), on the curve we are trying to find, will be distance $Y + 1/A$ from the directrix and distance $\sqrt{X^2 + (Y - 1/A)^2}$ from the focus, and these two distances are equal, by our requirement. (In particular, the origin will be on the curve.) Solving $(Y + 1/A)^2 = X^2 + (Y - 1/A)^2$ gives $Y = AX^2$, the equation of a parabola. (Of course, one of the definitions of a parabola is the locus of points equidistant from a straight line and a given point, just our requirement.)

So the fourth calculation will be first to determine the parameter, A, for the parabola defined by elements `1` and `3`, then to find where it intersects the semi-infinite edge `01`. To find the intersection of a parabola with a straight line, we also transform the line into the favoured coordinate system for the parabola, and solve the quadratic equation. (In this case, line `01` is vertical in the rotated coordinate system, and so has the simple equation $X =$ constant, which is particularly easy to solve.) Transforming the line means rotating and translating the coordinates of its beginning vertex, and rotating its direction. (Since a direction is a difference between vertices, it is invariant under translation.) We will also rotate and translate the beginning vertex, `b13`, of the parabola, because we will eventually need to confirm, as we did with intersecting straight lines, whether the intersection point is on the part of the parabola that interests us.

Rotating the coordinate axes is easy, given the direction, (d_x, d_y), of the directrix: the rotation matrix consists of only these two quantities. The only subtlety is to rotate in such a way that the focus is above the directrix in the new Y direction. If the vertices $(b_x, b_y), (e_x, e_y)$ and (f_x, f_y) (the latter pair gives the focus) are in counterclockwise sequence, we rotate one way, otherwise the other way. Here, these are the beginning and end vertices of the directrix, `1`, and the coordinates of the focus, `3`. These are the vertices whose coordinates must be rotated.

$$\begin{pmatrix} {}_1b_X & {}_1e_X & {}_3f_X \\ {}_1b_Y & {}_1e_Y & {}_3f_Y \end{pmatrix} = \begin{pmatrix} d_x & d_y \\ -d_y & d_x \end{pmatrix} \begin{pmatrix} {}_1b_x & {}_1e_x & {}_3f_x \\ {}_1b_y & {}_1e_y & {}_3f_y \end{pmatrix} =$$

$$\begin{pmatrix} -6 & -5 \\ 5 & -6 \end{pmatrix} \begin{pmatrix} 4800 & 1200 & 2400 \\ 6600 & 3600 & 3000 \end{pmatrix} = \begin{pmatrix} -61800 & -25200 & -29400 \\ -15600 & -15600 & -6000 \end{pmatrix}$$

From this we have the semi-latus rectum, $2/A = \mid -15600 + 6000 \mid = 9600$. (Notice that by using d_x and d_y directly, instead of the cosine and sine, respectively, we have also scaled the coordinate system. For the sake of keeping integer values, we allow this scaling, but must remember to scale back again, by a factor $d_x^2 + d_y^2 = 61$, when we reverse the transformation.) We also have the translation needed, which is by a distance $(v_X, v_Y) = (f_X, (b_Y + f_Y)/2) = (-29400, \ -10800)$.

Now we rotate (and scale) and translate the beginning vertices of line `01` and of the parabola `13`. (Note that in this case, we already know the result of the former, because it is also the beginning vertex of line `1`, the directrix.)

$$\begin{pmatrix} {}_{01}b_X & {}_{13}b_X \\ {}_{01}b_Y & {}_{13}b_Y \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} {}_1d_x & {}_1d_y & -v_X \\ -\ {}_1d_y & {}_1d_x & -v_Y \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} {}_{01}b_x & {}_{13}b_x \\ {}_{01}b_y & {}_{13}b_y \\ 1 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} -6 & -5 & 29400 \\ 5 & -6 & 10800 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 4800 & 2792 \\ 6600 & 3780 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} -32400 & -6254 \\ -4800 & 2082 \\ 1 & 1 \end{pmatrix}$$

And we rotate (and scale) the direction of line `01` (which in this case is orthogonal to the directrix, `1`).

$$\begin{pmatrix} {}_{01}d_X \\ {}_{01}d_Y \end{pmatrix} = \begin{pmatrix} {}_1d_x & {}_1d_y \\ -\ {}_1d_y & {}_1d_x \end{pmatrix} \begin{pmatrix} {}_{01}d_x \\ {}_{01}d_y \end{pmatrix} = \begin{pmatrix} -6 & -5 \\ 5 & -6 \end{pmatrix} \begin{pmatrix} 5 \\ -6 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

where we replaced 61 as the result for ${}_{01}d_Y$ by 1 as a (valid) simplification. Thus, the equation for the line `01` is

$$0 = -\ {}_{01}d_Y(X - \ {}_{01}b_X) + \ {}_{01}d_X(Y - \ {}_{01}b_Y) = -X - 32400$$

Combining this with the equation for the parabola, $Y = AX^2$, we have $Y = 32400^2/19200 = 54675$ so the intersection point is $(-32400{,}54675)$ in the transformed coordinates.

The part of the parabola that interests us must have a smaller X value than that of its beginning vertex, (-6254,2082). We know this because we are building the skeleton to the left of the original edges, including the directrix, which we traversed in the direction of increasing X. Now we are coming back again, so X is decreasing. The above intersection point is indeed on the relevant part of the parabola.

The final step of the fourth calculation is to transform this back (remembering to scale by 1/61).

$$\begin{pmatrix} c_x \\ c_y \end{pmatrix} = 1/({}_1d_x^2 + \ {}_1d_y^2) \begin{pmatrix} {}_1d_x & -\ {}_1d_y \\ {}_1d_y & {}_1d_x \end{pmatrix} \begin{pmatrix} c_X + v_X \\ c_Y + v_Y \end{pmatrix} =$$

$$1/61 \begin{pmatrix} -6 & 5 \\ -5 & -6 \end{pmatrix} \begin{pmatrix} -32400 - 29400 \\ 54675 - 10800 \end{pmatrix} = \begin{pmatrix} 9675 \\ 750 \end{pmatrix}$$

We now have a parabola with both beginning and end points. We need a new relation to represent either semi-infinite or finite parabolas.

ParabEdges

$(el$	b_x	b_y	e_x	e_y	A	f_x	f_y	d_x	$d_y)$
:	:	:	:	:	:	:	:	:	:
13	2792	3780	9675	750	1/19200	2400	3000	-6	-5

And we can finish by updating *Edges* 01 with this new end point.

Edges

$(el$	b_x	b_y	e_x	e_y	d_x	$d_y)$
:	:	:	:	:	:	:
01	4800	6600	9675	750	5	-6

We call the endpoint we have just found b03.

The **fifth calculation** finds the right bisector of the line between two points, in this case, the vertices 0 and 3: the last Voronoi edge of an open polygon is this semi-infinite bisector, starting at the Voronoi vertex that completes the elimination of all the elements except the terminating vertices. The direction of the line between points 0 and 3 is

$$d_x = {}_3v_x - {}_0v_x, d_y = {}_3v_y - {}_0v_y$$

and the right bisector is orthogonal to this, rotated counterclockwise.

$$\begin{pmatrix} {}_{03}d_x \\ {}_{03}d_y \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} {}_3v_x - {}_0v_x \\ {}_3v_y - {}_0v_y \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} -4 \\ -6 \end{pmatrix} = \begin{pmatrix} 6 \\ -4 \end{pmatrix}$$

The beginning vertex is, of course, b03. So we have the final new edge of the Voronoi diagram,

Edges

$(el$	b_x	b_y	e_x	e_y	d_x	$d_y)$
03	9675	750			6	-4

The purpose of this section has been to present the above five kinds of calculation. They are not directly used to find the Voronoi diagram of the edge in figure 19, which is further decomposed and the Voronoi diagram found by merging Voronoi diagrams of the components. However, we will need the Voronoi diagram of this figure in the next section, which describes the merge, and we happen to have found it completely by the examples of the above five calculations. So we summarize them before proceeding to the merge step.

Reflex

$(el$	r_x	$r_y)$
0	4800	6600
3	2400	2000

Edges

$(el$	b_x	b_y	e_x	e_y	d_x	$d_y)$
1	4800	6600	1200	3600	-6	-5
2	1200	3000	2400	3000	2	-1
12	1200	3600	2792	3780	0.9933	0.1153
23	2400	3000	2792	3780	1	2
01	4800	6600	9675	750	5	-6
03	9675	750			6	-4

Parabolas

$(el$	b_x	b_y	e_x	e_y	A	f_x	f_y	d_x	$d_y)$
13	2792	3780	9675	750	1/19200	2400	3000	-6	-5

This converts back to the quadedge representation, with new relations for semi-infinite edges and for parabolic segments, as follows.

QuadEdge

(*edge1*	*dir1*	*edge1*	*dir1*)
1	0	01	2
01	2	1	0
1	2	2	0
2	0	12	0
12	0	1	2
2	2	23	0
23	0	2	2
12	2	23	2
23	2	13	0
13	0	12	2
01	0	13	2
13	2	03	0
03	0	01	0
01	1	03	1
03	1	13	1
13	1	23	1
23	1	2	1
2	1	1	1
1	1	01	1
2	3	23	3
23	3	12	1
12	1	2	3
2	3	23	3
1	3	12	3
12	3	13	3
13	3	01	3
01	3	1	3

VertFace

(*edge*	*org*	*dest*	*left*	*right*)
1	0	b12	V1	F3
2	b12	3	V2	F3
01	b03	0	V1	F3
12	b12	b13	F3	V2
13	b13	b03	V1	F3
23	3	b13	V2	F3
03	b03	∞	F3	F3

Geom

(*vf*	*x*	*y*)
0	4800	6600
b12	1200	3600
3	2400	3000
b13	2792	3780
b03	9675	750

InfinitySlopes

(*edge*	*x*	*y*)
03	6	-4

Parabolas

(*edge*	*dtrx*	*focus*	*A*)
13	1	3	1/19200

The right part, shown in figure 20, has a very simple Voronoi structure, which we show directly in its quadedge representation.

QuadEdge

(*edge1*	*dir1*	*edge1*	*dir1*)
1'	0	0'1'	0
0'1'	0	1'	0
1'	2	3'	0
3'	0	2'3'	0
2'3'	0	1'2'	0
1'2'	0	1'	2
3'	2	3'4'	0
3'4'	0	3'	2
0'1'	3	3'4'	1
3'4'	1	3'	1
3'	1	1'	1
1'	1	0'1'	3
1'	3	1'2'	3
1'2'	3	0'1'	1
0'1'	1	1'	3
1'2'	1	2'3'	3
2'3'	1	3'	3
3'	3	3'4'	3
3'4'	3	2'3'	1

VertFace

(*edge*	*org*	*dest*	*left*	*right*)
1'	0'	2'	V1'	F4'
3'	2'	4'	V3'	F4'
0'1'	0'	∞	F4'	V1'
1'2'	2'	∞	V1'	V2'
2'3'	2'	∞	V2'	V3'
3'4'	4'	∞	V3'	F4'

Geom

(*vf*	*x*	*y*)
0'	2400	3000
2'	3600	3600
4'	6600	2400

InfinitySlopes

(*edge*	*x*	*y*)
0'1'	−1	2
1'2'	−1	2
2'3'	2	5
3'4'	2	5

None of the code has been shown for this section. It is a straightforward application of the domain algebra, mainly, with selections to find the edges needed for each step, and with Cartesian products (joins with renaming) to combine two edges when their intersection is needed.

We are now ready to describe the merge.

6.4.2 The Merge

The central part of the skeleton-finding algorithm is to merge pairs of Voronoi diagrams for open polygons. This is done when climbing out of the recursive process that decomposed the original polygon into parts so simple that finding their Voronoi diagrams is trivial. In this section, we illustrate one such merge, between the left and right skeletons discussed above, giving the result in figure 21.

The calculations are all of the same types just discussed. In the previous section, among other procedures, we showed how to find the intersection points of two edges. What we did not show was how to determine which edges intersect with the new edge of the Voronoi diagram currently under construction. A summary of the merge in figure 21 is

`21'` hits `12` $\rightarrow$ `11'` hits `1'2'` $\rightarrow$ `12'` hits `2'3'` $\rightarrow$ `13'` hits `01` $\rightarrow$ `03'` hits `3'4'` $\rightarrow$ `04'`,

where the *separator edges*, i.e., the Voronoi edges separating the left (unprimed) component from the right (primed) component, `21', 11', 12', 13'`, and `03'` , hit either a left Voronoi edge (`12, 01`) or a right Voronoi edge (`1'2', 2'3', 3'4'`). We must determine all of these hits in linear time.

Notice, for future reference, that once we have determined a hit, the next new separator edge is found directly from the names of the two intersecting edges. For example, the separator edge `21'` hitting the left Voronoi edge `12` eliminates edge `2` from further consideration, and leads to `11'` as the next separator edge. Thus, we will modify our data structure for Voronoi edges to include the two parts of the name explicitly. This permits us to implement the $\rightarrow$ of the above sequence as a form of natural composition.

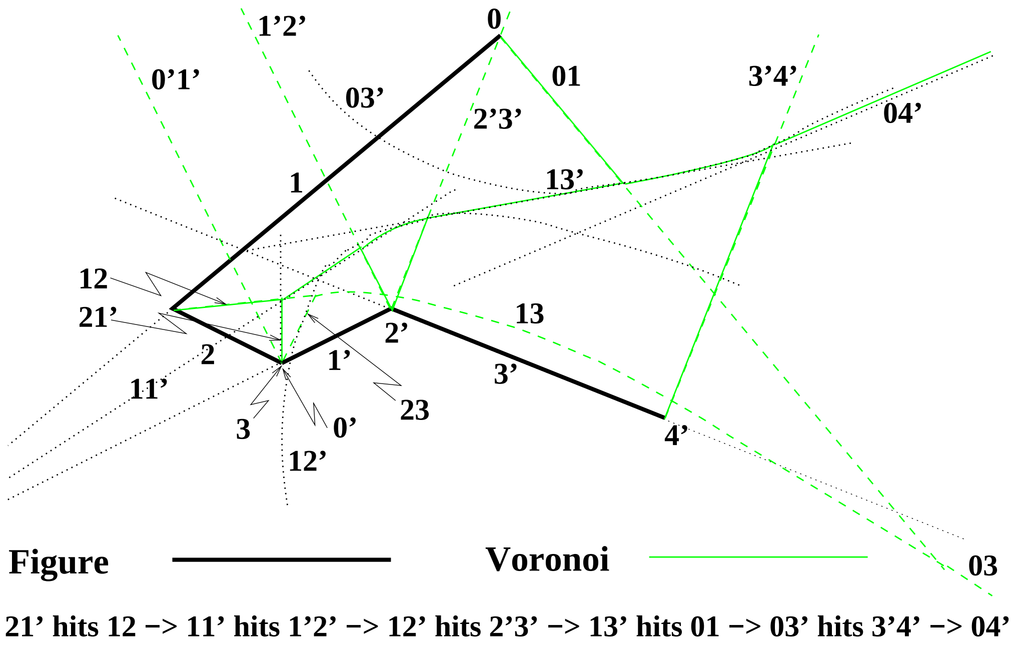


Figure 21: Merged Skeleton

The algorithm we follow is due to D. T. Lee [38], and is based on the earlier divide-and-conquer algorithm for Voronoi diagrams of point sets (see, e.g., [42]). The central consideration in these algorithms, which makes the merge linear, is that we do not need to revisit any of the edges in the Voronoi diagrams of the components while seeking the next edge intersected by the current separator edge. The edges it will hit belong to the Voronoi diagram of either the left side or the right side. If we inspect the edges of the current face of the left Voronoi diagram in counterclockwise order, and the edges of the current face of the right Voronoi diagram in clockwise order, we will not need to backtrack ([42]).

1. *Bisect angle* `21'` *and intersect with left or right Voronoi edge.*

The algorithm starts with the bisector of the angle between edge `2` of the left component and edge `1'` of the right. This is at the vertex that was called `3` in the left component and `0'` in the right. Here is the representation of edges `2, 1'` and the bisector `21'`, using two fields, *lel* and *rel*, to decompose the names of the edges into their components.

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
2		1200	3600	2400	3000	2	−1
	1'	2400	3000	3600	3600	2	1
2	1'	2400	3000			0	1

The next task is to find which existing Voronoi edge is intersected by this new edge, `21'`, and where. The counterclockwise sequence of edges in the left Voronoi diagram, and the clockwise sequence in the right Voronoi diagram, are given by the quadedge representation (using reverse sequencing for clockwise). We adapt the code of Appendix A to convert the cycles in *QuadEdge* to sequences which end at the last occurrence of the vertex originating the angle bisector (`3` or `0'` in this case). Here are the left- and right-hand sequences, after joining with *VertFace* for the other information needed.

leftCCW						*rightCW*					
(*id*	*seq*	*edge*	*dir*	*org*	*dest*)	(*id*	*seq*	*edge*	*dir*	*org*	*dest*)
V2	2	2	3	b12	3	V1'	−3	1'	3	0'	2'
V2	3	23	3	3	b13	V1'	−1	1'2'	3	2'	∞
V2	1	12	1	b12	b13	V1'	−2	0'1'	1	0'	∞

Constructing these sequences would be preceeded by a selection to find the Voronoi faces containing vertex 3 (left side) or 0' (right side), and succeeded by a selection to eliminate the tuples containing these vertices.

We can then sequence through each of the remaining tuples, to test them in ascending *seq* order for intersection with the angle bisector, 21'. (In the example so far, there is only one such remaining tuple on either side, because the Voronoi faces are both triangles. We find that each of these edges does intersect with 21', as follows (where (c_x, c_y) is the intersection point and (pos_x, pos_y) is this point relative to the beginning of 21').

Edges											
(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)	c_x	c_y	pos_x	pos_y
1	2	1200	3600	2792	3780	0.9933	0.1153	2400	3738	0	738
1'	2'	3600	3600			−1	2	2400	6000	0	3000

These (pos_x, pos_y) values tell us that both intersections fall in the relevant parts of the two edges. The intersection with the smaller of (pos_x, pos_y) is the winner, so we have a new vertex, (2400, 3738), the intersection of 21' with 12.

To anticipate, in the step following this, we will be in a new face of the left Voronoi diagram, and, in the right Voronoi diagram, we will be in the same face and will start with 1'2', the edge that lost out this time.

Since 21' hits 12 the next separator edge will be 11', and we can call the new vertex b11'. The edge data can be updated accordingly, as can *Geom* in the quadedge representation. Here are the new edges.

Edges							
(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
1	2	1200	3600	2400	3738	0.9933	0.1153
2	1'	2400	3000	2400	3738	0	1

We can also generate the following splice operations for *QuadEdge*, with the sequence numbers shown. To disconnect the figure edges from the Voronoi edges on both left and right sides:

1. *splice*((2,2),(23,0))
1. *splice*((1',0),(0'1',0))

To connect the left and right sides, and the new angle bisector:

2. *splice*((1',0),(21',0))
3. *splice*((21',0),(2,2))

This latter must be done in counterclockwise order, which is the same as the ascending order of $arctan(d_x, d_y)$ given that if *dir* is 2 or 3, the sign of (d_x, d_y) is negated.

We postpone the splices at the new vertex, b11', until we have edge 11' at the end of the next step.

2. *Bisect angle* 11' *and intersect with left or right Voronoi edge.*

We find the direction of the bisector of the angle between edges 1 and 1', and start the bisector at vertex b11'.

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
1		1200	3600	4800	6600	6	5
	1'	2400	3000	3600	3600	2	1
1	1'	2400	3738			0.8369	0.5473

The counterclockwise sequence of the new left Voronoi face gives only 01 to inspect. The clockwise sequence of the previous right Voronoi face gives 1'2' then 0'1'. The current separator edge, 11', intersects 01 and 1'2' on the edges, and 1'2' is first. The resulting new edges are

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
1	1'	2400	3738	3252	4296	0.8369	0.5473
1'	2'	3600	3600	3252	4296	−1	2

and the splices need only connect edges 11', 12, and 21' in $arctan(d_x, d_y)$ order:

4. *splice*((11',0),(12,2))
5. *splice*((12,2),(21',2))

3. *Find parabola* 12' *and intersect with left or right Voronoi edge.*

Since 11' hits 1'2', the next separator edge is 12', which is the parabola defined by directrix 1 and focus 2'. The vertex we have just created is b12'. The new parabola is

Parabolas

(*edge*	*dtrx*	*focus*	*A*)
12'	1	2'	1/24000

or, with details

ParabEdges

(*lel*	*rel*	b_x	b_y	e_x	e_y	*A*	f_x	f_y	d_x	d_y)
1	2'	3252	4296			1/24000	3600	3600	-6	-5

(Note that the rotation is the same as in the fourth calculation in section 6.4.1, because the directrix is the same. The focus is in a different position and further away.)

The counterclockwise sequence of the previous left Voronoi face gives only 01 to inspect, and the clockwise sequence of the new right Voronoi face gives 2'3'. The current separator edge, the parabola, intersects both, but 2'3' first. The resulting new edges are

ParabEdges

(*lel*	*rel*	b_x	b_y	e_x	e_y	*A*	f_x	f_y	d_x	d_y)
1	2'	3252	4296	3984	4566	1/19200	3600	3600	-6	-5

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
2'	3'	3600	3600	3984	4566	2	5

Going back to the previous new vertex, b12' we must splice to connect 12', 11', and 1'2' in that order. For the straight edges, 11' and 1'2', this is given by $arctan(d_x, d_y)$, and for the parabola, 12', we can surmise that it is not between 11' and 1'2', because it is the new edge.

6. *splice*((`12'`,`0`),(`11'`,`2`))
7. *splice*((`11'`,`2`),(`1'2'`,`2`))

4. *Bisect angle* `13'` *and intersect with left or right Voronoi edge.*

Since `12'` hits `2'3'`, the next separator edge is `13'`. We find the direction of the bisector of the angle between edges `1` and `3'`, and start the bisector at vertex `b13'`.

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
1		1200	3600	4800	6600	6	5
	3'	3600	3600	6600	2400	5	-2
1	3'	3984	4566			0.988	0.156

The candidate intersections are with `01`, again, from the left, and `3'4'` from the right: `01` is first, and the new edges are

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
0	1	4800	6600	6180	4920	-6	-5
1	3'	3984	4566	6180	4920	0.988	0.156

We must splice `13'`, `12'`, and `2'3'`, in that order, to connect them at vertex `b13'`.

8. *splice*((`13'`,`0`),(`12'`,`2`))
9. *splice*((`12'`,`2`),(`2'3'`,`2`))

5. *Find parabola* `03'` *and intersect with left or right Voronoi edge.*

Since `13'` hits `01`, the next separator edge is `03'`, which is the parabola defined by directrix `3'` and focus `0`. The vertex we have just created is `b03'`. The new parabola is

Parabolas

(*edge*	*dtrx*	*focus*	*A*)
03'	3'	0	1/34800

or, with details

ParabEdges

(*lel*	*rel*	b_x	b_y	e_x	e_y	*A*	f_x	f_y	d_x	d_y)
0	3'	6180	4920			1/34800	4800	6600	5	-2

We are now beyond the Voronoi diagram of the left component. The clockwise sequence of the previous right Voronoi face gives `3'4'`. The current separator edge, the parabola, intersects this.

ParabEdges

(*lel*	*rel*	b_x	b_y	e_x	e_y	*A*	f_x	f_y	d_x	d_y)
0	3'	6180	4920	7800	5400	1/34800	4800	6600		

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
3'	4'	6600	2400	6180	4920	2	5

The connections at vertex `b03'` are of `03'`, `01`, and `13'`.

10. *splice*((`03'`,0),(`01`,2))
11. *splice*((`01`,2),(`13'`,2))

6. *Right bisect line from* `0` to `4'`

Since `03'` hits `3'4'`, the next separator edge is `04'`. We find the direction of the right bisector of the points `0` and `4'`, and start it at vertex `b04'`.

Reflex

(*el*	r_x	r_y)
0	4800	6600
4'	6600	2400

Edges

(*lel*	*rel*	b_x	b_y	e_x	e_y	d_x	d_y)
0	4'	6180	4920			7	3

We are now outside both Voronoi diagrams, so this is the last edge.

The final connections are 12. *splice*((`04'`,0),(`3'4'`,2))
13. *splice*((`3'4'`,2),(`03'`,2))

The quadedge data structure contains all the information needed to code this sequence, and, in particular, it can give the *Edges, Reflex*, and *ParabEdges* relations, and can absorb the updates to these relations obtained during the calculations. The relational and domain algebras can be used to find many bisectors and so on at once, at the risk of finding ones which are not needed and so increasing the complexity of the calculation. The intention of the merge sequence is to keep the complexity linear, so the relational and domain operations must be repeated many times, following selections to isolate those parts of the relations to be worked on at any one time. The linear cost of the merge will depend on a good implementation of the relational operators.

6.4.3 A Whole Polygon

Figure 22 shows a complete polygon, containing the two components we have just merged and two other components which have been merged together, then the combination merged with the result of the last section. This gives an example of a polygon with a hole.

6.4.4 Grass Fire Skeletons

Another perspective on the skeleton of a polygon is that it is the set of points inside the polygon where the fire goes out if the polygon contained grass and the boundary is set on fire. This leads to algorithms whose costs depend on the area of the polygon, rather than on the number of edges. In most cases, such algorithms are likely to be more expensive than the divide-and-conquer skeleton algorithm we have just developed, but they have some advantages. First, they are readily parallelized, and this makes them especially accessible to the relational and domain algebras, and thus suitable for secondary storage. Second, although this may not be of interest to map makers, they are superior in higher dimensions, such as in finding medial planes of (three-dimensional) polyhedra.

Figure 23 shows the advancing front of the grass fire. We see that the technique must detect discontinuities, or "singularities", in the front, where the skeleton lies. A robust approach, and an overview of various methods, is available [45]. These methods are frequently used when the polygons are provided as images: their costs are functions of the number of pixels.

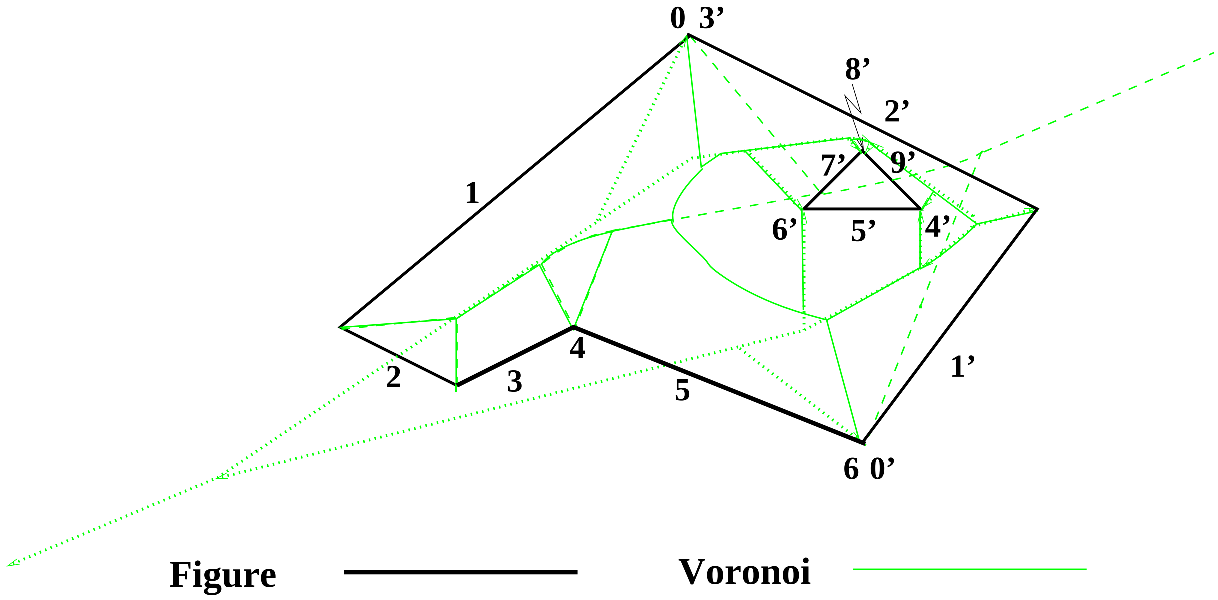


Figure 22: The Result of Merging Four Components

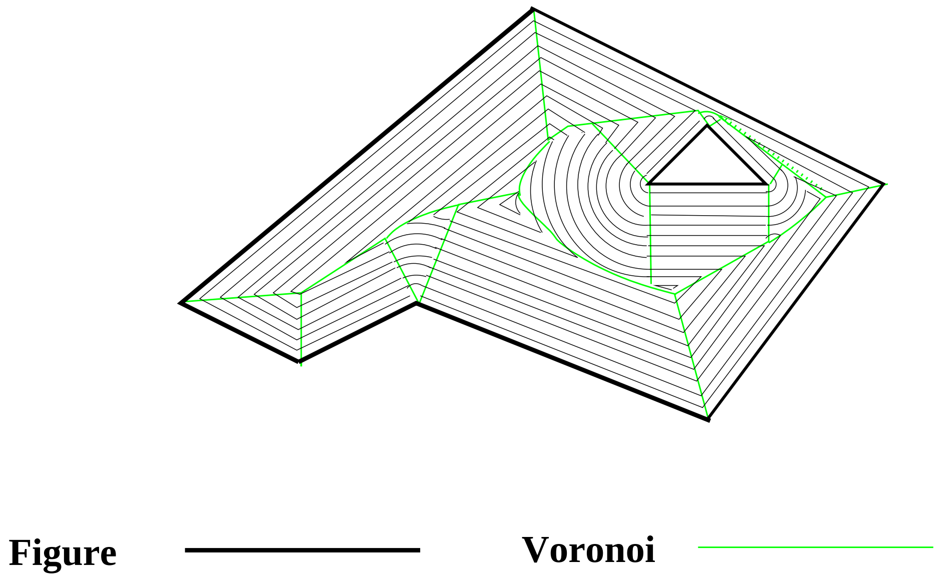


Figure 23: Skeleton by Wavefront Techniques

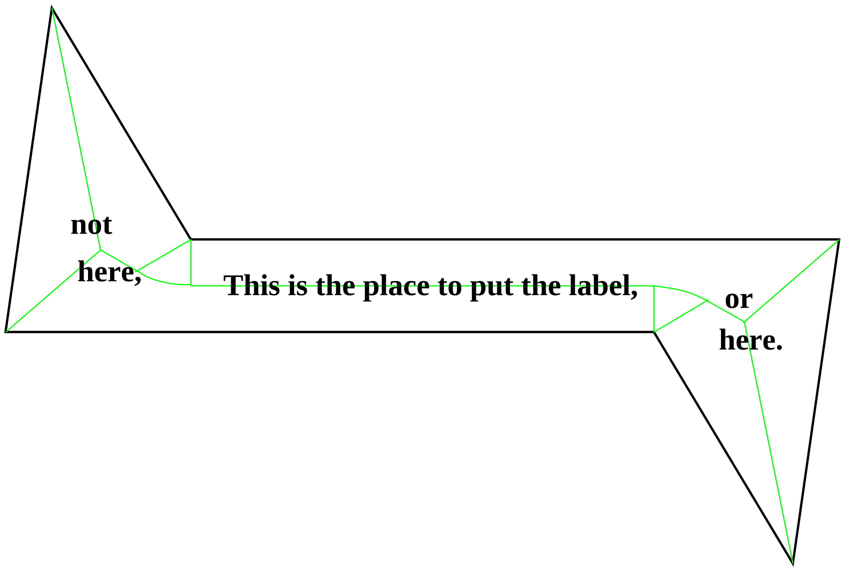


Figure 24: Labelling a Polygon on its Skeleton

6.4.5 Example: Label Placement

The idea of a skeleton of a polygon was introduced by Blum [9] as a means of simplifying polygons for pattern recognition. It can also be used to find the best place to print labels in polygons. Figure 24 illustrates this. Often, we can use the skeleton to find the “fattest” part of the polygon, by moving the centre of a variable-sized circle, which touches the edges or reflex vertices of the polygon, from vertex to vertex of the skeleton, and choosing the vertex that gives the circle the maximum radius. The label can be centred at this vertex. But we see from figure 24 that this may not be the best placement of the label, especially if we can write the label at an angle. However, the skeleton can be used in other ways to find such better alternatives.

Of course, placing a label in a single polygon hardly touches the surface of the general problem of label placement. What if the polygon is one of many contained in a larger polygon, such as provinces in a country or islands in a lake? Then labelling the contained polygons might follow the above prescription, but the containing polygon must be labelled so that its label does not collide with any of the others. We can *start* by placing the labels of the contained polygons, then treating these labels themselves as holes in the containing polygon. Inevitably, some subsequent adjustments must be made. Even the problem of labelling a collection of random *points* in two dimensions, penalizing overlaps between each label and either other labels or other points, is computationally intractable [41] and so heuristics must be used.

6.5 Implementing Spatial Predicate Approximations

In section 3.5, above, we discuss spatial predicates and their approximations. All of the predicates considered there are implicit in the quad-edge structure. Here, we would like to derive explicitly the three approximations or models considered earlier: kdAllen (section 3.5.1), kdString (section 3.5.2), and 9Inter (section 3.5.3). We work with the two intersecting triangles of figure 11 (Before) in section 6.1.

2dAllenMetric

(*face*	*face′*	*xss*	*xse*	*xes*	*xee*	*yss*	*yse*	*yes*	*yee*)
Δ1	Δ2	255	765	-765	-255	-255	765	-765	255
Δ2	Δ1	-255	765	-765	255	255	765	-765	-255

2dAllen

(*face*	*face′*	*sxss*	*sxse*	*sxes*	*sxee*	*syss*	*syse*	*syes*	*syee*)
Δ1	Δ2	+	+	-	-	-	+	-	+
Δ2	Δ1	-	+	-	+	+	+	-	-

2dAllenTern

(*face*	*face′*	*xtern*	*ytern*)
Δ1	Δ2	72	20
Δ2	Δ1	20	72

2dAllenDir

(*face*	*face′*	*dir*)
Δ1	Δ2	empty string
Δ2	Δ1	empty string

Figure 25: Data Structure for the Overlay Example

6.5.1 2dAllen

We can easily find the minimum bounding rectangles of the two triangles from the auxiliary relations, *VertFace* and *Geom*, of the quad-edge representations (figure 12).

let *face* **be** *left*;
let *xs* **be equiv min of** *x* **by** *face*;
let *ys* **be equiv min of** *y* **by** *face*;
let *xe* **be equiv max of** *x* **by** *face*;
let *ye* **be equiv max of** *y* **by** *face*;
MBR <− [*face, xs, ys, xe, ye*] **in** (*VertFace* [*org* **join** *vf*] *Geom*);

From this, the metric 2dAllen distances between any pair of features can be found by Cartesian product (we do not write the simple domain algebra to rename each field, f, to f').

let *xss* **be** $xs' - xs$;
let *xse* **be** $xe' - xs$;
let *xes* **be** $xs' - xe$;
let *xee* **be** $xe' - xe$;
let *yss* **be** $ys' - ys$;
let *yse* **be** $ye' - ys$;
let *yes* **be** $ys' - ye$;
let *yee* **be** $ye' - ye$;
2dAllenMetric <− [*face, face′, xss, xse, xes, xee, yss, yse, yes, yee*] **in**
(*MBR* **join** [*face′, xs′, ys′, xe′, ye′*] **in** *MBR*);

This result and the following three relations are shown for the example in figure 25. (Note that the code produces a symmetrical result, with tuples for both the Δ1–Δ2 relationship and the Δ2–Δ1 relationship.)

To find the sign pattern only, from *2dAllenMetric*, we need domain algebra such as

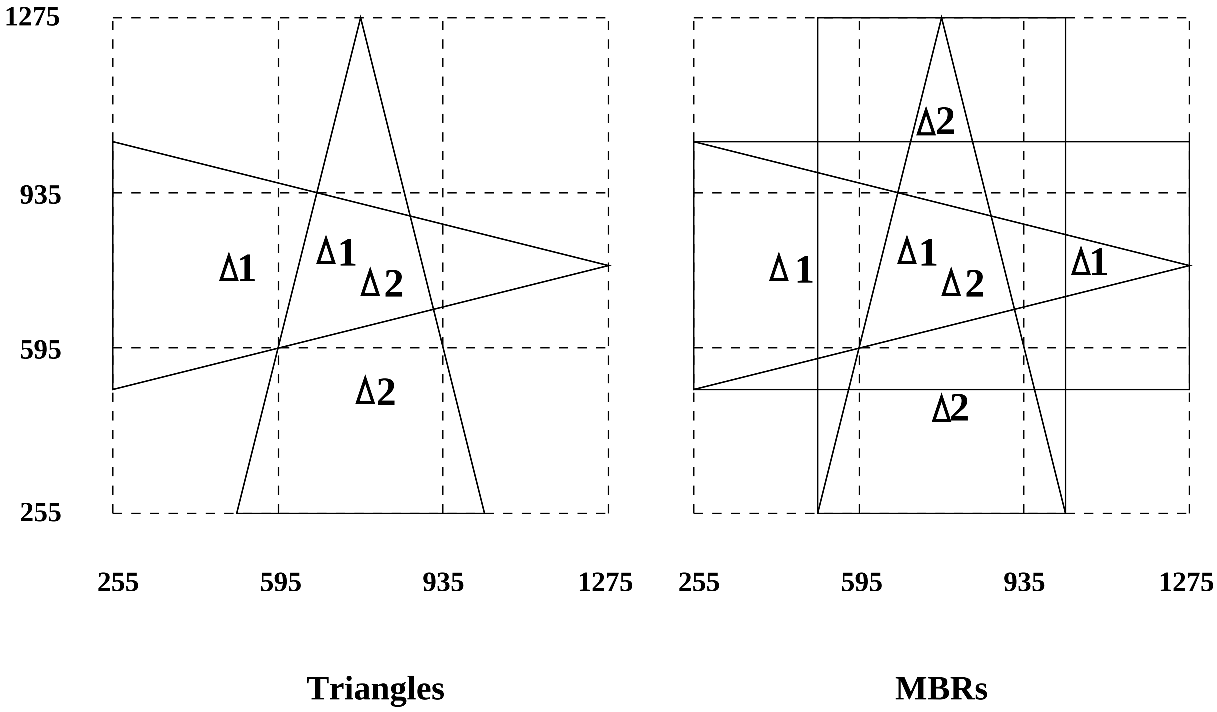


Figure 26: Grid Approximation to Overlapped Triangles

let *sxss* **be if** $xss < 0$ **then** `" − "` **else if** $xss = 0$ **then** `" 0 "` **else** `" + "`;
and so on, followed by a projection from *2dAllenMetric*.

To find the ternary value of the sign pattern, we need a similar conversion of each value to the ternary digit, `0`, `1`, or `2`

let *txss* **be if** $xss < 0$ **then** `0` **else if** $xss = 0$ **then** `1` **else** `2`;
and so on, followed by

let *xtern* **be** $txee + 3\times(txes + 3\times(txse + 3\times(txss)))$;
let *ytern* **be** $tyee + 3\times(tyes + 3\times(tyse + 3\times(tyss)))$;
and a projection from *2dAllenMetric*.

The ternary values can be used to find compass directions.

let *xdir* **be if** $xtern \geq 73$ **then** `"W"` **else if** $xtern \leq 19$ **then** `"E"` **else** `""`;
let *ydir* **be if** $ytern \geq 73$ **then** `"S"` **else if** $xtern \leq 19$ **then** `"N"` **else** `""`;
followed by a projection from *2dAllenTern*. We see from figure 25 that the two triangles of the example are considered by these definitions to coincide.

6.5.2 2dString

A 2dString approximation to the two triangles of section 6.1 starts with a grid approximation. Figure 26 (Triangles) shows one such approximation, which divides the region with the two triangles symmetrically into nine grid cells. A cell is considered to contain part of a triangle if it is more than half-filled by it. Thus, two cells contain the horizontal triangle, $\Delta1$, and two contain the vertical triangle, $\Delta2$, giving a total of three cells, because one is common.

The quad-edge representation of the 24 edges of the grid requires 96 tuples, and we do

Cell

(x	y	$face$)	$xrank$	$yrank$	x_min_rank	x_max_rank	y_min_rank	y_max_rank
255	595	Δ1	0	1	0	1	1	1
595	255	Δ2	1	0	1	1	0	1
595	595	Δ1	1	1	0	1	1	1
595	595	Δ2	1	1	1	1	0	1

2dStringDir

($face$	$face'$	x_min_rank	x_max_rank	y_min_rank	y_max_rank	x_min_rank'	x_max_rank'	y_min_rank'	y_max_rank')	$xdir$	$ydir$	dir
Δ1	Δ2	0	1	1	1	1	1	0	1	empty strings		
Δ2	Δ1	1	1	0	1	0	1	1	1	empty strings		

2dStringDist

($face$	$face'$	$xrank$	$yrank$	$xrank'$	$yrank'$)	x_min_dist	x_max_dist	y_min_dist	y_max_dist	$neighb$
Δ1	Δ2	0	1	1	0	0	1	0	1	0
Δ2	Δ1	1	0	0	1	0	1	0	1	0

Figure 27: Relations for 2dString Approximation

not show it. By a process of overlay between the grid and each triangle in turn, like that of section 6.1, we can arrive at the decomposition of the triangles into their pieces in each grid cell. Comparing the areas of these pieces with the 340×340 units in each square cell, we can obtain the relation *Cell* in figure 27, where we are about to derive the virtual fields to the right of the parentheses:

let *xrank* **be** (**fun** + **of** 1 **order** x) − 1;
let *yrank* **be** (**fun** + **of** 1 **order** y) − 1;
let *x_min_rank* **be equiv min of** *xrank* **by** *face*;
let *x_max_rank* **be equiv max of** *xrank* **by** *face*;
let *y_min_rank* **be equiv min of** *yrank* **by** *face*;
let *y_max_rank* **be equiv max of** *yrank* **by** *face*;

Renaming fields with primes and taking a Cartesian product using **join** enables us to calculate directions. The relational formulation enables us to define directions for 2dStrings as we did with 2dAllen, distinguishing east from west in the x-direction, north from south in the y-direction, and combining the two to get all eight compass directions. This breaks up the definitions of section 3.5.2 into a component for each dimension.

let *xdir* **be if** *x_max_rank*<*x_min_rank'* **then** `"W"` **else**
if *x_max_rank'* <*x_min_rank* **then** `"E"` **else** `""`;
let *ydir* **be if** *y_max_rank*<*y_min_rank'* **then** `"S"` **else**
if *y_max_rank'* <*y_min_rank* **then** `"N"` **else** `""`;
let *dir* **be** *xdir* **cat** *ydir*;

We see in figure 27 (*2dStringDir*) that these definitions, like those for 2dAllen, also consider the two triangles to coincide.

To find nearest neighbours, we take a second Cartesian product, *2dStringDist*, also shown in figure 27, define distances

let *x_min_dist* **be equiv min of** abs(*xrank* − *xrank′*) **by** *face, face′*;
let *x_max_dist* **be equiv max of** abs(*xrank* − *xrank′*) **by** *face, face′*;
let *y_min_dist* **be equiv min of** abs(*yrank* − *yrank′*) **by** *face, face′*;
let *y_max_dist* **be equiv max of** abs(*yrank* − *yrank′*) **by** *face, face′*;
let *neighb* **be if** *x_max_dist*=0 & *y_max_dist*=0 **then** 0 **else** 1;

The result in figure 27 (*2dStringDist*) is that the two triangles are deemed to be nearest neighbours.

Since 2dStrings are capable of representing relationships among more than two components of a figure, we can also find kth-nearest neighbours, but this requires careful analysis of the structure of each component, including gaps and holes.

Figure 26 also shows the grid approximation for the minimum bounding rectangles of each of the triangles. The 2dString approximation for this is

$\Delta 1 < \Delta 2 \Delta 1 : \Delta 2 < \Delta 1 \qquad \Delta 2 < \Delta 1 \Delta 1 : \Delta 2 < \Delta 2$

which differs from the 2dString for the original triangles:

$\Delta 1 < \Delta 2 \Delta 1 : \Delta 2 \qquad \Delta 2 < \Delta 1 \Delta 1 : \Delta 2$

Note that we do not need to derive the actual 2dStrings themselves to do the calculations to find directions and nearest-neighbour distances.

6.5.3 9Inter in Two Dimensions

Deriving the 2dString approximation from the full quad-edge representation required overlay and face-intersection calculations, only slightly simplified by the rectilinear properties of the grid. Each of the kdAllen and kdString relationship approximations involved the quadratic costs of Cartesian products in order to pair up spatial features in binary relationships. These efforts must be traded off against the relative ease of evaluating predicates involving the approximate relationships once they have been precomputed.

The 9-intersection topological approximation also requires Cartesian products, and apparently involves calculating nine intersections of faces and their boundaries. In fact, this work can be reduced to between four and six tests, with one more possible test, in two dimensions, for the dimensionality of the intersection of boundaries. We see this by "mining" the d-dimensional matrices in figure 3 to give the following dependences among the nine matrix elements, `ii`, `ib`, `ie`, `bi`, `bb`, `be`, `ei`, `eb`, and `ee`.

`ii`='-' ⇒ `ib`='-' & `bi`='-'
`ie`='-' ⇒ `be`='-' & `ib`='-'
`ei`='-' ⇒ `eb`='-' & `bi`='-'
`ie`='d' ⇒ `be`='$d-1$'
`ei`='d' ⇒ `eb`='$d-1$'
`ee`='d'

Finally, we also notice that only `bb` can take on more than one non-'-' value, namely 0 or 1 (in two dimensions).

This means that we can start with four tests, `ii`, `ie`, `ei`, and `bb`. The first three are face-face intersection tests and the fourth is line-line. If these are all '-', then only the exteriors of the features intersect and all other intersections are void: the features are disjoint.

If `ii`≠'-' then `ii`=d and we must test `ib` and `bi`.

If `ie`≠'-' then `ie`=d, `be`=$d-1$ and we must test `ib`.

If `ei`≠'-' then `ei`=d, `eb`=$d-1$ and we must test `bi`.

Finally, if `bb`≠'-' then we must further test `bb` to determine its dimensionality.

Both face-face and line-line intersection tests are special cases of the overlay calculations discussed in section 6.1, and the dimension of `bb` is 1 if and only if the two boundary edges are collinear (section 6.1.1).

Postscript. At this point in the tutorial, we hope to have persuaded the reader that there is no spatial data processing that needs special language constructs, even though we have not formulated every possible spatial operation in database terms. It should now be relatively safe for the reader to approach the rest of the field of spatial databases without risk of disorientation. A concise summary of the current state of confusion of the field is given by Adam and Gangopadhyay [1]. To be explicit, for the reader's benefit, the primary wrong turn taken by the field appears in chapter 4 of that book, which visits many of the extensions to database "models" that have been produced to make databases spatial. (The distinction between "field-based" and "object-based" GIS applications, is also artificial, because both end up using spatial subdivisions, usually polygons. The distinction misleadingly separates Voronoi diagrams (field-based) from the quad-edge structure (object-based), for example.)

7 Spatial On-Line Analytical Processing

The term "on-line analytical processing" (OLAP) was coined by Codd [17] to contrast with "on-line transaction processing" (OLTP), the dominant use of database systems up to that point. He writes

> ..relational DBMS were never intended to provide the very powerful functions for data synthesis, analysis and consolidation that is being defined as multi-dimensional data analysis..

and we cannot dispute his claim, for Codd also invented relational databases. We can, however, regret that his early serendipity was thus later lost, for, intended or not, relational databases are perfectly capable of all the operations of OLAP. (It is noteworthy that much subsequent research on OLAP has used the relational model. It does not, however, require extensions to relational functionality or syntax, even though SQL is inadequate [26].)

These operations are primarily aggregations, which we have seen are the realm of the domain algebra. They also involve descent and ascent through hierarchies.

7.1 Data Cubes

We start with an example with no overtly spatial component. Consider precipitation measurements taken over a day at four stations, distinguishing rain from snow and hail (figure 28). We will suppose that the amounts of precipitation are given in compatible units, e.g., equivalent millimeters of rainfall.

It is easy to calculate the total precipitation over all time (by type and station)

let *totTS* **be equiv** + **of** *amount* **by** *type, station*;

or by type

let *totT* **be equiv** + **of** *amount* **by** *type*;

or just the grand total

let *tot* **be red** + **of** *amount*;

These amounts are shown, once for each group, beside the relation in figure 28. (The total over all stations makes sense only as a step to taking an average, but we will suspend disbelief for the moment.) Gray et al. [26] call this sequence a *rollup*.

In fact, we can calculate all possible (sub)totals with some simple code. This amounts to $3^2 - 1$ totals, for the three fields, *type, station* and *time*. With judicious use of null values,

DataCube(*type*	*station*	*time*	*amount*)	*TotTS*	*totT*	*tot*
rain	1	0000	20	30	80	120
rain	1	0600	10			
rain	1	1200	0			
rain	1	1800	0			
rain	2	0000	0	10		
rain	2	0600	10			
rain	2	1200	0			
rain	2	1800	0			
rain	3	0000	0	30		
rain	3	0600	20			
rain	3	1200	10			
rain	3	1800	0			
rain	4	0000	10	10		
rain	4	0600	0			
rain	4	1200	0			
rain	4	1800	0			
snow	1	0000	0	10	30	
snow	1	0600	0			
snow	1	1200	10			
snow	1	1800	0			
snow	2	0000	0	10		
snow	2	0600	0			
snow	2	1200	10			
snow	2	1800	0			
snow	3	0000	0	0		
snow	3	0600	0			
snow	3	1200	0			
snow	3	1800	0			
snow	4	0000	0	10		
snow	4	0600	10			
snow	4	1200	0			
snow	4	1800	0			
hail	1	0000	0	5	10	
hail	1	0600	0			
hail	1	1200	0			
hail	1	1800	5			
hail	2	0000	0	0		
hail	2	0600	0			
hail	2	1200	0			
hail	2	1800	0			
hail	3	0000	0	5		
hail	3	0600	0			
hail	3	1200	0			
hail	3	1800	5			
hail	4	0000	0	0		
hail	4	0600	0			
hail	4	1200	0			
hail	4	1800	0			

Figure 28: The Precipitation Datacube

generated by the **union** operator, we can create a new tuple in the original relation for each total, and we do not need to recompute any sum (which the above domain algebra would do).

We write an equivalence reduction followed by an update for each of the three fields, *time, station* and *type*.

let *amount* **be** *tot*;

let *tot* **be equiv** + **of** *amount* **by** *type, station*;
update *DataCube* **add** [*type, station, amount*] **in** [*type, station, tot*] **in** *DataCube*;

let *tot* **be equiv** + **of** *amount* **by** *station, time*;
update *DataCube* **add** [*station, time, amount*] **in** [*station, time, tot*] **in** *DataCube*;

let *tot* **be equiv** + **of** *amount* **by** *type, time*;
update *DataCube* **add** [*type, time, amount*] **in** [*type, time, tot*] **in** *DataCube*;

This gives the fully expanded datacube, shown in figure 29. This time, only the tuples with nonzero precipitation have been shown from the original datacube: the others could have been omitted from the start without affecting the sums (although omissions would affect averages). The tuples with the aggregate values all have at least one $\mathcal{DC}$ null value. They are shown in three groups, in the order in which these groups are added by the code above.

This buildup of aggregate tuples can be imagined as adding a plane surface to a rectangular polyhedron (the "datacube") in each direction. Thus, the $3 \times 4 \times 4$ cube is expanded by adding first a 3×4 plane in the *time* direction, then a 4×5 plane in the *type* direction, and finally a 4×5 plane in the *station* direction. The resulting expanded datacube is $(3+1) \times (4+1) \times (4+1)$ in size. It contains all possible aggregates in the three dimensions of the datacube.

Figure 30 shows a picture of the buildup of these three faces of the datacube.

In general, a loop of d aggregations will calculate all $2^d - 1$ possible sets of aggregates in d dimensions. A datacube of size $\sum_0^d s_i$ expands to $\sum_0^d (1+s_i)$.

Gray et al. [26] call the above the *cube* operator, which they introduce as a generalization of rollup. (They were reluctant to use null values, and introduced an `ALL` value instead: the domain algebra can do this if desired.) The planar faces holding the generated aggregates are called *cross-tabs*.

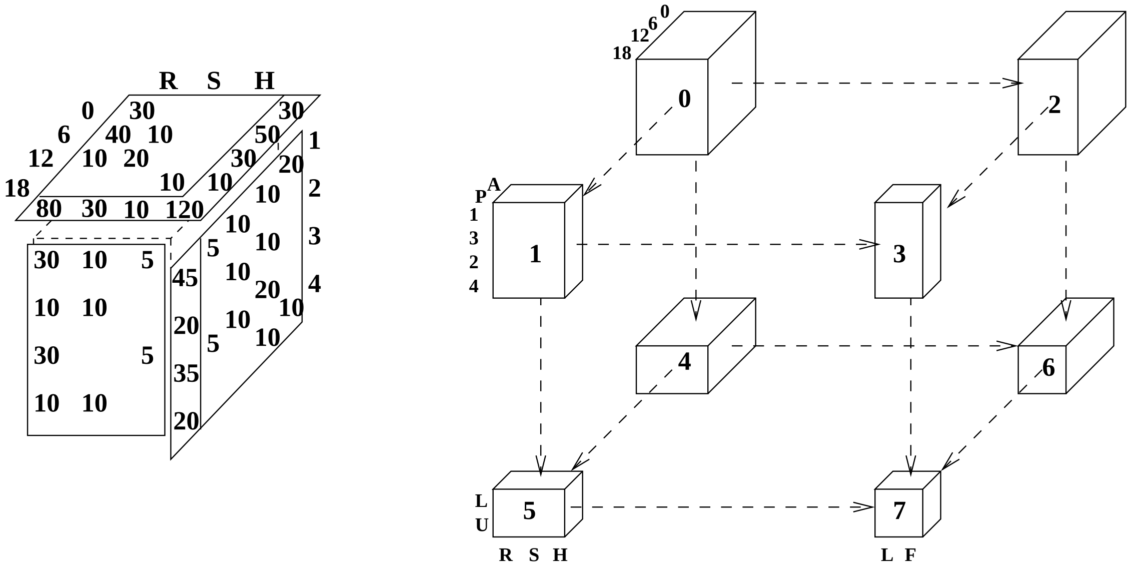
Of the 100 numbers in this example, only 49 are nonzero: 11 of the initial 48, 9 of the next 12, 15 of the next 20, and 14 of the final 20. In figure 29, the bulk of the tuples shown (38 of 49) are aggregates. This suggests that executing the above code all at the outset, and thus "materializing" the aggregates for storage, may not be the best practice. On the other hand, it could be expensive to materialize nothing but evaluate an aggregate each time it is needed. Considerable research is under way to determine strategies for partial pre-aggregation, but this is beyond the scope of this tutorial. (See, e.g. [32].)

Even the totals we have calculated may be too detailed. We may need to abstract each of the dimensions of the datacube into concept hierarchies. For example, the four times could be divided into `AM` and `PM`, the three types of precipitation could be divided into `liquid` and `frozen`, and the four stations could fall into regions `lower` ({`1, 3`}) and `upper` ({`2, 4`}), respectively.

These three hierarchies are captured in three auxiliary relations, which we call *rays* because, taken together with *DataCube*, they form what is known as a *star schema*.

DataCube(type	*station*	*time*	*amount)*
rain	1	0000	20
rain	1	0600	10
rain	2	0600	10
rain	3	0600	20
rain	3	1200	10
rain	4	0000	10
snow	1	1200	10
snow	2	1200	10
snow	4	0600	10
hail	1	1800	5
hail	3	1800	5
rain	1	$\mathcal{DC}$	30
rain	2	$\mathcal{DC}$	10
rain	3	$\mathcal{DC}$	30
rain	4	$\mathcal{DC}$	10
snow	1	$\mathcal{DC}$	10
snow	2	$\mathcal{DC}$	10
snow	4	$\mathcal{DC}$	10
hail	1	$\mathcal{DC}$	5
hail	3	$\mathcal{DC}$	5
$\mathcal{DC}$	1	0000	20
$\mathcal{DC}$	1	0600	10
$\mathcal{DC}$	1	1200	10
$\mathcal{DC}$	1	1800	5
$\mathcal{DC}$	1	$\mathcal{DC}$	45
$\mathcal{DC}$	2	0600	10
$\mathcal{DC}$	2	1200	10
$\mathcal{DC}$	2	$\mathcal{DC}$	20
$\mathcal{DC}$	3	0600	20
$\mathcal{DC}$	3	1200	10
$\mathcal{DC}$	3	1800	5
$\mathcal{DC}$	3	$\mathcal{DC}$	35
$\mathcal{DC}$	4	0000	10
$\mathcal{DC}$	4	0600	10
$\mathcal{DC}$	4	$\mathcal{DC}$	20
rain	$\mathcal{DC}$	0000	30
rain	$\mathcal{DC}$	0600	40
rain	$\mathcal{DC}$	1200	10
rain	$\mathcal{DC}$	$\mathcal{DC}$	80
snow	$\mathcal{DC}$	0600	10
snow	$\mathcal{DC}$	1200	20
snow	$\mathcal{DC}$	$\mathcal{DC}$	30
hail	$\mathcal{DC}$	1800	10
hail	$\mathcal{DC}$	$\mathcal{DC}$	10
$\mathcal{DC}$	$\mathcal{DC}$	0000	30
$\mathcal{DC}$	$\mathcal{DC}$	0600	50
$\mathcal{DC}$	$\mathcal{DC}$	1200	30
$\mathcal{DC}$	$\mathcal{DC}$	1800	10
$\mathcal{DC}$	$\mathcal{DC}$	$\mathcal{DC}$	120

Figure 29: The Precipitation Datacube Fully Aggregated



Datacube **Cube of Cubes**

Figure 30: The Datacube and a Cube of Cubes

TimeRay(*time*	*AP*)	*StationRay*(*station*	*UL*)	*TypeRay*(*type*	*LF*)
`0000`	`am`	`1`	`low`	`rain`	`liq`
`0006`	`am`	`3`	`low`	`snow`	`frz`
`0012`	`pm`	`2`	`upp`	`hail`	`frz`
`0018`	`pm`	`4`	`upp`		

In general, these ray relations can capture hierarchies of arbitrary depth, not just depth two. Or they can store auxiliary information about each of the three principal dimensions of the data cube.

The code to aggregate on all possible combinations of these coarser classifications is a sequence of three sets of statements, quite similar to the datacube generator, above.

let *amount* **be** *tot*;

let *tot* **be equiv** + **of** *amount* **by** *type, station, AP*;
let *time* **be** *AP*;
update *DataCube* **add** [*type, station, time, amount*] **in** [*type, station, AP, tot*]
  in (*DataCube* **join** *TimeRay*);

let *tot* **be equiv** + **of** *amount* **by** *LF, station, time*;
let *type* **be** *LF*;
update *DataCube* **add** [*type, station, time, amount*] **in** [*LF, station, time, tot*]
  in (*DataCube* **join** *TypeRay*);

let *tot* **be equiv** + **of** *amount* **by** *type, LU, time*;
let *station* **be** *LU*;
update *DataCube* **add** [*type, station, time, amount*] **in** [*type, LU, time, tot*]
  in (*DataCube* **join** *StationRay*);

DataCube			
(*type*	*station*	*time*	*amount*)
rain	1	am	30
rain	2	am	10
rain	3	am	20
rain	3	pm	10
rain	4	am	10
snow	1	pm	10
snow	2	pm	10
snow	4	am	10
hail	1	pm	5
hail	3	pm	5
liq	1	0000	20
liq	1	0006	10
liq	2	0006	10
liq	3	0006	20
liq	3	0012	10
liq	4	0000	10
frz	1	0012	10
frz	2	0012	10
frz	4	0006	10
frz	1	0018	5
frz	3	0018	5
liq	1	am	30
liq	2	am	10
liq	3	am	20
liq	3	pm	10
liq	4	am	10
frz	1	pm	15
frz	2	pm	10
frz	3	pm	5
frz	4	am	10

rain	low	0000	20
rain	low	0006	30
rain	low	0012	10
rain	upp	0000	10
rain	upp	0006	10
snow	low	0006	10
snow	low	0012	10
snow	upp	0012	10
hail	low	0018	10
rain	low	am	50
rain	low	pm	10
rain	upp	am	20
snow	low	am	10
snow	low	pm	10
snow	upp	pm	10
hail	low	pm	10
liq	low	0000	20
liq	low	0006	30
liq	low	0012	10
liq	upp	0000	10
liq	upp	0006	10
frz	low	0012	10
frz	low	0018	10
frz	upp	0006	10
frz	upp	0012	10
liq	low	am	50
liq	low	pm	10
liq	upp	am	20
frz	low	pm	20
frz	upp	am	10
frz	upp	pm	10

Figure 31: The Precipitation Datacube with Hierarchies

Figure 31 shows the tuples that this *adds* to *DataCube* (omitting the eleven originals), in three groups.

Figure 30 shows the cube of cubes that this process constructs from the original datacube. The first three statements create cube 1 from cube 0; then cubes 2 and 3 are built; followed by 4–7. If each hierarchy were three levels deep, the cube of cubes would be $3 \times 3 \times 3$ and three further join statements would be needed.

Generating any one of these cubes from one of finer classification is also called *rollup*.

Finally, we can use our earlier cube operation to compute the aggregation faces for each cube in the cube of cubes.

Once the generation is done, either of the cube-of-cubes or of the aggregation faces, rollup consists simply of selecting tuples with null values (or the dummy value, ALL) in the desired fields. The inverse operation is called *drill-down.* If we are presented, for instance, with the aggregation faces shown in figure 28 and want to "drill down" to the hidden numbers behind the aggregations, we use the field values that interest us to select all tuples with these values. This will give both the base data and their aggregate(s).

In any of these cubes, i.e., in the *DataCube* relation, we can select tuples with one or more values of one or more fields, and still get the aggregations. This is called *slice-and-dice*.

As with the cube operation, the issue of when to do the aggregation—before or after the slice-and-dice, for instance—is of practical importance due to the costs of computing and the sizes of the results, but this is not the programming issue of how to do it.

The aggregations we have looked at so far are numeric. As well as sums, we could find products (e.g., *and*s of probabilities), minima, maxima, boolean **and**s and **or**s, various averages (e.g., arithmetic, geometric, or harmonic), standard deviations, and more complicated aggregates such as *or*s of probabilities. Beyond numeric aggregates, we might be interested in aggregating *sets* of items. For example, the field to be aggregated might be *type* instead of *amount*, and we might want to know the set of precipitation types for each station, for instance.

This is done even more directly than numeric aggregation: it needs only three projections, followed by the three updates we used for the datacube. Here is the segment of the resulting datacube that gives precipitation types per station.

DataCube(*station*	*time*	*amount*	*type*)
:	:	:	:
1	$\mathcal{DC}$	$\mathcal{DC}$	`rain`
1	$\mathcal{DC}$	$\mathcal{DC}$	`snow`
1	$\mathcal{DC}$	$\mathcal{DC}$	`hail`
2	$\mathcal{DC}$	$\mathcal{DC}$	`rain`
2	$\mathcal{DC}$	$\mathcal{DC}$	`snow`
3	$\mathcal{DC}$	$\mathcal{DC}$	`rain`
3	$\mathcal{DC}$	$\mathcal{DC}$	`hail`
4	$\mathcal{DC}$	$\mathcal{DC}$	`rain`
4	$\mathcal{DC}$	$\mathcal{DC}$	`snow`

Both types of aggregation could be combined. For instance, we could define a *wet* region as one whose station accumulated over 30 units of precipitation, and ask what kinds of precipitation fall in wet regions.

DataCube(*station*	*time*	*amount*	*type*)
:	:	:	:
`wet`	$\mathcal{DC}$	$\mathcal{DC}$	`rain`
`wet`	$\mathcal{DC}$	$\mathcal{DC}$	`snow`
`wet`	$\mathcal{DC}$	$\mathcal{DC}$	`hail`
`dry`	$\mathcal{DC}$	$\mathcal{DC}$	`rain`
`dry`	$\mathcal{DC}$	$\mathcal{DC}$	`snow`

7.2 Spatial Data Cubes

Spatial OLAP adds the problem of aggregating spatial data [31]. For example, each of the four stations in the previous example could be replaced by a polygonal region representing its catchment area. For simplicity, we show these as triangles, and, for interest, the triangles overlap each other. Here are two of the relations, including the geometry, of the quad-edge data structure for the original four triangles. (The faces are numbered 1..4 after the stations, their corresponding complement faces are $U1..U4$, and the coordinates given for the faces are the positions of the stations themselves.)

VertFace(*edge*	*org*	*dest*	*left*	*right*)
a	*A*	*D*	3	*U*3
b	*C*	*F*	1	*U*1
c	*E*	*H*	4	*U*4
d	*G*	*B*	2	*U*2
e	*I*	*A*	3	*U*3
f	*J*	*C*	1	*U*1
g	*K*	*E*	4	*U*4
h	*L*	*G*	2	*U*2
i	*D*	*I*	3	*U*3
j	*F*	*J*	1	*U*1
k	*H*	*K*	4	*U*4
l	*B*	*L*	2	*U*2

Geom(*vf*	*x*	*y*)
A	1	2
B	2	1
C	8	1
D	9	2
E	9	8
F	8	9
G	2	9
H	1	8
I	5	6
J	4	5
K	5	4
L	6	5
1	7	5
2	3	5
3	5	3
4	5	7

The aggregation faces of a spatial datacube, with region as the aggregation field, would show combined shapes for each aggregate. Figure 32 shows the part of this result that aggregates *station* over all *amount*s by *type* and *time*. The aggregates are shown as sets and as shaded parts of the overall shape.

These spatial aggregations could be carried in the flat relational algebra, using a relation for the datacube and the three relations of the quad-edge data structure, but the joins needed become confusing and conceptually unwieldy. So instead we use nested relations and the notion of an abstract data type. The *spatialUnion* operator, discussed at the end of section 6.1, is a redop operator on the quad-edge structure, If we take this combination of data structure and operator as an abstract data type, we can use one instance of it per tuple in the *DataCube* relation. Then the aggregation can be done with a simple **equiv** *spatialUnion* **of** the field containing this abstract data type, and all thinking about joins can be buried in the implementation of nesting and data abstraction.

8 Spatial Data Mining

"Data mining" means knowledge discovery in databases, and is a branch of machine learning specialized to large quantities of data on secondary storage. It is a large field, and we touch on it in this tutorial only enough to give a springboard to its specialization, spatial data mining. The examples presented will be particularly simplistic, aimed at giving a quick grasp of the core of the field and not at all at the central problems of efficiency. These are, as usual, tackled by algorithm refinements and by good implementation of the underlying formalisms. We do not advance, for instance, beyond the level of Chapter 4 in Witten and Frank [50].

Note that in particular we do not try to advance the point of view that data mining would profit by considering causal relationships as mentioned in section 3.5.6. Instead, we limit ourselves to the major approaches of classification, association, and generalization dealt with in the literature. We conclude with spatial applications of some of these ideas.

type	*time*	*amount*	*station*
rain	0000	–	1,4
rain	0600	–	1,2,3
rain	1200	–	3
snow	0600	–	4
snow	1200	–	1,2
hail	1800	–	1,3

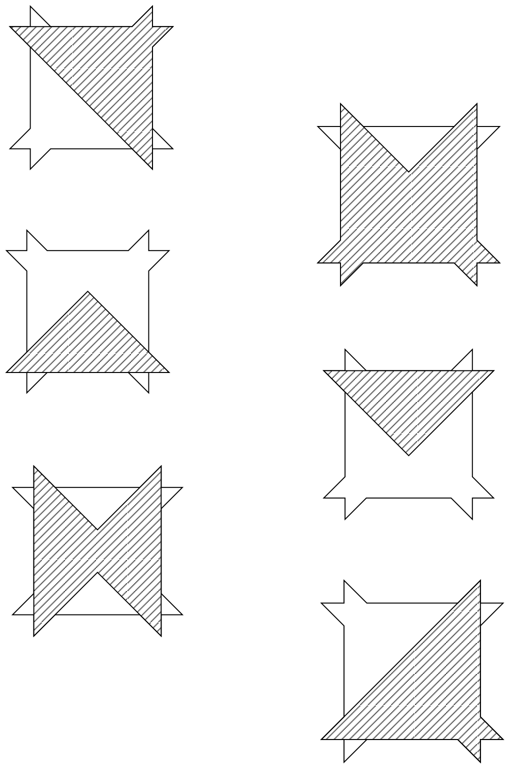


Figure 32: Regions Aggregated by *Type* and *Time*

8.1 Classification

The job of a classification algorithm is to find a procedure whereby a tuple with known fields but unknown class can be assigned to a class. The algorithm starts with a set of tuples whose fields and classes are both known (the "training data"). Here is a classic example [43].

Training	(*Outlook*	*Temperature*	*Humidity*	*Windy*	*Class*)
1	`sunny`	`hot`	`high`	`f`	`N`
2	`sunny`	`hot`	`high`	`t`	`N`
3	`overcast`	`hot`	`high`	`f`	`P`
4	`rain`	`mild`	`high`	`f`	`P`
5	`rain`	`cool`	`normal`	`f`	`P`
6	`rain`	`cool`	`normal`	`t`	`N`
7	`overcast`	`cool`	`normal`	`t`	`P`
8	`sunny`	`mild`	`high`	`f`	`N`
9	`sunny`	`cool`	`normal`	`f`	`P`
10	`rain`	`mild`	`normal`	`f`	`P`
11	`sunny`	`mild`	`normal`	`t`	`P`
12	`overcast`	`mild`	`high`	`t`	`P`
13	`overcast`	`hot`	`normal`	`f`	`P`
14	`rain`	`mild`	`high`	`t`	`N`

(The first column is a tuple identifier for later reference, and is not part of the data.) The first four fields are the data supplied for classification, and *Class* indicates whether the tuple is a positive or a negative instance of the classification sought. Once the classification procedure has been discovered, typical applications might be to find the *Class* of the tuple

(`sunny`, `cool`, `high`, `t`, `?`).

In this example, *Class* has only two values, `N` and `P`. In general, the tuples may classify into several classes.

Classification algorithms make heavy use of aggregates, and can start with a datacube. We do this here, for clarity, although in practice the whole datacube is never used. Figure 33 shows the datacube for the above dataset. The datacube shown stores pairs of integers as its basic elements. The first integer of the pair is the count of the number of `N` tuples for the given values of the other fields. The second integer is the count of `P` tuples. Ordinarily, these counts would never exceed 1, but we have reduced a four-dimensional problem to three by eliminating the *Temperature* field, so two of the tuples repeat the values of the other fields.

The first classification procedure we look at builds a *decision tree*. The "theory" extracted from the training data is thus a tree, which will be applied to future tuples of unknown class. The decision tree is constructed to minimize information. This seems contrary, but information theory defines information to be the measure of surprise one experiences in receiving a message (no surprise: we knew it already; no information), and surprise is what a classifier should eliminate. Information theory is statistical, and information is a function of the probability of an outcome, $-p \lg p$, to be precise. This quantity is positive, since $p \leq 1$, is zero in the two cases of certainty ($p = 0$ and $p = 1$), and is maximum when uncertainty is greatest ($p = 1/2$). The information we will be minimizing is the expected information needed to classify any new tuple.

The information content of the unprocessed *Training* tuples, vis-a-vis the classification into `N`egative and `P`ositive instances is

$$-\frac{5}{14}\lg\frac{5}{14} - \frac{9}{14}\lg\frac{9}{14} = (14\lg 14 - 5\lg 5 - 9\lg 9)/14 = 0.940$$

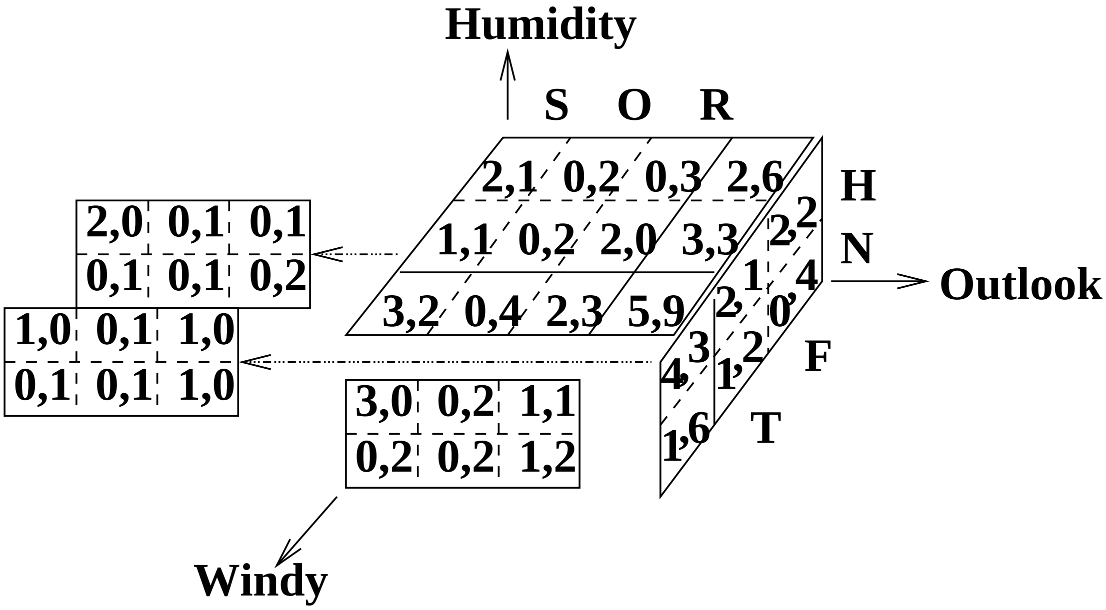


Figure 33: The DataCube for the Weather Classification

bits, where the base of the logarithm, lg, is 2. The 5 and the 9 are from the final aggregate in the datacube, the common corner of the aggregate faces. Since we will need it frequently, we define

$$\mathcal{I}(n,p) = ((n+p)\lg(n+p) - n\lg n - p\lg p)/(n+p)$$

A decision tree builder looks at each field to find the one which, if made the root of the tree, would result in the least information being needed for a subsequent search. The expected value of this information for the *Outlook* field is $5\mathcal{I}(3,2)/14 + 4\mathcal{I}(0,4)/14 + 5\mathcal{I}(2,3)/14 = 0.694$, where the arguments for the three instances of $\mathcal{I}(..)$ come from the aggregate edge of the datacube for *Outlook*. Similar calculations for the other two edges give

$$\frac{7}{14}\mathcal{I}(4,3) + \frac{7}{14}\mathcal{I}(1,6) = 0.788$$

for *Humidity*, and

$$\frac{8}{14}\mathcal{I}(2,6) + \frac{6}{14}\mathcal{I}(3,3) = 0.892$$

for *Windy*. Since *Outlook* gives the smallest of these, it becomes the root.

To find the subtrees, we repeat this process for the two aggregate planes containing this edge for *Outlook*. For *Outlook*=`sunny`, this requires us to compare

$$\frac{3}{5}\mathcal{I}(2,1) + \frac{2}{5}\mathcal{I}(1,1) = 0.951$$

(*Windy*) with

$$\frac{3}{5}\mathcal{I}(3,0) + \frac{2}{5}\mathcal{I}(0,2) = 0$$

(*Humidity*), which is smaller, so *Humidity* forms the subtree below *Outlook*=`sunny`.

For *Outlook*=`overcast`, the total information, $\mathcal{I}(0,4)$, is already zero, so no subtree is needed: every *Class* for *Outlook*=`overcast` is `P`.

Finally, for *Outlook*=`rain`, the comparisons

$$\frac{3}{5}\mathcal{I}(0,3)+\frac{2}{5}\mathcal{I}(2,0)=0$$

(*Windy*) with

$$\frac{2}{5}\mathcal{I}(1,1)+\frac{3}{5}\mathcal{I}(1,2)=0.951$$

(*Humidity*) give the subtree to *Windy*.

The upper tree in figure 34 is the final decision tree. This can be used as the final classifier, or it can be converted to a set of rules by reading off each path as a conjunction.

if *Outlook*=`sunny` **and** *Humidity*=`high` **then** *Class*=`N`
if *Outlook*=`sunny` **and** *Humidity*=`normal` **then** *Class*=`P`
if *Outlook*=`overcast` **then** *Class*=`P`
if *Outlook*=`rain` **and** *Windy*=`f` **then** *Class*=`P`
if *Outlook*=`rain` **and** *Windy*=`t` **then** *Class*=`N`

(Doing the full four-dimensional datacube, which includes *Temperature*, gives the same result. We can see from the interior of the datacube that *Temperature* does not split any entry into both an `N` and a `P`, so it has no effect.)

The domain algebra needed to find the expected information along any row or column of the aggregate faces is similar to that needed to compute the datacube in the first place, but involving

$$Sum((n+p)\lg(n+p)-n\lg n-p\lg p))/Sum(n+p).$$

A faster algorithm computes only the aggregates needed by the above processing.

A simpler, but less complete, classifier is the *1R* ("one-rule") approach, which looks at only one field. This uses the aggregate edges of the datacube to find how many errors are caused by simply asserting, for example, that all tuples where *Outlook*=`sunny` are classified `N`. The field with the fewest errors for all values is chosen. The total error is

$$Sum(n \min p).$$

For *Outlook*, this is 5−3 + 4−4 + 5−3 = 4; for *Humidity* it is 7−4 + 7−6 = 4; for *Windy* it is 8−6 + 6−3 = 5 (and for *Temperature*, left out of the datacube, it is 4−2 + 6−4 + 4−3 = 5). Breaking the tie for smallest error between *Outlook* and *Humidity*, we choose *Outlook* and get the (approximate) classifier

if *Outlook*=`sunny` **then** *Class*=`N`
if *Outlook*=`overcast` **then** *Class*=`P`
if *Outlook*=`rain` **then** *Class*=`P` The lower tree in figure 34 is the equivalent of these rules in tree form.

A third classifier simplifies in another direction. One-rule supposed that we could isolate a single field which controls the classes, neglecting other fields. The *Naive Bayes* approach assumes that all fields contribute equally. Again, we need only the aggregate edges of the datacube. We convert the counts to probabilities by dividing each `N` count by the total number of `N`s for that field, and the `P` count by the total `P`s for the field. We do this for each field, and so for each of the edges where the aggregate faces of the datacube meet.

With these probabilities, and with no further processing, we can treat any new tuple that comes along needing classification. For each field in the new tuple, we look up the probability of that value occurring for that field if *Class* were `N`, and again for the case that *Class* were

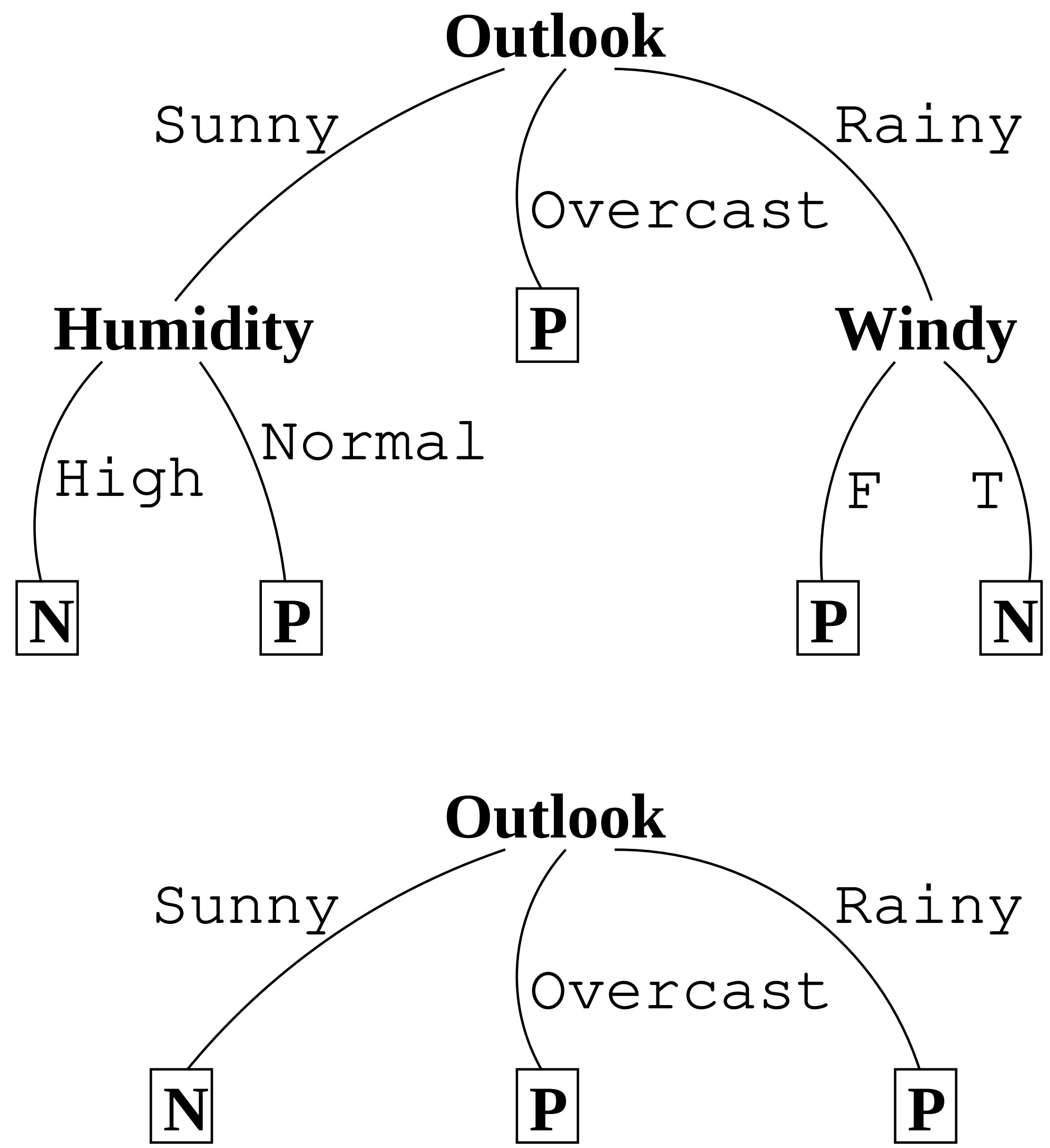


Figure 34: Decision Trees for the Weather Classification

P. The probability of the tuple being N (respectively, P) is the product of these probabilities multiplied (this is where Bayes' prescription is introduced) by the overall probability of N, 5/14 (respectively, P, 9/14). The larger product wins.

Thus, for the tuple

(sunny, cool, high, t, ?)

the two products are

$$\frac{3}{5} \times \frac{1}{5} \times \frac{4}{5} \times \frac{3}{5} \times \frac{5}{14} = 0.0206$$

for N, and

$$\frac{2}{9} \times \frac{3}{9} \times \frac{3}{9} \times \frac{3}{9} \times \frac{9}{14} = 0.0053$$

for P, so the tuple is classified N.

Figure 35 shows the parts of the datacube used by each of the above three classifiers, and the results of the calculations.

The following unclassified tuples are classified as shown by the three methods. These do not necessarily agree.

Outlook	*Temperature*	*Humidity*	*Windy*	*Class*		
				D. Tree	*1R*	*Bayes*
sunny	cool	high	t	N	N	N
rain	hot	high	f	Y	Y	N
rain	hot	high	t	N	Y	N

A fourth classifier uses the training data itself as the "theory", and needs no aggregates. This is *instance-based learning*, which uses distance measures on the multidimensional space of the training data (in our case, *Training* has four dimensions) to find the training tuple nearest to the new tuple to be classified. The new tuple takes the *Class* of this neighbour. If the data is noisy, more than one neighbour may be used, and a vote taken, possibly weighted by distance.

Figure 36 is four four-dimensional maps, flattened into 2D, which help us visualize the weather data in the above examples. The original training data are given in the top two maps. The N regions predicted by the decision tree are shaded in the upper left map, and the N regions predicted by 1R are shaded in the upper right map. The lower two maps show predictions only, Bayes on the left and IBL on the right. Note that Bayes incorrectly predicts one of the original training points. The IBL predictions are iterated: the training data form the 0th iteration. At the first iteration, some verdicts are tied (shown as "?"), because equal numbers of nearest neighbours contradict each other. Some predictions cannot be made because no training points are nearest neighbours (shown as "-"). Most of these are cleared up in a second iteration, which uses the predictions of the first iteration (thereby effectively going to second-nearest neighbours to get the classification). One prediction needs a third iteration, and two never get resolved by nearest neighbours.

So far, there is no spatial component to these classification techniques. To extend them to spatial data mining, we replace the counts in the analyses by regional areas. This assigns importance according to how much ground is covered by a tuple, instead of weighing each tuple equally. The datacube changes, but the analyses do not. Figure 37 shows the new datacube resulting from the regions shown in figure 1: the tuple identifiers from the *Training* relation at the beginning of this section are identified with the region numbers in figure 1. The upper number in each box is the (aggregate) area for the N tuples with the given values of the other fields, and the lower number is the area for the P tuples. The regions do not

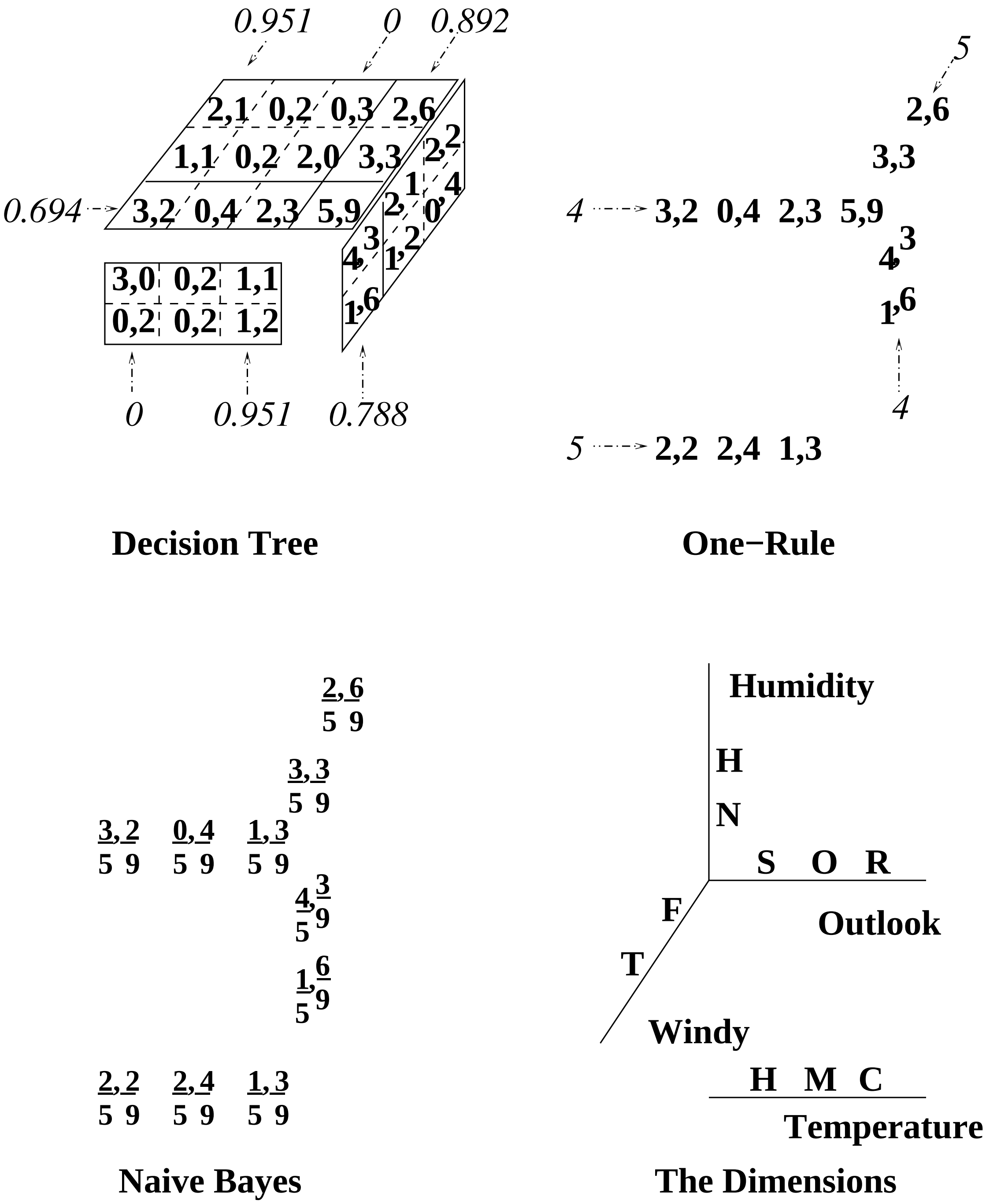


Figure 35: Parts of DataCube Used by Decision Tree, 1R, and Bayes

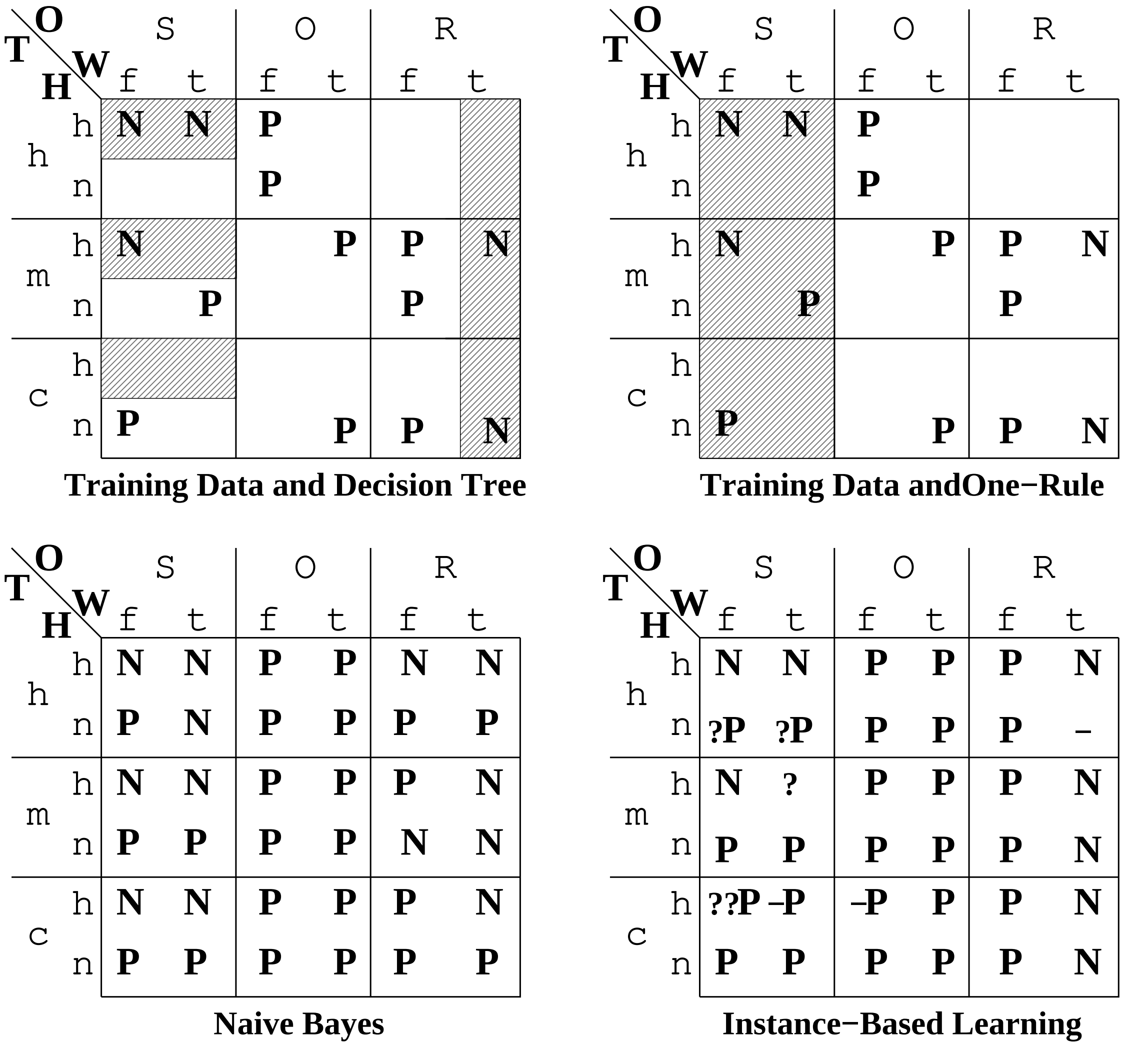


Figure 36: The Four-Dimensional Weather Training Data

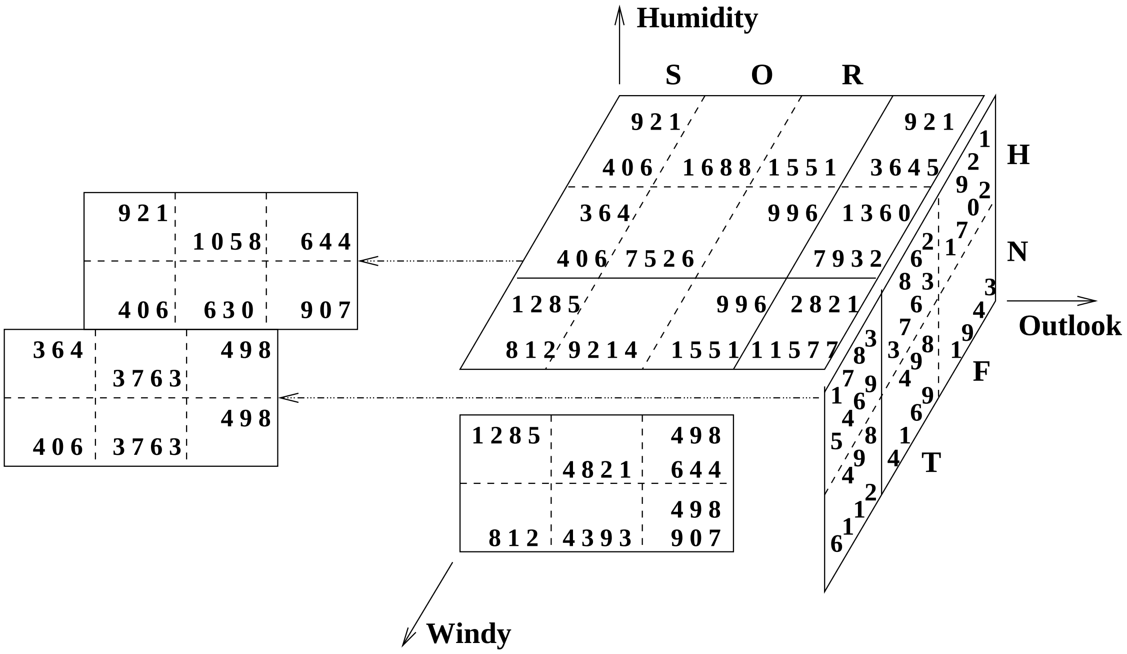


Figure 37: DataCube for *Training* Weighted by Region Areas

appear in the datacube except through their areas. They can be considered an additional field in *Training*, clearly a nested abstract data type.

The results of the classification methods on this spatially weighted data are:

- the decision tree is the same, mainly because of the absence of `N`s for *Outlook*=`overcast` and the clear categorizations by *Humidity* when it is `sunny` and by *Windy* when it is `rainy`;
- the one-rule is the same, for the same reasons;
- the Bayes classification is largely the same, but differs in six places (the numbers of `N`s and `P`s are the same as before, which is strange since the overall probability of `N` is much lower now: many of the Bayesian decisions are very closely tied, and the result is what happens in some riding-based elections);
- instance-based learning is unaffected since weights play no part.

The significance of using areas as weights, instead of just counting tuples, is that the size of a region reporting given weather conditions is now a part of its influence on the outcome.

8.2 Association

Association mining attempts to find rules of the form “if one set of items occurs in a given situation, then a second set of items also occurs”, for specific sets of items. The usual motivation offered is to help retail stores to discover associations among sets of products in customers’ shopping baskets. Here is an example which appeared in the early literature.

ShoppingBaskets(*xact*	*item*)	*confden*
8	beans	2
9	beans	2
2	beer	2
5	beer	2
1	bread	5
2	bread	5
3	bread	5
4	bread	5
7	bread	5
1	butter	5
2	butter	5
3	butter	5
4	butter	5
6	butter	5
1	coffee	3
3	coffee	3
4	coffee	3
2	milk	2
4	milk	2
9	rice	2
10	rice	2

(The virtual field, *confden*, shown, is used a little later.)

Examples of an association rule in this case might be

if {`bread, coffee`} *then* {`butter`}
if {`butter`} *then* {`bread`}

The set before *then* is the *antecedent*, and the set after *then* is the *consequent*. Associated with each is a set of transactions, which is the intersection of the sets associated with the individual items. This intersection must be non-empty for any set of items considered.

Since the general problem is to find *sets* of items that associate, the computational cost is potentially exponential. This worst case is not usually reached because the pairs of associated sets must share at least one transaction. We further reduce the number of results (and the amount of work) by imposing two, user-specified, criteria.

The first criterion is that the cover exceed a given minimum. The *cover set* of a set of items is defined to be the associated set of transactions. The *cover* of a set of items is the size of the cover set, and the cover of a rule is the size of the intersection of the cover sets of the antecedent and of the consequent. Requiring the cover to be of a minimum size specifies that the rule must be based on a threshhold number of entries in the database. Users may specify relative cover, called support. The *support* of a set or of a rule is its cover divided by the total number of transactions.

The covers of {`bread`}, {`butter`}, {`bread, butter`}, {`coffee`}, {`bread, coffee`} and {`bread, butter, coffee`} in the above example are, respectively, 5, 5, 4, 3, 3, and 3. The supports are, respectively, 0.5, 0.5, 0.4, 0.3, and 0.3, because there are ten transactions in the example.

The second criterion is that the confidence exceed a given minimum. The *confidence* of a rule is its cover divided by the cover of its antecedent set. This requirement sets a threshhold on the number of transactions associated with the intersection of antecedent and consequent relative to the number of transactions associated with the antecedent. We are more confident in a rule if most of the transactions in the antecedent figure in the rule as a whole, rather than if only a few of the transactions associated with the antecedent participate in both sides.

The confidence of the first rule, above, is $3/3 = 1.0$. The confidence of the second rule is $4/5 = 0.8$.

We can specify these quantities in the domain algebra.

let *cover* **be equiv** + **of** 1 **by** *item, item′*;
let *confden* **be equiv** + **of** 1 **by** *item*;

Confden is the denominator of the confidence, while *cover* is the numerator. It is not an error that they are both specified by the same expression. This is an idiom of the domain algebra: *confden* is intended to be actualized before a join, while *cover* will be actualized after the join. For this reason, we anticipated by showing *confden* in the table for *ShoppingBaskets*, above.

Here is the rest of the code for the simple case of singleton sets in the antecedents and consequents.

let *item′* **be** *item*;
SingletonRules <− [*item′*, *item*]
where *item′* $\neq$ *item* **and** *cover*/ *confden* $\geq$ *minconf* **in** (*ShoppingBaskets* **join** [*xact, item′*, *confden*] **where** *confden* $\geq$ *mincover* **in** *ShoppingBaskets*);

To make the whole process clear, we will show the result of this code as if the two threshholds, *mincover* and *minconf*, were both zero. Then we will see what happens when they are set up to 3 and 0.8, respectively.

SingletonRules(*item′*	*item*)	*cover*/ *confden*	*xacts*
beans	rice	1/2	9
beer	bread	1/2	2
beer	butter	1/2	2
beer	milk	1/2	2
bread	beer	1/5	2
bread	butter	4/5	1,2,3,4
bread	coffee	3/5	1,3,4
bread	milk	2/5	2,4
butter	beer	1/5	2
butter	bread	4/5	1,2,3,4
butter	coffee	3/5	1,3,4
butter	milk	2/5	2,4
coffee	bread	3/3	1,3,4
coffee	butter	3/3	1,3,4
coffee	milk	1/3	4
milk	beer	1/2	2
milk	bread	2/2	2,4
milk	butter	2/2	2,4
milk	coffee	1/2	4
rice	beans	1/2	9

Here, *confden* is the cover of *item′* in *SingletonRules*, and *cover* is the number of *xacts* shared by *item′* and *item*.

If *mincover* is set to 3, no tuple with a denominator, *confden*, less than 3 appears: this removes nine of the above 20 tuples. If *minconf* is set to 0.8, all but four of the remaining tuples go, leaving us with the rules

if {bread} *then* {butter}

if {`butter`} *then* {`bread`}
if {`coffee`} *then* {`bread`}
if {`coffee`} *then* {`butter`}

This gives the idea, but we must go on to deal with all possible *sets* of items. We do this in two stages. First, we find all possible item sets and their associated transaction sets and covers. Then we use an adaptation of the above code to discover the rules. The code we show for this procedure is not optimally efficient; we leave the touchups as an exercise for the reader.

To generate and use all subsets of items, we use nested relations to ease our thinking. We must first create a nested relation from *ShoppingBaskets*, and this requires one new operation we did not discuss in section 5.4. To add a level of nesting, we add a **relation** operator to the domain algebra. This pushes a named set of fields one level down, and creates a singleton relation in each top-level tuple with the name given after the **let** keyword. We need

let *items* **be relation**(*item*);
let *xactset* **be relation**(*xact*);
let *xacts* **be equiv union of** *xactset* **by** *item*;

followed by the actualization

SBsets <− [*items, xacts*] **in** *ShoppingBaskets*;

Actually, to use *mincover* to eliminate most of the tuples, we replace this last line by more restrictive code.

let *cover* **be** [**red** + **of** 1] **in** *xacts*;
SBsets <− [*items, xacts*] **where** *cover* ≥ *mincover* **in** *ShoppingBaskets*;

(Note that we use anonymity in the first of these statements to lift the level of the field in *cover* so that *cover* becomes an integer-valued field, not nested, and can be tested against *mincover* in the second line in the normal way.)

This reformats the original relation. We now go on to generate all subsets above the minimum cover. (Any tuple that we eliminated with the test on *cover* cannot contribute further, because merging sets by intersection can only reduce the cover further.) We find the closure of *SBsets* under set union of the *items* sets and set intersection of the *xacts* sets. As with most closure operations, this is most clearly done by a recursive view. First, some domain algebra.

let *items′* **be** *items*;
let *xacts′* **be** *xacts*;
let *items″* **be** *items* **union** *items′*;
let *xacts″* **be** *xacts* **join** *xacts′*;
let *items* **be** *items″*;
let *xacts* **be** *xacts″*;

Next, the recursive view.

SBsetsClos **is** *SBsets* **union** [*items, xacts*] **in** [*items″*, *xacts″*]
where [] **in** *xacts″*
in *SBsets* **join** [*items′*, *xacts′*] **in** *SBsetsClos*;

With *mincover* at 3, this produces the nested relation

SBsetsClos(	*items* (*item*)	*xacts*) (*xact*)	*confden*
	bread	1	5
		2	5
		3	5
		4	5
		7	5
	butter	1	5
		2	5
		3	5
		4	5
		6	5
	coffee	1	3
		3	3
		4	3
	bread butter	1 2 3 4	4 4 4 4
	bread coffee	1 3 4	3 3 3
	butter coffee	1 3 4	3 3 3
	bread butter coffee	1 3 4	3 3 3

(The first three tuples of the seven are the initial *SBsets*.)

Again, we have shown *confden* for future reference. This must be generated by slightly different code. For each *xacts* relation, one per tuple of the nested relation, we just count its tuples.

let *confden* **be** [**red** + **of** 1] **in** *xacts*;

As before, we have effectively the same specification of *cover*

let *cover* **be** [**red** + **of** 1] **in** $xacts''$;

This leads us into the second part of the calculation, the modification of the above flat-relation code for nested relations. (We do not repeat the above specifications for $items'$, $xacts'$, or $xacts''$.)

GeneralRules <− [$items'$, *items*]
where (**not**(*items* **comp** $items'$) **and** [] **in** $xacts''$ **and** $cover/confden \geq minconf$)
in (*SBsetsClos* **join** [$xacts'$, $items'$, *confden*] **where** $confden \geq mincover$ **in** *SBsetsClos*);

(Note that the composition operator, **comp**, on two relations with the same fields, produces a nullary relation, which we have decreed in section 6.1.1 to be a boolean. Therefore we can negate it with **not**.)

Here are the results, controlled by *mincover*, but before selecting with *minconf*.

GeneralRules(	*items′*	*items)*	*xacts″*	*cover/confden*
	(item)	*(item)*	*(xact)*	
	`bread`	`butter`	1	4/5
			2	
			3	
			4	
	`bread`	`coffee`	1	3/5
			3	
			4	
	`bread`	`butter`	1	3/5
		`coffee`	3	
			4	
	`butter`	`bread`	1	4/5
			2	
			3	
			4	
	`butter`	`coffee`	1	3/5
			3	
			4	
	`butter`	`butter`	1	3/5
		`coffee`	3	
			4	
	`coffee`	`bread`	1	3/3
			3	
			4	
	`coffee`	`butter`	1	3/3
			3	
			4	
	`coffee`	`bread`	1	3/3
		`butter`	3	
			4	
	`bread`	`coffee`	1	1/4
	`butter`			
	`bread`	`butter`	1	3/3
	`coffee`		3	
			4	
	`butter`	`bread`	1	3/3
	`coffee`		3	
			4	

From these eleven tuples the confidence threshold selects seven rules, the four we had above for the singleton sets, plus the following three.

if {`coffee`} *then* {`bread, butter`}
if {`bread, coffee`} *then* {`butter`}
if {`butter, coffee`} *then* {`bread`}

A more complicated example introduces name-value pairs into the sets, but the calculation is not significantly changed. For example, the weather data of section 8.1 could be mined for association. Association mining can thus be seen as a generalization of classification mining, which associated the other fields with single values of a favoured, *Class* field.

An example of spatial association mining can be developed along the same lines as the example of spatial classification mining, by using areas as weights in the calculation of cover and confidence. We could, for instance, replace *ShoppingBaskets* by the isomorphic relation, *SpeciesDistrib*, using the areas of the first ten regions in figure 1.

SpeciesDistrib(*region*	*species*)	*confden*
8	beaver	994
9	beaver	994
2	bees	683
5	bees	683
1	deer	6739
2	deer	6739
3	deer	6739
4	deer	6739
7	deer	6739
1	cedar	3474
2	cedar	3474
3	cedar	3474
4	cedar	3474
6	cedar	3474
1	cougar	2346
3	cougar	2346
4	cougar	2346
2	mice	1008
4	mice	1008
9	fisher	683
10	fisher	683

We have again anticipated by showing *confden*, which is the sum of the areas of the *region*s in figure 1.

The significance of replacing counts by regional areas is that weight is given to the area containing a population of a given species, rather than weighting all regions equally, irrespective of size.

We show how this affects the singleton rules, and leave the mining of the remaining subsets to the reader.

SingletonRules(*item'*	*item*)	*cover/confden*	
beaver	fisher	406/683	=0.59
bees	deer	364/994	=0.37
bees	cedar	364/994	=0.37
bees	mice	364/994	=0.37
deer	bees	364/6739	=0.5
deer	cedar	2976/6739	=0.44
deer	cougar	2346/6739	=0.35
deer	mice	1008/6739	=0.15
cedar	bees	364/3474	=0.10
cedar	deer	2976/3474	=0.86
cedar	cougar	2346/3474	=0.68
cedar	mice	1008/3474	=0.29
cougar	deer	2346/2346	=1.00
cougar	cedar	2346/2346	=1.00
cougar	mice	644/2346	=0.27
mice	bees	364/1008	=0.36
mice	deer	1008/1008	=1.00
mice	cedar	1008/1008	=1.00
mice	cougar	644/1008	=0.64
fisher	beaver	406/683	=0.59

Using the same *mincover* and *minconf* threshholds, this gives three of the singleton rules we had before.

if {`cedar`} *then* {`deer`}
if {`cougar`} *then* {`deer`}
if {`cougar`} *then* {`cedar`}

(The rule we no longer have enough confidence in, because the supporting area is too small, is

if {`deer`} *then* {`cedar`}.)

8.3 Generalization

Another way to mine data is to generalize each tuple and count the numbers of identical tuples that result. These counts can be taken as votes supporting each distinct generalization. If the result still contains too many tuples to make simple sense, generalize further. Ways to generalize include maintaining a hierarchy of values for each field and moving up the hierarchies; or omitting fields altogether. We might do the latter for fields that have no generalization hierarchy. Han, Cai, and Cercone [29] provide example hierarchies for a relation *Students*(*Name, GPA, Status, Major, Birthplace*), which we show in figure 38.

With these concept hierarchies, a tuple,

`Wise, 3.9, freshman, literature, Toronto`

could generalize to

`--, excellent, freshman, literature, Toronto, 1`

where the *Name* field has been omitted because it has no generalization hierarchy, and where a count, `1`, for this particular tuple, has been appended as a new field. If thirteen tuples from *Students* all generalize to this same tuple, we have

`--, excellent, freshman, literature, Toronto, 13`.

The original tuple could also generalize to

`--, 3.9, ugrad, literature, Toronto, 1`

GPA

- ANY
  - poor
    - 0..1.99
  - average
    - 2..2.99
  - good
    - 3..3.49
  - excellent
    - 3.5..4

Status

- ANY
  - ugrad
    - freshman
    - sophomore
    - junior
    - senior
  - grad
    - M.S.
    - M.A.
    - Ph.D.

Major

- ANY
  - art
    - literature
    - music
    - painting
  - science
    - biology
    - chemistry
    - physics

Birthplace

- ANY
  - Canada
    - Alberta
      - Calgary
      - Edmonton
    - B.C.
      - Burnaby
      - Vancouver
      - Victoria
    - Ontario
      - Hamilton
      - Toronto
      - Waterloo
  - Foreign
    - China
      - Beijing
      - Nanjing
      - Shanghai
    - India
      - Bombay
      - New Delhi
    - Germany
    - Sweden

Figure 38: Example Field Hierarchies

or to

    --, 3.9, freshman, art, Toronto, 1

etc. The generalizations could further generalize to, say,

    --, excellent, ugrad, art, Toronto, 1.

The concept hierarchy for *Birthplace* allows further levels of climbing. We might get

    --, excellent, ugrad, art, Ontario, 1

or, further up,

    --, excellent, ugrad, art, Canada, 1.

The ultimate generalization gives one tuple for the entire relation,

    --, ANY, ANY, ANY, ANY, 300

but this would clearly be *overgeneralizing* as far as extracting anything useful from mining is concerned. [29] allow the user to specify thresholds for the number of tuples in the final result and for the number of distinct values a field may have in the result.

This process of generalization is familiar to us from OLAP (section 7.1). It is essentially the cube of cubes shown in figure 30, with a new field, *Count*, added to aggregate on. (The omitting of the *Name* field can be achieved by giving it the trivial generalization hierarchy, `ANY` on top of all possible *Name* values.)

We can explore the datacube of figure 30 using these two kinds of generalization. If we accumulate *amount* while generalizing fields *type* and *time* and omitting field *station*, we get

DataCube(*type*	*station*	*time*	*amount*)
liq	ANY	a.m.	50
liq	ANY	p.m.	30
frz	ANY	a.m.	10
frz	ANY	p.m.	30

which tells us that the temperature was dropping over the region increasing the snow and hail to half of the precipitation in the afternoon from 16% in the morning.

If we generalize fields *station* and omit fields *type* and *time*,

DataCube(*type*	*station*	*time*	*amount*)
ANY	low	ANY	80
ANY	upp	ANY	40

we see that the lower stations had twice the precipitation the upper stations did. Generalizing *time* and omitting the other fields

DataCube(*type*	*station*	*time*	*amount*)
ANY	ANY	a.m.	60
ANY	ANY	p.m.	60

reveals that the precipitation was the same in the morning as in the afternoon.

To discover that the storm was moving in a southwesterly direction, below the divide between the upper and lower stations, would require some spatial knowledge and another distinction between easterly and westerly stations. This latter is in addition to the generalization we already made, giving `upper` vs. `lower`, and leads to generalization *graphs* as opposed to the tree hierarchies we have so far considered.

Spatial generalization mining is easily absorbed into this technique, since spatial data has natural hierarchies of containment. The *Birthplace* field in our earlier example is a case in point. It produced the largest concept tree and is the furthest from being complete. Rather than reinvent its concept hierarchy explicitly, we can refer to a map of the world and do point-in-polygon operations to generate the memberships of cities in provinces or countries, and the membership of provinces in Canada.

8.4 Predicate Mining, Illustrated for Spatial Data

The above suggestions for spatial mining extensions to the classical examples of data mining are severly limited, since they use spatial data only to compute weights for the classical processes. Section 3.5 discussed predicates, spatial and otherwise, and their hierarchies. We now consider some more ambitious examples of spatial data mining, involving predicates. It turns out, once again, that spatial predicates can be handled in just the same way as any other predicate.

We start by using generalization to convert binary predicates, such as proximity of two features or their visibility from each other, to unary predicates, such as proximity of a feature to a certain class of features. This will be useful, because the resulting unary predicates can be mined by the methods we have discussed above. We show this in the case of association mining, and produce from the results some general implications among predicates. We also show how special treatment of these predicates can be used for classification mining.

8.4.1 Generalization and Predicate Simplification

We use a moderately complicated example, involving the spatial binary predicate *Visible*, the spatial ternary predicates *Distance* and *Direction* (which describe two objects and their numerical distance and direction from each other), and the binary nonspatial predicate *Townsize*, which assigns a population to each town. We will use thresholds to reduce the ternary predicates to the respective binary predicates *CloseTo* and *WestOf*, and then we will use generalization to reduce these to unary predicates which we will pass on to the next section for association mining.

We start with three relations that we can suppose to have been derived from a map and related data, represented as in section 4. *FromMap* lists features and the longitudes and latitudes of their centres of mass. *TownSize* provides auxiliary information about the populations of features that are towns. *Visible* gives the result of involved geospatial calculations that determine which features are visible from others. These relations are shown in figure 39.

The source of these three relations is the spatial data which we intend to mine. In addition, we supply some background knowledge in the form of a concept hierarchy, *ConceptHier*, which we will use to generalize the features used in the predicates. This is also shown in figure 39. In addition, the concept hierarchy contains other elements which can be represented compactly by domain algebra statements:

let *PopSize* **be if** *Pop*≥250000 **then** `"large"` **else**
  if *Pop*≥10000 **then** `"medium"` **else** `"small"`;
let *Dir* **be if** 45≤*Angle* **and** *Angle*<135 **then** `"north"` **else**
  if 135≤*Angle* **and***Angle*<225 **then** `"west"` **else**
  if 225≤*Angle* **and***Angle*<315 **then** `"south"` **else** `"east"`;

We can put *FromMap* together with itself using some joins, selections to find towns in relation to other features, and some more geometry to calculate distances and angles. Here is a result, which anticipates the next paragraph by including only features that are close to the towns. This gives the ternary predicates, *Distance* and *Direction*, in a single relation for conciseness.

FromMap

(*Feature*	*Lat*	*Long*)
Regina	50-27	104-37
Sault Ste Marie	46-31	84-20
Winnipeg	49-53	97-09
Vancouver	49-15	123-07
Schumacher	48-28	81-20
Estevan	49-08	102-59
Red Deer	52-16	113-48
Garibaldi	49-58	123-09
Hinton	53-25	117-34
Lethbridge	49-42	112-49
TC(Sask 1)	50-27	
US border 49	49-00	
Laird	46-24	84-04
US(Mich)75		84-20
US border 46	46-29	
Lake Superior	47-00	88-00
TC(Man 1)	49-52	
TC (BC 1)	49-15	
Georgia Strait	50-00	125-00
McIntyre	48-28	81-15
Alta 2		113-51
Coast Mtns	56-00	130-00
Rocky Mtns	56-00	122-00

ConceptHier

(*Feature*	*FeatGen*)
Regina	town
Sault Ste Marie	town
Winnipeg	town
Vancouver	town
Schumacher	town
Estevan	town
Red Deer	town
Garibaldi	town
Hinton	town
Lethbridge	town
Coast Mtns	mountains
Rocky Mtns	mountains
Georgia Strait	water
Lake Superior	water
US border 49	national bdy
US border 46	national bdy
TC(Sask 1)	autoroute
US(Mich)75	autoroute
TC(Man 1)	autoroute
TC(BC 1)	autoroute
Alta 2	autoroute
Laird	mine
McIntyre	mine

TownSize

(*Town*	*Pop*)
Regina	250000
Sault Ste Marie	100000
Winnipeg	750000
Vancouver	1000000
Schumacher	500
Estevan	10000
Red Deer	50000
Garibaldi	100
Hinton	10000
Lethbridge	60000

Visible

(*Town*	*Feature*)
Hinton	Rocky Mtns
Garibaldi	Coast Mtns

Figure 39: Three Relations From a GIS, and a Concept Hierarchy

Displacement					
(*Town*	*Feature*	*FeatGen*	*Dist*	*Angle*	*Dir*)
Regina	TC(Sask 1)	autoroute	0	270	south
Regina	US border 49	national bdy	160	270	south
Sault Ste Marie	Laird	mine	25	300	south
Sault Ste Marie	US(Mich)75	autoroute	5	270	south
Sault Ste Marie	US border 46	national bdy	0	270	south
Sault Ste Marie	Lake Superior	water	0	150	west
Winnipeg	TC(Man 1)	autoroute	10	270	south
Winnipeg	US border 49	national bdy	100	270	south
Vancouver	TC (BC 1)	autoroute	through	0	east
Vancouver	US border 49	national bdy	20	270	south
Vancouver	Georgia Strait	water	on	180	west
Schumacher	McIntyre	autoroute	10	0	east
Estevan	US border 49	national bdy	20	270	south
Red Deer	Alta 2	autoroute	6	180	west
Garibaldi	Coast Mtns	mountains	contain	0	east
Hinton	Rocky Mtns	mountains	30	210	west
Lethbridge	Rocky Mtns	mountains	100	180	west

We recall from section 3.5.6 that "closeness" is a relative notion, so we define it differently depending on the type of feature, as specified by *FeatGen*. Here is a possible definition.

let *CloseTo* **be if** *FeatGen*=`"national bdy"` **then** $Dist \leq 200$ **else**
if *FeatGen*=`"mountains"` **then** $Dist \leq 200$ **else**
if *FeatGen*=`"provincial bdy"` **then** $Dist \leq 50$ **else**
if *FeatGen*=`"mine"` **then** $Dist \leq 50$ **else**
if *FeatGen*=`"autoroute"` **then** $Dist \leq 10$;

Under this definition, every tuple in *Displacement* happens to satisfy *CloseTo*.

Now we can use the concept hierarchies to reduce the ternary predicates *Distance* and *Direction* given by *Displacement*, and the binary predicates *TownSize* and *Visible*, to a collection of unary predicates describing each town. For *TownSize* we define

let *Predicate* **be** *Popsize* **cat** `" town"`;

For *Visible* we define

let *Predicate* **be** *FeatGen* **cat** `" visible"`;

And for *Displacement* we need both

let *Predicate* **be** `"near "` **cat** *FeatGen*;

and

let *Predicate* **be** *FeatGen* **cat** `" to "` *Dir*;

The result of applying these four definitions of *Predicate* to the three relations and the concept hierarchy is the following seventeen unary predicates.

```
autoroute to east
autoroute to south
autoroute to west
large town
medium town
mine to south
mountains to east
mountains to west
mountains visible
national bdy to south
near autoroute
near mine
near mountains
near national bdy
near water
small town
water to west
```

Some of the results are given in the next section.

8.4.2 Association and Predicate Implication

If we suppose that, by a process of manual analysis which is beyond the automation considered in this tutorial, we have decided to focus on the seven of the above unary predicates that are restricted to directions only if there are mountains to the west, and ignoring nearness of mountains other than being visible or to the west, then the following is a result of the above.

AssociationPredicates

(*Town*	*Predicate*)
Regina	large town
Regina	near autoroute
Regina	near national bdy
Sault Ste Marie	near mine
Sault Ste Marie	near autoroute
Sault Ste Marie	near national bdy
Sault Ste Marie	near water
Winnipeg	large town
Winnipeg	near autoroute
Winnipeg	near national bdy
Vancouver	large town
Vancouver	near autoroute
Vancouver	near national bdy
Vancouver	near water
Schumacher	near autoroute
Estevan	near national bdy
Red Deer	near autoroute
Garibaldi	mountains visible
Hinton	mountains visible
Hinton	mountains to west
Lethbridge	mountains to west

Our motivation for this arbitrary selection of results is to save further space and analysis in this tutorial: the result is now isomorphic to *ShoppingBaskets* of section 8.2, under the mappings

xacts	$\leftrightarrow$	*Town*
1	$\leftrightarrow$	`Regina`
2	$\leftrightarrow$	`Sault Ste Marie`
3	$\leftrightarrow$	`Winnipeg`
4	$\leftrightarrow$	`Vancouver`
5	$\leftrightarrow$	`Schumacher`
6	$\leftrightarrow$	`Estevan`
7	$\leftrightarrow$	`Red Deer`
8	$\leftrightarrow$	`Garibaldi`
9	$\leftrightarrow$	`Hinton`
10	$\leftrightarrow$	`Lethbridge`
item	$\leftrightarrow$	*Predicate*
`coffee`	$\leftrightarrow$	`large town`
`bread`	$\leftrightarrow$	`near autoroute`
`butter`	$\leftrightarrow$	`near national bdy`
`milk`	$\leftrightarrow$	`near water`
`beer`	$\leftrightarrow$	`near mine`
`beans`	$\leftrightarrow$	`mountains visible`
`rice`	$\leftrightarrow$	`mountains to west`

We can then conclude the following association rules, with the same support and confidence factors, from the analysis of section 8.2.

`near autoroute` $\Leftrightarrow$ `near national bdy`
`large town` $\Rightarrow$ `near autoroute` **and** `near national bdy`
`near autoroute` **and** `large town` $\Rightarrow$ `near national bdy`
`near national bdy` **and** `large town` $\Rightarrow$ `near autoroute`

From this discussion, we find that association mining applies to unary predicates, which may be spatial or not, and which can be extracted from map and other data. These predicates may, of course, be written with universally quantified variable, x, if we like

`nearAutoroute`(x) $\Leftrightarrow$ `nearNationalBdy`(x)

and so on, if such a fashion is needed.

8.4.3 Classification and Predicate Approximation

In the above discussion of generalization and association mining we have used predicate approximation, but not for the important purpose outlined in section 3.5, which is to speed up processing, especially spatial processing. Since we have also not yet elaborated on spatial classification mining, we tackle both now (bearing in mind that the two are not indissolubly linked: spatial predicate approximation can speed up any kind of mining or query processing).

We pick a different example, but similar in structure to the example developed in the last two sections. There is a *Distance* relation, which can be derived from a map. There are two relations embodying a three-level concept hierarchy of features, *Grouping*, the first level, and *Categories*, the upper level, which also contains a threshold value below which the distance can be considered *closeTo* for each category of features. These three are shown in figure 40

Distance

(*Neighbourhood*	*Feature*	*Dist*)
Oromocto	Legislature	12
Oromocto	Town Hall	0
Oromocto	U.N.B.	10
Oromocto	CFB Gagetown	0
Rossdale	Legislature	12
Rossdale	U. Alberta	2
Rossdale	Klondike Pk	4
Crystal Beach	Parliament	15
Crystal Beach	Carleton U.	12
Crystal Beach	HQ Nortel	1
Richmond	Town Hall	0
Richmond	U.B.C.	8
Richmond	YVR	0
Dundas	Town Hall	0
Dundas	McMaster U.	0
Dundas	Stelco Inc	5
Milton-Park	Concordia U.	1
Milton-Park	Town Hall	1
Milton-Park	McGill U.	0
Milton-Park	Nortel	5
Sandy Hill	Parliament	2
Sandy Hill	U. Ottawa	1
Sandy Hill	Carleton U.	5
Sandy Hill	N.R.C.	3
Assiniboia	Legislature	5
Assiniboia	U. Manitoba	10
Assiniboia	Winnipeg Airport	0
Yorkville	Legislature	1
Yorkville	U. Toronto	0
Yorkville	HQ Bank of Montreal	3
Waterloo	Town Hall	0
Waterloo	U. Waterloo	0
Waterloo	Waterloo Maple	0
Ste-Foy	National Assembly	7
Ste-Foy	U. Laval	0
Ste-Foy	Desjardins	8
Rockcliffe Park	Parliament	4
Rockcliffe Park	U. Ottawa	3
Rockcliffe Park	Rockcliffe Airport	2
Ottawa South	Parliament	5
Ottawa South	Carleton U.	1
Ottawa South	Expt Farm	1
Verdun	Town Hall	0
Verdun	Concordia U.	3
Verdun	Nortel	1

Grouping

(*Feature*	*Group*	)
Carleton U.	univD	
CFB Gagetown	indus	
Concordia U.	univD	
Desjardins	indus	
Expt Farm	indus	
HQ Bank of Montreal	indusHQ	
HQ Nortel	indusHQ	
Klondike Pk	indus	
Legislature	prov	
National Assembly	prov	
Nortel	indus	
N.R.C.	indus	
Parliament	fed	
Rockcliffe Airport	indus	
McMaster U.	univDM	
Stelco Inc	indusHQ	
Town Hall	municip	
U.Alberta	univDM	
U.B.C.	univDM	
U. Laval	univDM	
U. Manitoba	univDM	
U.N.B.	univD	
U. Ottawa	univDM	
U. Ottawa	univDM	
U. Toronto	univDM	
U. Waterloo	univD	
Waterloo Maple	indus	
Winnipeg Airport	indus	
YVR	indus	

Categories

(*Group*	*DistThresh*	*Categ*)
fed	15	Govt
prov	15	Govt
municip	1	Govt
univDM	1	Univ
univD	1	Univ
univ	1	Univ
indusHQ	5	Ind
indus	1	Ind

Figure 40: One Relation from a GIS, and Two Giving a Concept Hierarchy

We are also given socio-economic data about neighbourhoods in the relation *Classific*, which classifies them into `rich` or `poor`. We will seek spatial proximities that influence this classification: to industry, to universities, and to each of the three levels of government.

Classific	
(*Neighbourhood*	*RichPoor*)
Assiniboia	poor
Crystal Beach	rich
Dundas	rich
Milton-Park	poor
Oromocto	poor
Ottawa South	rich
Richmond	rich
Rockcliffe Park	rich
Rossdale	poor
Sandy Hill	rich
Ste-Foy	rich
Verdun	poor
Waterloo	rich
Yorkville	rich

We start the processing by defining "close to" (relative to the type of feature) as $Dist \leq DistThresh$, and selecting those features that are close to neighbourhoods in the combination of *Distance*, *Grouping*, and *Categories*.

Close <− [*Neighbourhood, Feature, Group, Categ*]
    where $Dist \leq DistThresh$ **in** *Distance* **join** *Grouping* **join** *Categories*;

This produces 31 tuples.

Close (*Neighbourhood*	*Feature*	*Group*	*Categ*)
Oromocto	N.B.Legislature	prov	Govt
Oromocto	Town Hall	municip	Govt
Oromocto	CFB Gagetown	indus	Ind
Rossdale	Alberta Legis.	prov	Govt
Crystal Beach	Parliament	fed	Govt
Crystal Beach	Nortel	indusHQ	Ind
Richmond	Town Hall	municip	Govt
Richmond	YVR	indus	Ind
Dundas	Town Hall	municip	Govt
Dundas	McMaster U.	univDM	Univ
Dundas	Stelco Inc	indusHQ	Ind
Milton-Park	Town Hall	municip	Govt
Milton-Park	McGill U.	univDM	Univ
Sandy Hill	Parliament	fed	Govt
Sandy Hill	U. Ottawa	univDM	Univ
Assiniboia	Manitoba Legis	prov	Govt
Assiniboia	Winnipeg Airport	indus	Ind
Yorkville	Ontario Legis	prov	Govt
Yorkville	U.Toronto	univDM	Univ
Yorkville	HQ Bank of Montreal	indusHQ	Ind
Waterloo	Town Hall	municip	Govt
Waterloo	U.Waterloo	univD	Univ
Waterloo	Waterloo Maple	indus	Ind
Ste-Foy	National Assem	prov	Govt
Ste-Foy	U. Laval	univDM	Univ
Rockcliffe Park	Parliament	fed	Govt
Ottawa South	Parliament	fed	Govt
Ottawa South	Carleton U.	univD	Univ
Ottawa South	Expt Farm	indus	Ind
Verdun	Town Hall	municip	Govt
Verdun	Nortel	indus	Ind

But we have not said anything about how *Dist* is measured, and so about how precise the meaning of "close to" is. Since the data in the map could be very extensive, it makes sense to calculate distance by a succession of approximations. The first approximation is to consider the crowfly distance between the boundaries of rectangles (MBRs) containing the features. We suppose that all the above calculations were done using this approximation.

The second approximation can be applied to those features that were found to be close by the first approximation. This is to consider the crowfly distance between the boundaries of the polygons that actually define the features. This more expensive calculation need not be applied to any features that were found too far apart by the first approximation, since if the MBR distance is too great, the polygon distance also will certainly be too great. We will suppose that using the second approximation changes the values of *Dist* in figure40, but happens not to put any distances over the respective thresholds.

The third calculation is the most expensive, and so is performed only for those features found close by the second approximation. This is the road distance between polygon boundaries. It requires geospatial calculations on a map database that contains road information as well as everything else. So we go back for a third iteration on the above data, and we suppose that still nothing changes significantly except for the distance between `Verdun` and `Nortel` on Nun's Island just across the channel: the road distance is 3 km, whereas the first and second approximation distances were about 1 km. This is now over the threshold specified for distances from ordinary industry, and so the `Verdun`–`Nortel` tuple vanishes from *Close* on this third pass.

We now have 30 tuples in *Close*: those shown, except the last.

We would like to turn this result into a classification relation with one tuple per neighbourhood, with attributes *Govt, Univ*, and *Ind*, and with the *RichPoor* classification attribute. For *Govt*, we want to record which of the three levels of government the neighbourhood is near (and if more than one, we wish to keep only the highest-ranking in the order `fed`, `prov`, and `municip`). For *Univ* and *Ind*, we will not care which type of university or industry is involved, but will simply record `yes` if the feature is close and `no` if it is not.

So we need a ranking of levels of government:

let *GovRank* **be if** *Categ*=`"Govt"` **and** *Group*=`"fed"` **then** 3 **else**
if *Categ*=`"Govt"` **and** *Group*=`"prov"` **then** 2 **else** 1;

We will select tuples for which *GovRank* has the maximum value by *Neighbourhood* and *Category*.

We need a single attribute, *Value*, which will hold the level of government if *Categ* is `Govt`, which will be `yes` if an industry is close and `no` if not, or which will be `yes` if a university is close and `no` if not. This requires us to construct a set of `no` values which we will use to fill the gaps:

let *ValNo* **be** `"no"`;
CategNo <– ([*Neighbourhood*] **in** *Distance*) **join** [*Categ, ValNo*] **in** *Categories*;

(For instance, `Oromocto` has no `Univ` tuple in *Close*, above: this obliges us to supply a tuple for `Univ` which indicates that no university is close to Oromocto.)

Finally, we need a definition of *Value* which uses these `no`s where necessary, and takes `yes`s and levels of government from *Close*. This exploits the null values that result from unmatched tuples in the outer join, **union**:

let *Value* **be if** *Feature*=**null then** *ValNo* **else**
if *Categ*=`"Govt"` **then** *Group* **else** `"yes"`
CloseGenl <– ([*Neighbourhood, Categ, Value*]
where *GovRank*=**equiv max of** *GovRank* **by** *Neighbourhood, Categ*
in
(*Close* **union** *CategNo*)) **join** *Classific*;

Here is the result. (The virtual attributes, *Govt, Univ*, and *Ind*, are defined immediately below.)

CloseGenl

(*Neighbourhoods*	*RichPoor*	*Categ*	*Value*)	*Govt*	*Univ*	*Ind*
Oromocto	poor	Govt	prov	prov	no	yes
Oromocto	poor	Ind	yes	prov	no	yes
Oromocto	poor	Univ	no	prov	no	yes
Rossdale	poor	Govt	prov	prov	no	no
Rossdale	poor	Ind	no	prov	no	no
Rossdale	poor	Univ	no	prov	no	no
Crystal Beach	rich	Govt	fed	fed	no	yes
Crystal Beach	rich	Ind	yes	fed	no	yes
Crystal Beach	rich	Univ	no	fed	no	yes
Richmond	rich	Govt	municip	municip	no	yes
Richmond	rich	Ind	yes	municip	no	yes
Richmond	rich	Univ	no	municip	no	yes
Dundas	rich	Govt	municip	municip	yes	yes
Dundas	rich	Univ	yes	municip	yes	yes
Dundas	rich	Ind	yes	municip	yes	yes
Milton-Park	poor	Govt	municip	municip	yes	no
Milton-Park	poor	Univ	yes	municip	yes	no
Milton-Park	poor	Ind	no	municip	yes	no
Sandy Hill	rich	Govt	fed	fed	yes	no
Sandy Hill	rich	Univ	yes	fed	yes	no
Sandy Hill	rich	Ind	no	fed	yes	no
Assiniboia	poor	Govt	prov	prov	no	yes
Assiniboia	poor	Ind	yes	prov	no	yes
Assiniboia	poor	Univ	no	prov	no	yes
Yorkville	rich	Govt	prov	prov	yes	yes
Yorkville	rich	Univ	yes	prov	yes	yes
Yorkville	rich	Ind	yes	prov	yes	yes
Waterloo	rich	Govt	municip	municip	yes	yes
Waterloo	rich	Univ	yes	municip	yes	yes
Waterloo	rich	Ind	yes	municip	yes	yes
Ste-Foy	rich	Govt	prov	prov	yes	no
Ste-Foy	rich	Univ	yes	prov	yes	no
Ste-Foy	rich	Ind	no	prov	yes	no
Rockcliffe Park	rich	Govt	fed	fed	no	no
Rockcliffe Park	rich	Univ	no	fed	no	no
Rockcliffe Park	rich	Ind	no	fed	no	no
Ottawa South	rich	Govt	fed	fed	yes	yes
Ottawa South	rich	Univ	yes	fed	yes	yes
Ottawa South	rich	Ind	yes	fed	yes	yes
Verdun	poor	Govt	municip	municip	no	no
Verdun	poor	Univ	no	municip	no	no
Verdun	poor	Ind	no	municip	no	no

The domain algebra for the virtual attributes, above, is

let *Govt* **be equiv max of if** *Categ*=`"Govt"` **then** *Value*
else `""` **by** *Neighbourhood*;
let *Univ* **be equiv max of if** *Categ*=`"Univ"` **then** *Value*
else `""` **by** *Neighbourhood*;

let*Ind* **be equiv max of if** *Categ*=`"Ind"` **then** *Value*
else `""` **by** *Neighbourhood*;

followed by

Training <− [*Govt, Univ, Ind, RichPoor*] **in** *CloseGenl*;

and we see that *Training* is isomorphic to *Training* in section 8.1 if we leave out the attribute *Temperature* and link *Govt* with *Outlook*, *Univ* with *Humdity*, *Ind* with *Windy*, and, of course, *Class* with *RichPoor*.

The decision tree (or any of the other classification results) is the same as before, and we can conclude (from this evidently manipulated data) that

- if the neighbourhood is near the federal government it is rich;
- if the neighbourhood is near a provincial government then if it is near a university it is rich otherwise it is poor;
- if the neighbourhood is near a municipal government then if it is near industry it is rich otherwise it is poor.

9 Spatial Collaboration on the Internet

There are many paradigms for collaborative work, most of which go back to the origins of humanity: conversation, which has a take-turns protocol; meetings, where turn-taking is moderated by a chairman; teams, where interaction is structured by a shared construct or goal; negotiation; arbitration; etc. The Internet, and related computer networking, have introduced remote variants, such as multi-user domains ("MUDs"), with a chat protocol in a shared environment, and simple constructs, such as meeting schedulers (which those familiar with spatio-temporal concepts will recognise as an *or*-overlay of impossible times). Before the Internet, of course, the postal system and the telephone provided remote take-turns conversations; and printing made available replicated shared resources in the form of books and lighter reading.

The two central elements of computer-supported cooperative work (CSCW) are shared resources and the protocols for sharing them. We consider the protocols to be social in nature, rather than technological, and that, given a technology supporting sufficiently transparent sharing, the protocols developed over the millennia will be used, as appropriate for the situation, by the participants. We are primarily interested in technologies which are as invisible as the air between two talkers and as straightforward as a shared blackboard or a common map.

We look at two forms of sharing. In the first, there is instant exchange among participants. This would apply to a brainstorming session, for example. We call this the *discussion* mode of sharing. In the second, individuals agree to work on their own on some aspect of a shared project, and later negotiate the integration of their work with that of others. This is the *checkout-checkin* form of sharing. It is suitable whenever more focussed work is needed on the common goal.

It is important that a participant in either of these modes each have access to heir own preferred view of the shared resource. For example, some may want WYSIWYG for a text document while others prefer marked-up text. For a design project, some may want a perspective view, and some may want a plan view. For geospatial collaboration, some may prefer an orthographic projection and others transverse mercator. Indeed, any single participant may simultaneously need multiple views.

Thus, the sharing must take place at the level of a common abstraction, rather than at the level of the display representation. This leads us to consider at least two layers on top of the transport layer of the ISO/OSI and of the TCP/IP models for Internet communication. The lower layer will be abstract, and at the level of the relations basically representing the shared resources and constructs. The upper layer will offer specific display representations.

We can distinguish two classes of user corresponding to these two layers. The *programmer-user* deals with the relational abstraction, which is the level we have discussed exclusively so far in this tutorial. The *end-user* interacts with the displays and is the ultimate participant in the collaborations. The programmer-user's interest is in the relations and the high-level relational operators that provide the displays with something to show the end-user. The end-user, on the other hand, is interested in all the details of the data which the programmer-user considers only as abstract relations.

In fact, the issues of Internet protocols and the details of the sockets that link these application layers with the TCP/IP of the transport layer are beyond the scope of this tutorial. So, for simplicity, we interject another layer in between, and use electronic mail, which is familiar to everybody, as the application protocol on which we base the communication of the shared work at the relational level.

Thus, we assume email as the basis for this discussion, on top of which we build first the relational layer, then the display layer. This is not to claim that email should be used for this; a more direct protocol would undoubtedly be more effective. However, it gives us two simplifying advantages, in addition to familiarity on the part of the reader.

The first advantage is that we get an almost memoryless protocol. Apart from the updates to the shared relations, no state of interactions must be maintained. By making the individual participants independent of each other, this greatly simplifies the design of the collaborative software. The second advantage follows, namely that the whole approach is completely decentralized. There is no central resource or control, only the participants' copies, which are automatically kept up to date, according to whether the mode is *discussion* or *checkout-checkin*, to the extent that an automated system can keep them up to date. (Beyond this extent, the participants themselves are invited to decide the current state of the project, and provided with tools to help.)

The next two subsections discuss the relational and display layers, respectively. The relational layer must take care of the sharing, and so must support updates caused either by the display editors above it or by the emails, below it, from other users' relational layers. This involves locking in case simultaneous updates produce nonsense, and relaying the events of updates from the layer above or below to the layer below or above, respectively. This requires us to discuss event database techniques for programming and for concurrency.

The display layer supports editors which give different views of data in a single data structure. It is also concerned with sharing and with relaying update events, but in the context of a single user whose actions on different views we will not consider to be concurrent.

9.1 Concurrency and Events

The relational layer is an intermediary between the Internet communications and the displays. Both of these layers update relations, and these updates are the events that the relational layer relays through. The updates must be protected from certain kinds of concurrency, and this requires locking mechanisms.

Both concurrency mechanisms and event handlers are techniques for coordinating independent processes. Event techniques are used when one process, usually a program, is to respond to another, often a user interacting with the program. Concurrency techniques are

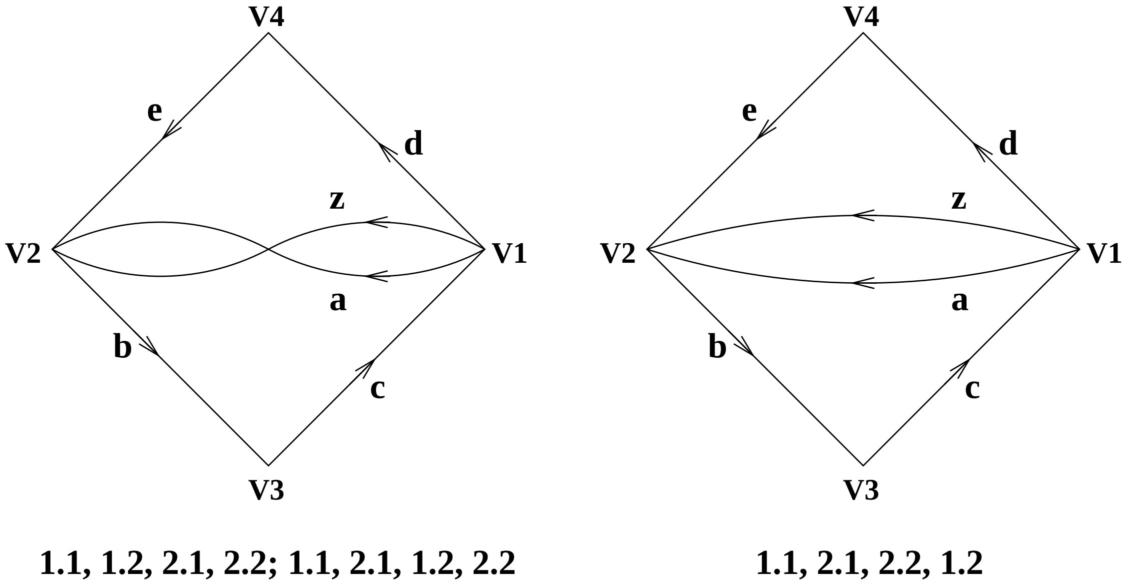


Figure 41: Interleaved *Splice* Operations

used to delay a process until some other process has accomplished a specified task. The approaches are quite different. The blocking mechanism that implements delay in concurrency control implicitly puts the invoking process to sleep and relinquishes its computing resources to other processes which may be sharing the same CPU. The event handler, on the other hand, is simply a procedure body, and introduces no new concepts of time or synchronization to the programming language that uses it.

9.1.1 Concurrency

A process sharing a resource with another may have to force the other to delay until it is finished, even in the most cooperative environment. Figure 41 shows the results of two users each adding an edge between vertices `V1` and `V2` of a shared diagram represented as a quad-edge data structure. The update operations involve splices:

User 1	*User 2*
1.1 *splice*(`a0, d0`)	2.1 *splice*(`z0, d0`)
1.2 *splice*(`a2, b0`)	2.2 *splice*(`z2, b0`)

The two steps required by each of these users could be executed in six different interleaved sequences, with four different outcomes. Figure 41 shows the two of these with step 1.1 executed before step 2.1. The other two outcomes have the positions of `a` and `z` reversed. Presumably the right-hand result would be preferred to the left. But quite likely any of the four should be deemed a conflict: probably both users had the same idea, to connect `V1` and `V2` by a single edge, and only one of them should have carried this out.

In this case, there is a conflict between the two users, and the first one off the mark should have delayed the second until heir intentions were clear. The concurrency control mechanism to impose such a delay is a *lock*, which prevents access to the shared resource until it is released by the process that set it in the first place. The code, as well as the splices, should contain the instructions to lock the shared edge ends.

User 1	*User 2*
lock(`d0, b0`)	*lock*(`d0, b0`)
1.1 *splice*(`a0, d0`)	2.1 *splice*(`z0, d0`)
1.2 *splice*(`a2, b0`)	2.2 *splice*(`z2, b0`)
unlock(`d0, b0`)	*unlock*(`d0, b0`)

Now, *User 1* finishes before *User 2* is permitted to start, or vice versa, and we presume that *User 2* (resp., *User 1*) is shown the result.

Because the users are sharing the diagram, we do not lock the whole data structure. We lock the relations at the finest *granularity* possible. Here, this is the two intended endpoints of the new edge. Strictly, two operations interfere with each other only if they affect the same vertex at the same position in the cycle of edges around that vertex. All other operations may run concurrently. But we see from the left side of figure 41 that we might get undesired results if we do not protect the whole edge to be inserted.

Locks become especially important if the two users are updating locally stored copies of the shared diagram, which are supposed to remain in agreement with each other. *User 1* might run *splice*(`a0, d0`) on his local copy, and *User 2* *splice*(`z0, d0`) on hers. Then the system may transmit these updates to the other users' copies, so the effect on *User 1*'s copy is 1.1, 2.1, while the effect on *User 2*'s is 2.1, 1.1, and the new edges, `a0` and `z0`, leave the vertex in different orders at different sites.

The locking procedure must be to request a lock on all sites, then not proceed until all locks are acknowleged as successful. If the lock request must wait on some site(s), this means that another user has been granted the lock there and is about to be granted it everywhere: the request is denied and retried.

A waiting lock request (the lock having already been granted to another user) could be recorded at each site, and indicated on the display, perhaps by a flashing edge and vertex. This would serve to alert users of conflicting operations, so that one could voluntarily back down: otherwise, there is a problem of *deadlock*, two users waiting for each other, as when *User 1* has a lock on `d0` and is waiting for the lock on `b0` held by *User 2*, who is waiting for the lock on `d0`. Since the process is cooperative, we are less concerned about undetected deadlocks, or even about the inconsistencies that "two-phase locking" techiques prevent, than we would be in a fully automatic concurrent system.

The syntax which implements delays in the relational algebra is an almost invisible change from the T-selector

[<projection-list>] **where** <condition> **in** R

to

[<projection-list>] **when** <condition> **in** R

The interpretation of this is close to Linda's [10] blocking mechanism: if the T-selector (with **where**) would have returned an empty relation, the blocking T-selector (with **when**) blocks until R is updated, by some other process, in such a way that the result is no longer empty. (Unlike Linda, the blocking T-selector is fully deterministic, returning all relevant tuples in the case that it does not block.) This can be used to construct semaphores, which are the usual mechanism to implement locking.

Even greater confusion can arise if both users are trying to add the same edge, say, `a`, to the same or different places of the shared diagram. This can be avoided by having a system-wide generator for the internal identifiers of edges, which gives everyone a unique new edge whenever they create one. We can suppose from now on that two users do not share the edges they are adding to a shared diagram. (User-perceived names, such as `a`, could be extra data, not used to identify edges to the system, and so shareable.)

The fact that the system is distributed as well as concurrent also affects us when we want to *commit* an operation. This issue arises in the *checkout-checkin* mode of sharing, when a user, having completed a modification, wishes to have it generally accepted. The "two-phase commit" methodology for distributed transactions is useful here. In the voting phase, (actually a *veto* phase) everyone considers whether the modification is accepted, and in the commit phase, if it is accepted, the copies are updated. In this cooperative system, it is the users who vote (veto) and decide on the outcome, rather than some algorithm, as would be the case in an ordinary distributed database.

(Two-phase commit is inadequate in the presence of host or communication failures. For deeper treatment, an elegant and thorough treatment of transactions, concurrency, and distributed databases can be found in the early book by Bernstein, Hadzilacos, and Goodman [8].)

We can provide a differencing tool to show what has changed. We suppose that *checkout* makes a copy of the data at the site where the updates are made. It also notifies all sites that the data structure has been checked out. The first task of *checkin* is to run the difference operator on the old and changed copies. The result can then be broadcast as *splice*() arguments to all other sites. If nobody else has checked out the data, these arguments can be used to update all sites directly, using the above locking strategies. If somebody has also checked it out, negotiations must take place, possibly waiting for *checkin* to be initiated on all checked out copies.

The difference operator for quad-edge diagrams is a simple **diff** on relations (see section 5.3.1). Just as the union operator is not the pure set union of SQL, the difference join is not simple set difference. Suppose that *OldDiagram* and *CheckedOut* are the former and updated quad-edge relations, respectively.

OldDiagram($e1$	$d1$	$e2$	$d2$)
c	2	d	0
d	0	c	2
b	0	e	2
e	2	b	0
b	2	c	0
c	0	b	2
d	2	e	0
e	0	d	2
b	3	c	3
c	3	d	3
d	3	e	3
e	3	b	3
e	1	d	1
d	1	c	1
c	1	b	1
b	1	e	1

CheckedOut($e1$	$d1$	$e2$	$d2$)
c	2	d	0
d	0	a	0
a	0	c	2
a	2	e	2
e	2	b	0
b	0	a	2
b	2	c	0
c	0	b	2
d	2	e	0
e	0	d	2
b	3	c	3
c	3	a	3
a	3	b	3
d	3	e	3
e	3	a	1
a	1	d	3
e	1	d	1
d	1	c	1
c	1	b	1
b	1	e	1

Then we apply the following

NewVertices <− **where** $d1$ **mod** $2 = 0$ **in** *CheckedOut*;
OldVertices <− $[e2, d2]$ **where** $d1$ **mod** $2 = 0$ **in** *OldDiagram*;

difference <− *NewVertices* **diff** *OldVertices*;

which gives

difference(*e*1	*d*1	*e*2	*d*2)
d	0	a	0
b	0	a	2

and this is exactly the operands for the two *splice*() operations that changed *OldDiagram* to the new version of *CheckedOut*.

If another user has also checked out the relation, two such **diff** operations, one going each way, will identify the different updates that were made to the two checked out relations. This should help in the discussion about how the two versions are to be integrated again.

9.1.2 Distributed Processing

To transmit update events to other sites, we have decided to rely on email for the purposes of this discussion. For this, we need to introduce the notions of *source relation* and *sink relation*. These are pseudo relations, which can appear, respectively, to the right or left of an assignment arrow, and do such things as reading or printing relations, and, for our present interest, receiving or sending email.

Thus,

.print <− <*relation*>;

will print *relation* in some preformatted way to the current output device.

.mailout <− <*relation*>;

will send the relation as email, provided *relation* contains a field *To:*, and a field *Subject:*. Then each tuple will be mailed to its respective address, with the subject header.

These are *sink* pseudo-relations. Examples of *source* pseudo-relations are

<*relation*> <− *.read*;

which reads a relational expression from the current input device and evaluates it before assigning it to *relation*:

update <*relation*> **add when** <*condition*> **in** *.mailin*;

waits for the specified email to be received and adds it to *relation* (using an **update..add** statement, which can generate events for the display editors).

A simple example shows these mail constructs in the context of distributed processing, which avoids the complication of display interfaces (discussed in section 9.2) and human collaboration. We consider a distributed program to perform matrix multiplication, with a parent site coordinating and integrating results from child sites which each multiply one row of one matrix, A, by the entire second matrix, B. These are represented, respectively, by the relations $A(i, j, a)$ and $B(j, k, b)$, each with attributes for row, column, and value.

First, the matrix multiplication code in general needs one equivalence reduction from the domain algebra and one natural join from the relational algebra.

let c **be equiv** + **of** $a \times b$ **by** i, k;
C <−$[i, k, c]$ **in** (A **join** B);

This implementation can be used either to multiply the two matrices completely at a single site, or to multiply a row of A by all of B at a child site, depending on how many rows of A are represented by the relation $A(i, j, a)$.

We will use a simple convention for *.mailout* which requires only two special attributes, *To:* and *Subject:*. For *.mailin*, we can assume that any of the usual mail headers generated by the underlying email clients (such as *From:, Date:*, etc.) are available as attributes, but we will not use them here. With this convention, here is the code for the parent site.

let *To:* **be** "child" **cat** i;
let *Subject:* **be** "A";
.mailout <– [*To:*, *Subject:*, i, j, a] **in** A;
let *Subject:* **be** "B";
.mailout <– ([*To:*, *Subject:*] **in** A) **join** B;
for *count* <– 1 **to** [**red max of** i] **in** A
    update C **add when** *Subject:*="C" **in** *.mailin*;

The first *.mailout* has a different *To:* value for each child site, according to the value of i, and so mails a row of the matrix to each child. The second *.mailout* sends a copy of all of B to each child, one for each value of i in A. The *.mailin* loop iterates for each i in A to receive the responses from the child sites, and puts the union of these results (in any order) into C.

Each child site receives its mail, does the multiplication, and mails the result to the parent.

A <– **when** *Subject:*="A" **in** *.mailin*;
B <– **when** *Subject:*="B" **in** *.mailin*;
let c **be equiv** + **of** $a \times b$ **by** i, k;
C <–[i, k, c] **in** (A **join** B);
let *To:* **be** "parent";
let *Subject:* **be** "C";
.mailout <– [*To:*, *Subject:*, i, k, c] **in** C;

(It is also useful to have a mechanism for creating alias email addresses at the different sites. In this way, `parent` could be the same site as `child1`, say.)

This simple example provides the groundwork for more complicated cooperation involving humans, displays, and events. It also shows that distributed processing is a special case of distributed cooperative work.

9.1.3 Events

We have covered the basics of the concurrency control, notably locking, that must be used to protect the shared work from confusion. We have also presented the difference operations, which can pinpoint differences between map versions. We must now consider how to ensure that displays are refreshed, both locally and remotely, whenever an update is made anywhere. This applies both to the *discussion* mode and, at checkin time, to the *checkout-checkin* mode of collaboration.

The technique is to use events. We ensure that the display editors (to be discussed in the next section) are built to respond to events, whether from the user's mouse- and keyboard-actions, or from remotely-generated updates to the underlying data structure.

In contrast, to some extent, with delay synchronization, events are extremely simple concepts already familiar to any programmer. They are not, however, what the word, "event", implies in English. *An event handler is a procedure. An event is a system-generated procedure call.* When a display editor responds to a mouse click, it is executing a procedure, which was called by a system command in response to the end-user's action on the mouse. We provide similar procedure calls when a relation is updated, and the display editor must respond to them in the same way.

Furthermore, this mechanism makes it possible for two or more display editors, running concurrently on the same data structure, possibly offering different views of the same data, to affect each other's displays as a result of an update in one or another. The active editor, in which the user makes a change, updates the underlying relations; this in turn causes an event which results in procedure calls in the other editors to refresh their displays reflecting the change.

The syntax for an event handler is identical to that for a procedure, except that the name of the procedure must reflect the calling event. For relation updates, the general forms of the name are

[**pre**|**post**] **add** <*relation*>
[**pre**|**post**] **delete** <*relation*>
[**pre**|**post**] **change** <*relation*>:<*field*>

Postscript. In characterizing an event as only a system-generated procedure call, we have cut a Gordian knot in the research literature on events, and have also deprived ourselves of being able to write syntax for combinations of "events" (such as more than one bank withdrawal per hour, or withdrawals *or* deposits on a certain day). But the definition we give allows us to build a log of such "events", and use ordinary database querying or further event handlers to detect these compound events. The great variety of possibilities one might want to check for makes special syntax counterproductive in any case, given the general query capabilities already available.

9.2 Update Editors

We turn from the relational layer to the display layer. The user of the relational layer is the database programmer. The user of the display layer is the end user, whose interest is in the spatial data itself rather than the relational abstractions. The bridge between the two is an extension of the **update** statement in the relational layer.

Suppose we have built a display editor which emulates the interface of a G.I.S. and have called it **GISedit**. The relation layer interface might be

update *Map* **GISedit**;

where *Map* is a relation holding the map data.

The previous **update** commands invoked algorithms to add, delete, or change the relation. The **update .. edit** command invokes the named editor which opens window(s) to the end user to view and manipulate the data. When the end user closes the session, the **update** statement completes, with *Map* suitably changed. To the relational programmer, all the **update** commands behave in the same way. They execute, changing the relation. Only the duration of the **update .. edit** variant is likely to be longer, and unpredictable.

An editor used in **update .. edit** mode should directly update the relation, causing update events which can be handled by procedures in the relational layer. It should also be able to respond to update events that may arise from externally caused updates to the same relation, running concurrently. The most basic response is to refresh the display to reflect any external updates to the end user. In this way, update instructions that have arrived by email from remote participants will be displayed by the editor as soon as they are accomplished. Further, the local program could be running more than one display editor concurrently, to provide the end user with different views, and changes by any of these editors are reflected by the others.

(A purely functional form of editor can be invoked as a relational *expression*, rather than as an update *statement*. To the relational layer, it seems to be a unary operator, such as a T-selector. It does not change the operand relation, but returns an updated copy as the value of the relational expression. It cannot cause or respond to concurrent updates because it does not update. It is a *functional editor*, as opposed to the above *update editor*.)

Postscript. Cooperative computing on the Internet goes beyond the standard, client-server, paradigm. A client is a program which initiates an interaction, while a server is software which can respond to such an initiative. By analogy with the telephone system, the client dials and the server answers. An example of this is the world-wide web browser and website interaction. The server manages the pages of the website and responds to browsers requesting connection and subsequent services. The browser is the client and offers its users the services of document display, applet execution, and so on. In electronic mail, there is usually a server to distribute mail, with a corresponding "send" client, and a server to receive and sometimes manage one user's mail, with a corresponding "receive" client.

In our model of cooperative work, we rise above these considerations to a more symmetrical, peer-to-peer, perspective. An update at any site is broadcast to and reflected at all other sites. Beneath it all, of course, are the email client-server pairs, but the user, and to an important extent the programmer, need not be aware of this machinery behind the scenes.

9.3 Example: Distributed Workflow in Collaborative Mapmaking

To illustrate these ideas, we consider two simplified issues in mapmaking. The first is the process of inspection of maps by the client (government, for instance) after (or as) they are drawn by the contractor. The second is the problem of map registration when the contractor has divided the geographical region among several subcontractors, and the resulting regional maps must be concatenated.

9.3.1 Online Inspection

The traditional interaction between customer and contractor, which we seek to improve upon, was for the contractor to prepare the maps ordered by the client, then courier them to be inspected. The customer's inspector would note the incorrect street names, missing streets, buildings, hydrants, etc., and courier the drafts back for correction. This slow process could go on for several iterations.

The fanciest improvement to this would be to link the contractor's map editor to the inspector's in the way described in section 9.2, so that the inspector could detect in real time whenever the contractor makes a mistake. This is not useful because the inspector has other things to do than watch maps being drawn line by line, and because time zone differences in a large country make it further impracticable. So we look at an intermediate solution involving email.

In this approach, the contractor periodically mails the three components of the map, *QuadEdge, VertFace*, and *Geom* (see section 4.2), to the inspector, and awaits the comments. Since this can be done a little more frequently than the contractor would have sent a courier in the past, it is advisable to label each transmission, say with today's date, so that the contractor knows which draft the comments refer to when they come back. Here is the simplest code, at the contractor site.

```
let To: be "inspector@maps.gov";
let Subject: be "QuadEdge:" cat today;
.mailout <- [To:, Subject:, edge1, dir1, edge2, dir2] in QuadEdge;
let Subject: be "VertFace:" cat today;
.mailout <- [To:, Subject:, edge, org, dest, left, right] in VertFace;
let Subject: be "Geom:" cat today;
.mailout <- [To:, Subject:, vf, x, y] in Geom;
Comments <+ when From: = "inspector@maps.gov" and
        Subject = "Comments" cat today in .mailin;
```

At the inspector's end of the line, the three relations arrive by email

let *date* **be** substr(*Subject:*, ":");
QE <− [*From:, date, edge1, dir1, edge2, dir2*]
when substr(*Subject:*, 0, ":") = "QuadEdge" **in** *.mailin*;
VF <− [*From:, date, edge, org, dest, left, right*]
when substr(*Subject:*, 0, ":") = "VertFace" **in** *.mailin*;
G <− [*From:, date, vf, x, y*]
when substr(*Subject:*, 0, ":") = "Geom" **in** *.mailin*;

(the substring function has been streamlined for brevity). Using the data abstraction technique of section 10, to follow, we now suppose that the three relations have been combined into the nested relation, *Map*. The inspector can examine this using a functional editor (since hey does not need to alter the map) to generate comments, which are mailed back to the contractor.

Comments <− **GISedit** *Map*;
.mailout <− *Comments*;

9.3.2 Map Registration

A large mapping project may be delegated by the contractor to several subcontractors, with the resulting problem of combining the results into a single map at the end.

In an ideal world, this would be a simple matter of uniting the individual maps, and the *Map* nested relation gives us an easy way of doing this. The main contractor now plays the role of the inspector in the above discussion, and receives a draft map from each subcontractor. *Map* now has several tuples (whereas in section 9.3.1, above, it had only one), consisting of the nested components *QuadEdge, VertFace*, and *Geom*. The simple-minded union of these is achieved by three lines of domain algebra.

let *QuadEdgeU* **be red union of** *QuadEdge*;
let *VertFaceU* **be red union of** *VertFace*;
let *GeomU* **be red union of** *Geom*;

Unfortunately, the world is not ideal and there will be discrepancies among the map sheets provided by the separate subcontractors. This is the registration problem, which requires the contractor to check that features in each map align with the same features in adjacent maps, before uniting the maps. The subcontractors' maps all overlap at the edges, to permit this check. The inspector compares the coordinates, from the two maps, of the vertices of a shared feature, and the differences for any three vertices tell hem whether that part of one map has been scaled, rotated, or translated relative to the other.

We illustrate this with the feature of figure 5, which has two triangles. Figure 42 shows this feature as it might be represented at the adjacent edges of two different maps. The discrepancies are greatly exaggerated for visual clarity. As it happens, in one of the maps relative to the other, the upper triangle has been reduced by a scale factor of 1/2, then rotated 53.13 degrees, then translated vertically by 10% of its original height. The lower triangle has been treated almost alike, except that it was deformed by an initial scaling of 1/2 in the x direction and 2/3 in the y direction, before rotating and translating.

How do we discover what these (presumed accidental) transformations were, from the corresponding coordinates alone? Let's start supposing we know the transformations, calculating their effect on the coordinates, and then seeing if we can reverse the calculation.

A scaling of a point (x, y) by a factor s_x in the x direction and s_y in the y direction, followed by a rotation through an angle whose cosine and sine are c and s, respectively, followed by a translation t_x units in the original x direction and t_y units in the original y

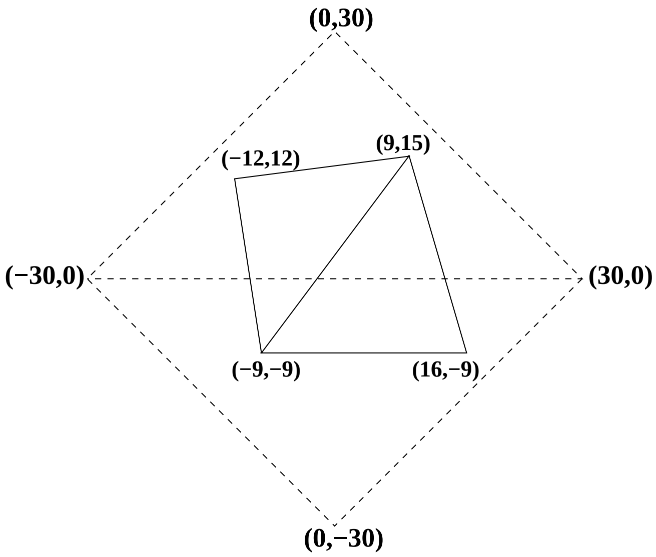


Figure 42: Two maps, one feature (exaggerated)

direction, is given by[3]

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \end{pmatrix} + \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

This gives the new coordinates, (x', y'), in terms of the old. Because we are going to have to invert this calculation, and find s_x, s_y, c, s, t_x, and t_y given x, y, x', and y' for a set of points (three points, to be precise, because there are six unknowns), it is more convenient to express the transformation solely in terms of matrix multiplication. We can do this by imagining a z-axis at right angles to the maps, and supposing that the map and all features on it are on the plane $z = 1$. Then translation becomes a shear transformation given by the leftmost matrix below.

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Note that the z-axis is not affected by the transformations, and that both the input and the result vectors have $z = 1$.

In our example, the transformation of the three points of the lower triangle is thus

$$\begin{pmatrix} 9 & 16 & -9 \\ 15 & -9 & -9 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3/5 & -4/5 & 0 \\ 4/5 & 3/5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1/2 & 0 & 0 \\ 0 & 2/3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 30 & 0 & -30 \\ 0 & -30 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

[3] This is a special case of an *affine* transformation, used for instance in fractal compression of images [6]. In the general affine transformation, there are two cs and two ss, and so two angles of rotation, one associated with the original x-axis and one with the y-axis. The extension of our discussion to the full affine transformation is minor, and we consider only single angles.

If we did not know the transformation, this would be

$$\begin{pmatrix} 9 & 16 & -9 \\ 15 & -9 & -9 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 30 & 0 & -30 \\ 0 & -30 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

We can rewrite this with a single matrix of unknowns

$$\begin{pmatrix} 9 & 16 & -9 \\ 15 & -9 & -9 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} cs_x & -ss_y & t_x \\ ss_x & cs_y & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 30 & 0 & -30 \\ 0 & -30 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

We find the unknowns by inverting the matrix of known coordinates that it is multiplied by

$$\frac{1}{1800} \begin{pmatrix} 9 & 16 & -9 \\ 15 & -9 & -9 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 30 & 30 & 900 \\ 0 & -60 & 0 \\ -30 & 30 & 900 \end{pmatrix} = \begin{pmatrix} 3/10 & -8/15 & 0 \\ 4/10 & 4/10 & 3 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} cs_x & -ss_y & t_x \\ ss_x & cs_y & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

We can represent the product matrix, which gives the result of the unknown scaling, rotation, and translation, as a relation, and we can use the domain algebra to solve for the separate unknowns. (The angle is now in radians.)

SclRotTrans(i	j	*val*)	t_x	t_y	s_x	s_y	c	s	θ
1	1	0.3	0	3	0.5	$0.66\overline{6}$	0.6	0.8	0.9273
1	2	0.4	0	3	0.5	$0.66\overline{6}$	0.6	0.8	0.9273
2	1	$-.53\overline{3}$	0	3	0.5	$0.66\overline{7}$	0.6	0.8	0.9273
2	2	0.4	0	3	0.5	$0.66\overline{6}$	0.6	0.8	0.9273
3	2	3	0	3	0.5	$0.66\overline{6}$	0.6	0.8	0.9273
3	3	1	0	3	0.5	$0.66\overline{6}$	0.6	0.8	0.9273

let t_x **be red max of if** $i = 1$ & $j = 3$ **then** *val* **else** 0;
let t_y **be red max of if** $i = 2$ & $j = 3$ **then** *val* **else** 0;
let s_x **be** sqrt(
  (**red max of if** $i = 1$ & $j = 1$ **then** *val*×*val* **else** 0) +
  (**red max of if** $i = 2$ & $j = 1$ **then** *val*×*val* **else** 0));
let s_y **be** sqrt(
  (**red max of if** $i = 1$ & $j = 2$ **then** *val*×*val* **else** 0) +
  (**red max of if** $i = 2$ & $j = 2$ **then** *val*×*val* **else** 0));
let c **be if** $s_x > 0$
  then sign(*val*)×(**red max of if** $i = 1$ & $j = 1$ **then** abs(*val*) **else** 0)/s_x
  else sign(*val*)×(**red max of if** $i = 2$ & $j = 2$ **then** abs(*val*) **else** 0)/s_y;
let s **be if** $s_x > 0$
  then sign(*val*)×(**red max of if** $i = 2$ & $j = 1$ **then** abs(*val*) **else** 0)/s_x
  else $-$sign(*val*)×(**red max of if** $i = 2$ & $j = 2$ **then** abs(*val*) **else** 0)/s_y;
let θ **be** arctan(c, s);

The inversion and multiplication of the three-by-three matrices also involves algebra which can be expressed by domain algebra. It is long but obvious, and we do not explicitly write it out.

Since the distortion of the maps may vary from place to place, we must repeat this solution for every triangle: the reductions, above, become equivalence reductions. The triangles we choose should be Delaunay triangles, since these minimize the linear dimensions of the region affected by each distortion. In general, the Delaunay triangles of one distorted map will not be Delaunay triangles of the second distorted map, but in practice the distortions will be small and the Delaunay triangulations should be the same. In any case, one of the two maps should be triangulated, and the same triangles used for the corresponding vertices of the other map.

The result of repeating the above solution for both triangles of the example would be

Diff(*triangle*	t_x	t_y	s_x	s_y	θ)
F1	0	3	0.5	$0.66\overline{6}$	0.9273
F2	0	3	0.5	0.5	0.9273

Note that these calculations give only the distortions of one map relative to the other. Since both maps may have been distorted, the results are a clue only to the differences in distortion. Two equally distorted maps will ring no bells.

Once manual intervention, alerted by calculations such as these, have brought the two maps into agreement, they may be united by the **union** operations at the beginning of this section. Similarly, all the component maps, once made to agree across their overlapping edges, can be united into the single map the contractor offered to provide.

The coding offered in this section only touches the more general problems underlying map registration, known as map conflation [51]. These problems include automated matching of features common to the two maps to discover which parts of the maps are to be registered with each other. Polygon skeleton techniques (see section 6.4) can be used.

10 Programming Language Techniques

It is commonly claimed that the relational formulation of data on secondary storage is limited in its power to capture complex structures, and so must be thrown away in favour of other paradigms. We have seen that "complex objects" are well supported by nested relations (which are in turn built from the classical, flat, relations) and that "multidimensional data" is relational and readily processed with the domain and relational algebras.

Yet we have in this tutorial hardly touched upon the most ordinary of programming language constructs, such as scoping, type systems, or procedural and data abstraction. We raised the need for the latter when we first proposed a spatial data structure of some three relations together with its basic operation of splicing. We now discuss some of these constructs.

To get to data abstraction, we must first consider procedural abstraction. Having data abstraction leads us to the notions of state, instantiation, and classes of instances. These are the ideas behind "object-orientation", which is a constellation of programming-language concepts.

Since they are not database concepts, they do not conflict with relations. There is no more problem about an object-oriented relational programming language than there is about an object-oriented list programming language or an object-oriented numerical programming language. In fact, the high level of programming offered by relations, which abstract over tuples and over looping, allows us to raise our sights from "objects" to their higher-level abstraction, *classes*. Just as a relational programming language has no concept of a tuple, it need have no concept of an individual object.

This is fortunate, because the term, "object", itself is a misnomer and becoming jaded. It misleads many into thinking that they are programming with chairs or balls or other "objects", and that there is something particularly "objective" and real about objects. We will use the term *instance*, but we will program with classes.

10.1 Procedural Abstraction

Object orientation is a programming technique for limiting the side effects of assignment and update operations. It achieves this by *encapsulating* mutable data in a package defined by its data structure and the operations devised for that data structure. The operations are represented by the much older programing language device, the *procedure*. This is an abstraction of a piece of program code which, usually, allows some parts of the code (usually variables) to be parametrized. Thus, it is called *parametric abstraction*. We call it *procedural abstraction* for familiarity.

A special form of procedure is the *function*. This is a procedure without *side effects*: every time it is invoked with a fixed set of values for its parameters, it gives the same result. That is, it has no memory, no internal data which gets changed by the invocation, no *state*. It is also, usually, a procedure in which one parameter is the output and the remaining parameters provide input only; it may be written so that the output is not named explicitly by a parameter, but is returned as a value using, say, the name of the function itself.

A function is thus the antithesis of object-orientation, because it has no state to worry about. Stateless, or "functional", programming is a very elegant paradigm, with proponents who argue that all programming should be functional. In a database, or secondary storage, context, this is an extreme stance: functional relational programming would require copying completely every relation needing the slightest modification—perhaps gigabytes of copying for a few tens of bytes of update.

But functions provide the basis for a relational form of procedural abstraction, because functions are also special cases of relations. Functions are relations on their parameters such that providing values for a subset of the parameters leads to a complete evaluation of the remaining parameters. The only limitation is that this evaluation works only one way. In conventional programming language functions, the input parameters are fixed and the output parameter (often only one) is, too.

Relations, on the other hand, through T-selectors, find the values of any subset of fields given values for any other subset of fields. There may even be many such values for each "output" field, because many tuples match the values of the "input" fields.

Since there is such a close connection between a programming language function and a database relation, it would be a shame to ignore it and simply parachute conventional procedural abstraction into the relational programming language. That would require new syntax and a mismatch of concepts, where the same concept can in fact be used for both and no new syntax introduced.

So the simple idea is to generalize functions so that they can work in more than one direction, and to use the relational algebra to invoke them. We call the generalized functions *computations*. Computations are specialized relations and are used in the relational algebra. They are, of course, not explicit relations, with a given set of tuples giving the relationship among the fields; they are implicit, with the relationship given by algorithmic code.

Here is the computation that gives the full relationship among the coordinates, (x, y), of a two-dimensional point, and its coordinates, (x', y'), after the coordinate axes have been rotated an angle θ counterclockwise. (The **alt** keyword separates the alternative blocks of code that give each possible direction of calculation.)

comp $rotate(x, y, x', y', \theta)$ **is**
$\theta <- \arccos(x \times x' + y \times y')/(x \wedge 2 + y \wedge 2));$
alt
$\{\ x' <- x \times \cos\theta + y \times \sin\theta;$
$\ \ y' <- -x \times \sin\theta + y \times \cos\theta;$
$\}$ **alt**

$\{\ x <\!- x' \times \cos\theta - y' \times \sin\theta;$
$\ \ y <\!- x' \times \sin\theta + y' \times \cos\theta;$
$\}$ **alt**
$\{\ x <\!- x'/\cos\theta - y \times \tan\theta;$
$\ \ y' <\!- -x' \times \tan\theta + y/\cos\theta;$
$\}$ **alt**
$\{\ x' <\!- x/\cos\theta + y' \times \tan\theta;$
$\ \ y <\!- x \times \tan\theta + y'/\cos\theta;$
$\}$

(If $v = \sin\theta$ and the units are such that the velocity of light is 1, the latter two are the Lorentz transformation of space-time geometry and its inverse, an interesting connection between spatial rotations and spatio-temporal rotations.)

Here are two invocations, using T-selectors from the relational algebra (θ in degrees). These are ordinary rotations, and inverses of each other.

$[x', y']$ **where** $\theta = 90$ **and** $x = 1.0$ **and** $y = 1.0$ **in** *rotate*;
$[x, y]$ **where** $\theta = 90$ **and** $x' = 1.0$ **and** $y' = -1.0$ **in** *rotate*;

For multiple invocations, we can use a join. Suppose a relation,

points(x, y),

contains many points. Then new coordinates can be found for all these points after axis rotation.

let θ **be** 90;
rotPoints <− $[x', y']$ **in** *rotate* **join** $[x, y, \theta]$ **in** *points*;

We see that *rotate* is in all ways a relation.

This computation, and many others, are purely functional, in the technical sense that they have no state and always give the same results for the same inputs. Computations *with* state are also important, and lead us towards object orientation. The simplest possible computation with state is a counter.

comp *counter*(*current*) **is**
state *count* **initial** 0;
count <− *count* + *current*;
alt
current <− *count*;

The field, *current*, can be input, usually with a value of 1, and then *count* will be incremented. Alternatively, it is the output giving the current value of *count*.

Thus far, we have no machinery which lets us use two or more counters in the same program. Making multiple uses of *rotate* would be no problem, because *rotate* has no state. But if we wanted to count both sheep and goats, we would have to make a copy of the state, *count*, inside *counter*. This is called *instantiation*, and is the heart of object orientation. An object-oriented language would instantiate with a keyword such as **new**, as in

sheep = **new** *counter*
goats = **new** *counter*

but this is too low-level for a relational language. We do not want to have to deal with individual instances. Before dealing with this problem, we look at data abstraction.

10.2 Data Abstraction

Abstract data types are a programming language construct invented to permit programmers to extend the language to include new types of data. Suppose the base language contains arithmetic, but a geometrical programmer needs two-dimensional vectors and operators to translate and rotate them. It would be handy to have *vector* type for which one vector, v, can be translated by the distance and direction given by another vector, t, by writing $v + t$. Or to rotate v by the direction of a vector, r, and increase its magnitude by a factor which is the magnitude of r: $v \times r$. The abstract data type the programmer could define for this would provide the vector type, the two operators (which might perfectly well "overload" the existing arithmetic addition and multiplication operators of the language), and the capability of creating new instances. (The way we have formulated the problem, by the way, this new type would be the complex numbers, and scaling can be accommodated by scalar multiplication, a further overloading.)

Many programming languages now provide syntax for this kind of language extension. An insightful paper by Atkinson and Morrison [5] shows that new syntax is not needed, but only *first class* procedures. A programming language construct or type is first class if it can be used anywhere any other construct or type in the language can. This is a relative ranking. For instance, in most languages, integers are favoured over arrays. To make a case in point, integers can be returned as the result of a function, but arrays cannot: arrays are second class. In functional programming languages, functions are usually first class, which means that they can be the arguments or the results of other functions. Atkinson and Morrison point out that if functions and procedures are first class, they can be returned by an ordinary procedure, which can be used to define the abstract data type.

In this way, we can have a procedure

complex(*translate, rotate, scale*)

which returns the procedures to do addition, multiplication, and scalar multiplication of vectors (we do not address the issue of overloading). If, furthermore, procedures can be *persistent*, i.e., stored on secondary storage and shared among different programs, *complex* can be so stored, and becomes a *library* procedure.

We can work an example once we deal with instantiation, which we now see is needed both for computations (procedures) with state and for data abstraction.

10.3 Classes and Instantiation

We can now define a *class* to be an abstract data type with state. Alternatively, it is the set of all possible instances for that state, subject to the declarations and operations defined in the abstract data type. We can illustrate with the complex number example, since it is simple and frequently used as an example. We omit parts, such as the *rotate* computation, that tell us nothing new. (Note that *rotate* here will not be the same as *rotate* above, in section 10.1, but will use complex (vector) arguments, having the form, say $rotate(v, \theta, u)$.)

comp *complex*(*assign, translate, scale*)
{ **domain** r **real**; // real part
 domain i **real**; // imaginary part
 state $c(r, i)$ **initial** {(0.0, 0.0)};
 let r' **be** r; **let** r'' **be** r;
 let i' **be** i; **let** i'' **be** i;

 domain *realpart* **real**;
 domain *imagpart* **real**;
 comp *assign*(*realpart,imagpart*) **is**
 { **let** r **be** *realpart*;
  let i **be** *imagpart*;
  c <− $[r, i]$ **in**;

```
} alt ...                      // block for c → a, b

domain v(r, i);
domain t(r, i);
domain u(r, i);
comp translate(v, t, u) is
redop
{ let r be r' + r'';
  let i be i' + i'';
  u <- [r, i] in (([r', i'] in v) join ([r'', i''] in t));
} alt ...                      // blocks for subtraction

domain s real;
comp scale(v, s, u) is
{ let r be r' × s;
  let i be i' × s;
  u <- [r, i] in [r', i'] in v;
} alt ...                      // blocks for scalar division

}
```

(Note in *assign* that c is a singleton relation consisting only of the tuple containing the *realpart* and *imagpart* that have been passed as parameters. Thus c is created by a projection from no relation. Note also that we have identified addition of vectors, the first block of *translate*, as a reduction operator (**redop**), so that it can be used in the domain algebra to calculate aggregates.)

This is the class definition. No instances have yet been created. Since we are thinking relationally, we imagine that we might have a relation with many points which we wish to express as complex numbers.

manyPoints(*identifier, x, y*)

where values have already been supplied for the three fields in each of many tuples. We must combine the complex number class, expressed as the computation, *complex*, with the data in the relation *manyPoints*, so as to instantiate a complex number for each tuple and eventually to hide in its real and imaginary parts x and y, respectively. To keep our thinking from descending to the level of individual instances, we must avoid using anything like a **new** operator to instantiate.

How do we combine computation and relation? With a **join**, just as before. Here is the code. First, we instantiate a complex number on every tuple.

instances <− *complex* **join** *manyPoints*;

This creates a relation with visible and hidden fields

instances(*identifier, x, y, assign, translate, scale*) c

The visible fields are written within the parentheses. The last three are computations, and have the same value for all tuples. The hidden field, c, is the state from *complex*, and is inaccessible except through the visible computations, notably *assign*. It is shown above outside the parentheses. It is, of course, a nested relation, albeit a simple one.

Next, we assign x and y as the values of the real and imaginary components, respectively, of the hidden complex number.

update *instances* **change** *assign*(x, y);

Here we must use an **update** statement, because the *assign* operation is non-functional.

Third, we can get rid of the working coordinates, x and y, by making a new relation with just the complex data type.

complexPoints <− [*identifier, assign, translate, scale*] **in** *instances*;

We have coded this class in a way which leads to a problem. We did not really need a state for complex numbers, and we coded *translate* and *scale* to be independent of the state. This saves us from the unpleasant asymmetry that plagues doctrinaire object orientation, which is obliged to write $a + b$ as $a.plus(b)$ instead of, at worst, $plus(a, b)$. But we did not code *assign* to be independent of state, which we could have done. So now our complex vector has no name, and there are no operations that can get at it. But we have illustrated both the stateful and the state-free aspects of data abstraction, and we can leave the reader to fix our compromise by moving it either fully to or fully away from an abstract data type with state.

It should also now be apparent to the reader how a whole data structure such as the quad-edge representation of planar subdivisions can be represented as an abstract data type with a state consisting of three or more relations, but which we can manipulate with the domain (and relational) algebra, and with updates of the sort illustrated above. We can furthermore edit any instance of such a class with a specially-written update editor of the sort discussed in section 9.2.

It follows that a library of the geospatial data structures and operations we have discussed in this tutorial can be provided, with clean interfaces so that G.I.S. professionals are spared the details of the implementations we have gone through. What this approach offers those professionals is a straightforawrd combination of spatial with non-spatial data in all G.I.S. contexts.

10.4 Example: A Map ADT

In section 9.3.1 we anticipated the ability, which we now have, to construct the abstract data type for *Map*, so we give it as an example of the foregoing. We do not give the details of the construction of what we can call *QEADT*, the abstract data type containing all the operations needed for the quad-edge representation of maps, because this would be to repeat much of the foregoing tutorial. But we will suppose that *QEADT* exports operators for map overlay, Delaunay triangulation, spatial datacubes, and so on from the earlier parts of the tutorial, together with all useful G.I.S. capabilities that the reader should by now know how to write in the relational and domain algebras. We will in particular assume an *assign* operator similar to that written in section 10.3, above, for complex numbers, and a *fetch* operator which allows us to extract the three basic components of the quad-edge data structure.

These three components are the relations *QuadEdge, VertFace*, and *Geom*. They will be stored as hidden nested components of the *Map* relation. We procede, given *QEADT*(*assign, fetch, ..*), just as we did to create *instances*, above, except that we here build only a single instance, and so begin with no relation corresponding to *manyPoints*.

We pick up from our omission at the end of section 9.3.1, where the inspector has received the three components as separate relations, and called them *QE, VF,* and *G*, respectively. Each has its topological and spatial fields, and the field, *From:*, giving the source. We will extract a *To:* field for the map as a whole from one of these *From:* fields, supposing them to be the same in all three relations, so we can mail back the comments.

let *QuadEdge* **be** *QE*;
let *VertFace* **be** *VF*;
let *Geom* **be** *G*;
let *To:* **be** [**red max of** *From:*] **in** *QE*;

let *Subject:* **be** `"Comments"`;
Maps <− [*Subject:, To:, QuadEdge, VertFace, Geom*] **in**;
MapForm <− [*Subject:, To:, QuadEdge, VertFace, Geom, assign, fetch, ..*] **in**
(*Maps* **ijoin** *QEADT*);
update *MapForm* **change** *assign*(*QuadEdge, VertFace, Geom*);
Map <− [*Subject:, To:, assign, fetch, ..*] **in** *MapForm*;

With *Map* thus an instance of the *QEADT* abstract data type, we can return to section 9.3.1 for the inspector to make comments and mail them back to the contractor.

10.5 Example: Distributed Geospatial Objects

Despite the singular inappropriateness of the term "object", we have seen that the essential idea behind programming language object orientation is the concept of state, from which instantiation follows. The simplest objects are thus things like counters (whose state is the current count), stopwatches (whose state is the current elapsed time), or even stack calculators (whose state is the stacked values). These can all be distributed so as to be remotely accessed. What is needed is a protocol which supports requests for instantiation followed by method invocations on particular instances. Thus, if I want to use a counter, such as defined in section 10.1, provided by another site, I need to send a request for an instance, get back an identifier for that instance, then send requests with that identifier, and the parameters, for specific methods. The remote instance is then mine to use until I have finished with it.

The email technique used in section 9.1.3 for Internet communication can be used for such a protocol, giving us a complete, if rudimentary, architecture for distributed objects. We should observe, however, that while state is the essence of object-orientation, none of our Internet applications depend on one site holding a state for another. Such stateless communication is preferable and safer between remote sites, because, among other considerations, it is not affected by crashes. This dichotomy between stateless remote communication and state in objects is significant enough to have led people to speak of "stateless objects", which is of course a self-contradiction.[4] What they are trying to describe is the earlier notion of library procedure. We should perhaps introduce the term "confused object" to denote either a remote object (with state) or a remote library procedure (without state).

A nice way around the inflexibility and expense of purchasing entire software systems in order to use only a very small part of their capabilities, is to "rent" remote facilities, in the form either of remote objects or remote library procedures. The care and maintenance of these facilities is then entirely in the hands of the rental agency, and users need pay only for what they use, and when they do. Examples of such remote services for geoprocessing include display (zoom, pan, select, identify, layer control, colour style, etc.), access (download from server, upload from local, overlay layers from different sources, etc.), transformation (e.g., geographic coordinates to projected UTM coordinates), label placement (see section 6.4.5), map registration (see section 9.3.2), terrain analysis (interpolation, shading, etc.), three-dimensional visualization, and so on. These do not require state to be maintained remotely, and so can be done with remote library procedures.

[4]Nonsense may be classified by example: "jumbo shrimp" is the standard example of oxymoron, juxtaposed opposites; "colourless green ideas sleep furiously" goes further and juxtaposes contradictions; "All mimsy were the borogroves" uses invented but evocative words in correct grammar; "Wear a fierce mat, do" (hint: "Darn Body Oatmeal's Dream"), or, bilingually, "Un petit, D'un petit, sa tante en vol", takes wordplay to extremes. "Stateless objects" falls into the second level of nonsense. ("Object" itself can be seen as a rare, single-word oxymoron.)

The important part of an architecture to support this rental approach is called a *broker*. The job of a broker is to provide and advertise descriptions of the available remote services, to handle the users' interaction with them (usually by running the facility on the agency's machine), and to manage billing and payment. A handy acronym for a confused object broker is *COB*. This is the service provided by a collaboration of almost all providers of Internet enabled software, who have called it *CORBA* (Common Object Request Broker Architecture).

We conclude this section by suggesting a *geoCOB* which actually needs state, and therefore objects. Consider a weather recording system, with remote stations distributed across the nation, or the world. The state of each of these stations is its current readings of temperature, humidity, wind strength and direction, etc., or possibly a history of these readings. Each station offers methods to report on individual readings, to provide smoothed analyses of trends of single variables or correlations among different variables, and so on. At the next level up, the regional states are the results of these reports and analyses for all stations within the region, and the regional methods provide averages, standard deviations, and other statistical analyses. Finally, at the global level, averages and so on are cast into a choropleth map over all regions.

11 Conclusions

We have examined in detail calculations that represent the work of our group on geospatial data warehousing, mining, collaboration, and distribution over the Internet, and we have described them in tutorial form. In doing so, we have shown that geospatial operators, algorithms, and processes can be expressed in a single programming language, tuned only for secondary storage and not specifically for spatial data. As well as providing a theoretical framework for our specific project on collaborative geospatial decision-making, the tutorial establishes a computational basis for geographical information systems in general.

12 Acknowledgements

We are indebted to the Networks of Centres of Excellence program for support through the GEOIDE Project, GEODEM, and the Natural Sciences and Engineering Research Council of Canada. We acknowledge the work of many predecessors, whose imaginative innovations are so pleasing to explain: such as the quad-edge representation of spatial data; the ideas of datacubes, and of association, classification, and generalization data mining; the powerful underlying concept of data relations and their high-level operators; and the wonderful paradigms that underly modern programming languages.

We are grateful to Chris Gold for his advocacy of the quad-edge representation, and to Godfried Toussaint for bringing up to date our summary of computational geometry complexities in section 3.3, and for numerous other consultations.

References

[1] N. R. Adam and A. Gangopadhyay. *Database Issues in Geographic Information Systems.* Kluwer Academic Publishers, Boston, 1997.

[2] S. G. Akl and K. A. Lyons. *Parallel Computational Geometry.* Prentice-Hall, Engelwood Cliffs, N.J., 1993.

[3] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–43, November 1983.

[4] L. Anselin. Interactive techniques and exploratory spatial data analysis. pages 253–66. (in [39]).

[5] M. P. Atkinson and R. Morrison. *Persistent First Class Procedures are Enough*, volume 181 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1984.

[6] M. Barnsley. *Fractals Everywhere*. Academic Press, Inc., San Diego, 1988.

[7] Y. Bédard. Principles of spatial database analysis and design. pages 413–24. (in [39]).

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, Reading, Mass., 1987.

[9] H. Blum. A transformation for extracting new descriptors of shape. In W. Whaten-Dunn, editor, *Proc. Symp. Models for Perception of Speech and Visual Form*, pages 362–80, Cambridge, MA, 1967. M.I.T. Press.

[10] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[11] S-K Chang, Q-Y Shi, and C-W Yan. Iconic indexing by 2-d strings. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-9(3):413–28, May 1987.

[12] F. Chin, J. Snoeyink, and C. Wang. Finding the medial axis of a simple polygon in linear time. *Discrete Computational Geometry*, ??(??), 1999.

[13] N. R. Chrisman. *Exploring Geographic Information Systems*. John Wiley and Sons, Inc., New York and Toronto, 1997.

[14] E. Clementini, P. Di Felice, and P. van Oosterom. A small set of formal topological relationships for end-user interaction. In D. Abel and B. C. Ooi, editors, *Advances in Spatial Databases - Third International Symposium, SSD'93*, pages 27–95, Singapore, June 1993. Springer-Verlag. Lecture Notes in Computer Science LNCS 692.

[15] A. Clouâtre. *Implementation and Applications of Recursively Defined Relations*. PhD thesis, McGill University, School of Computer Science, 1987.

[16] E. F. Codd. Further normalization of the data base relational model. In R. Rustin, editor, *Data Base Systems*, pages 34–64. Prentice-Hall, Engelwood Cliffs, N. J., 1972.

[17] E.F. Codd, S.B. Codd, and C.T. Salley. Providing OLAP to user-analysts: An IT mandate. Technical report, E. F. Codd & Associates, Hyperion Solutions, Sunnyvale, CA, 1993. http://www.arborsoft.com/essbase/wht_ppr/coddps.zip, http://www.arborsoft.com/essbase/wht_ppr/coddTOC.html.

[18] D. J. Coleman. GIS in networked environments. pages 317–29. (in [39]).

[19] C. J. Date. *A Guide to the SQL Standard*. Addison-Wesley Longman Inc., Reading, Mass., 1997. (with Hugh Darwen).

[20] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwartzkopf. *Computational Geometry: Algorithms and Applications*. Spring-Verlag, Berlin Heidelberg, 1997.

[21] A. K. Dewdney. *The Turing Omnibus: 61 Excursions in Computer Science*. Computer Science Press, Rockville, MD, 1989.

[22] M. Egenhofer. Spatial SQL: A query and presentation language. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):85–96, Jan 1994.

[23] M. J. Egenhofer and J. R. Herring. Categorizing binary topological relations between regions, lines, and points in geographic databases. Technical Report 94-1, NCGIA, Orono, ME, 1994.

[24] A. A. Freitas and S. H. Lavington. *Mining Very Large Databases with Parallel Processing*. Kluwer Academic Publishers, Boston, 1998.

[25] C. M. Gold. Private communication. A quad-edge implementation, July 1997.

[26] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichert, M. Venkatarao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–53, 1997.

[27] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4:74–123, 1985.

[28] M. J. Haigh. *An Introduction to Computer-Aided Design and Manufacture*. Blackwell, Oxford, 1985.

[29] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In Li-Yan Yuan, editor, *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 547–59, Vancouver, Canada, August 1992. Morgan Kaufmann.

[30] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2000.

[31] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Proc. 1998 Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'98)*, pages 144–58, Melbourne, Australia, April 1998. Springer Verlag.

[32] V. Harinarayan, A Rajarman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 205–16, Montreal, Canada, June 1996. ACM Press.

[33] Environmental Systems Research Institute Inc. *Understanding GIS, The ARC/INFO Method*. GeoInformation International, a division of Pearson Professional Limited (United Kingdom) and John Wiley & Sons, Inc. (United States), Redlands, Calif., 1996.

[34] Environmental Systems Research Institute Inc. *Using ArcView GIS*. GIS by ESRI, 1996.

[35] Ian Johnson. *Understanding MapInfo, a Structured Guide.* Archaeological Computing Laboratory, University of Sydney, Sydney 2006, Australia, 1996.

[36] D. N. Jump. *AutoCAD Programming.* TAB Books Inc., Blue Ridge Summit, PA, 1989.

[37] Morris Kline. *Mathematical Thought from Ancient to Modern Times.* Oxford University Press, New York and Oxford, 1990. In three volumes.

[38] D. T. Lee. Medial axis transformation of a planar shape. *I.E.E.E. Transactions on Patern Analysis and Machine Intelligence*, PAMI-4(4):363–9, July 1972.

[39] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, editors. *Geographical Information Systems: Principles and Technical Issues*, volume I. John Wiley & Sons, Inc., New York, Toronto, 1999.

[40] P. Marchand, Y. Bédard, and G. Edwards. A hypercube-based method for spatio-temporal exploration and analysis. Spatial predicate hierarchies, 2001.

[41] J. Marks and S. Schieber. The computational complexity of cartographic label placement. Technical Report TR-05-91, Harvard University, Center for Research in Computing Technology, 1991.

[42] F. P. Preparata and M. I. Shamos. *Computational Geometry An Introduction.* Springer-Verlag, New York, 1985.

[43] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[44] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company Ltd., Reading, Mass. & Don Mills, Ont., 1990.

[45] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. W. Zucker. Hamilton-Jacobi skeletons. Technical Report SOCS-01.10, McGill University, School of Computer Science, Sept. 2001.

[46] Mark Stephens. *MiniCad Version 6 Tutorial Reference Manual.* conVRgence Ltd., Barnet, Herts.

[47] G. T. Toussaint. Pattern recognition and geometrical complexity. In *Proc. 5th International Conference on Pattern Recognition*, pages 1324–47, Miami Beach, December 1980. I.E.E.E.

[48] A. Valdivia-Martinez. Relix implementation of GIS queries: ARC/INFO and MapInfo. Master's thesis, McGill University, School of Computer Science, June 1998.

[49] P. van Oosterom. GIS interoperability. pages 385–400. (in [39]).

[50] I. H. Witten and E. Frank. *Data Mining.* Morgan Kaufman, San Francisco, 2000.

[51] S. Yuan and C. Tao. Development of conflation components. In *Proceedings of Geoinformatics'99*, Ann Arbor, Mich., June 1999.

A Equivalence of Sets and Sequences for Cycles in Graphs

We have used two important representations for the topological component of spatial data, the set and the sequence. This appendix shows how to map between them, and thus that they are equivalent. Since cycles (e.g., of vertices around a polygon, or edges around a vertex or a face) are important, we discuss disjoint cycles in a graph. The results are general for this case, but an example helps to follow them.

G_{seq}(id	seq	el)	$\Leftrightarrow$	G_{set}(s	e)
f	1	a		a	f
f	2	f		b	c
e	1	b		c	d
e	2	c		d	e
e	3	d		e	b
e	4	e		f	a

(The *id*entifier of each cycle in G_{seq} is one of the elements in the cycle, which must be unique if the cycles are not disjoint.)

We use the programming notation, not SQL, for the selections, projections, and joins in these mappings.

To map from the sequence representation to the set representation is easy and requires only the domain algebra.

let s **be** el
let e **be par succ of** el **order** seq **by** id

The worst-case cost is $N \log N$, because of the sorting needed for the partial functional mapping. However, each cycle can be processed separately, and since we can expect each cycle to be small enough to fit into RAM, a good implementation is likely to cost only n transfers from and to secondary storage, with negligible in-RAM sorting and processing costs. (Here, N is the number of records, and n is the (smaller) number of blocks.)

To map back from the set to the sequence representation requires more sophistication. We use the **union** operator, akin to SQL's, which combines two relations on the same fields by taking the set union of the records. We also use, **comp**, the *natural composition*, which is just **join**, except that the join fields are omitted from the result. This saves us the complication of some of the renaming. Both of these were defined in section 5.3.

The mapping procedure joins G_{set} to itself, using **comp**, as many times as the length of the longest cycle, and introduces a nested relation in the intermediate results to keep track of the sequence of the elements of the cycles. G_0 is G_{set} augmented by a *level* count, $l_0 = 1$, which will generate the sequence numbers. In the loop, G_1 accumulates all cycles that have returned to their starting points, G_2 holds the cycles that have not returned to their starting points, and G_3 actualizes the increasing level count, l, and the growing sequences, q, in the join of G_{set} with itself. G_2 eventually becomes empty if G_{set} contains only cycles, and the loop condition may be read "something in G_2", i.e., while G_2 is not empty..

let l_0 **be** 1
let l_1 **be** $l + l_0$
let l **be** l_1 //rename
let q_0 **be relation** (e, l)

```
let q₁ be q union relation (e, l)
let q be q₁                     //rename
G₀ <−[s, e, l₀, q₀] in G_set
G₁ <−φ                          //empty
G₃[s, e, l, q <−s, e, l₀, q₀]G₀ //rename
while [] in G₂
{     G₁ <−G₁ union [s, q] where s = e in G₃
      G₂ <− where s ≠ e in G₃
      G₃ <−[s, e, l, q] in [s, e, l₁, q₁] in (G₂[e comp s]G₀)
}
let id be s                     //rename
let el be e                     //rename
let seq be l                    //rename
G_seq <− [id, [seq, el] in q] where (s = equiv max of s by [e] in q) in G₁
```

The last four lines take the result of the loop, namely all the cycles and their sequences accumulated in q, selects a representative of each cycle by arbitrarily taking the one with the largest value of the s field, which becomes the *id*. $[e]$ **in** q projects the set of elements of the cycle from the sequence of elements of the cycle: the sequence is different for each starting point, but the set is the same for all. Finally, the projection flattens the nested relation by the anonymity of [*seq, el*] **in** q.

If G_{set} were modified to include an identifier for each cycle (which it could get from G_{seq}), the implementation could cluster on this field, and the cost of this inverse mapping could also be linear in transfers from and to secondary storage, with the loop being executed in RAM for each cluster. This supposes that each cycle is small enough to fit into RAM, a plausible supposition for spatial data.

B Terminology

Any collection of ideas with commercial potential is inadequately specified.
— Corollary of Murphy's Law

Spatial knowledge has many roots—geometry, topology, graphics, computer-aided design and geographical information systems being only the principal ones. As a result, terminology is rampant and diverse. We here provide a unification of terms as used in this tutorial. We attempt to stay as close to the specialized usages as is consistent with unification. We characterize and illustrate terms rather than defining them.

B.1 Terminology: Geometry and Topology

The *geometrical* aspects of spatial features are concerned with metric properties, particularly leading to notions of distance and related ideas of length, area, angle, etc. This is usually captured using a coordinate system, with one coordinate for each dimension.

Topology abstracts away from metric ideas to focus on more general relationships. The topological terms we shall use in two dimensions are *vertex, edge*, and *face*. (These are characterized, circularly, below.) Although not metric, topology is not free from numbers or equations, as Euler's formula, $v + f = e + 2$ (1750, for the topological equivalent of a sphere) illustrates.

A *point* is both a topological and a geometrical concept. Euclid ($\sim$ 300 BC) defined it as that which has no part, but we lesser mortals must elaborate. Geometrically, a point has coordinates. In general, we will distinguish between arbitrary points in a space and points that occur in the features we are processing (and hence that must be represented by the data structure). For instance, the program to find which polygon contains a given point is supplied an arbitrary point, which we need not retain, but holds the points that are the corners of the polygons in its internal representation. The latter are *nodes*. Special nodes are called *vertices*, a term used only topologically. A *vertex* is a cycle of edges or a cycle of faces.

An *arc* is a geometrical concept, consisting of at least two nodes, which are its endpoints, connected by sections of lines (possibly curved, possibly straight). Extra nodes in an arc can be used to shape it: one intermediate node might specify a circular arc (*circarc*); several intermediate nodes can specify a spline arc (*splinarc*) or a *polyline*. The reason for distinguishing nodes from vertices, above, is that when we abstract from geometry to topology, only the two endpoint nodes of an arc must abstract to vertices. Any intermediate nodes need play no topological role.

A *side* is an arc with exactly two nodes.

An *edge* has two meanings. Topologically, it connects two vertices. Geometrically, it is a geodesic side, that is, the shortest distance between the two points (nodes) that are its vertices. Thus, an edge(G) in the plane or in higher-dimensional Euclidian space is a straight side, while an edge(G) on a sphere is a section of a great circle. The context should always be clear, so we do not introduce a separate term for a straight edge. On the other hand, context does not always make clear whether we are saying "edge" topologically or geometrically, so we use *edge(T)* or *edge(G)* when necessary. In the following, we add the (G) or (T) prefix to each new term for clarity, even if the term is only ever used in one way.

(Our "arc(G)" has the same intent as the *chain* of the Spatial Data Transfer Format (SDTF), in that it may have zigs and zags. But our "node(G)" is intended to separate the zigs and zags, whereas "node" in SDTF marks only the endpoints of the chain. This resembles a geometric variant of our "vertex(T)", but not quite. We use "vertex(T)" to conform to the usual use in topology, and "node(G)" to indicate a point which must be represented in the data structure. If we defined *endnode(G)* to be the end node of an arc, then "arc(G)" and "endnode(G)" would be synonymous, respectively, with the SDTF "chain" and "node".)

A *polygon(G)* is a cycle of sides, or a cycle of nodes (or a cycle of angles: "$\gamma\omega\nu\iota\alpha$" in Greek). Special cases are *simple polygons*, whose sides do not intersect except at nodes (the "dreaded bowtie" is a counterexample) and which have no "holes"; and *embedded polygons*, whose nodes all lie within a prescribed two-dimensional space, and whose sides are edges(G). A *planar polygon* is embedded in the plane and has straight edges; a *spherical polygon* is embedded in a sphere; a *toroidal polygon* is embedded in a torus; counterexamples are a closed organic molecule, or a beaded necklace (on its owner, with or without knots). When the context is clear, "polygon" means "planar polygon". A *facet*, or a *hedron*, is a simple, planar polygon.

A *face(T)* is a cycle of edges, or a cycle of vertices.

A *subdivision(T)* is a set of (usually connected) vertices, edges and faces. It subdivides a topological sphere, the plane (which is often combined with the "point at infinity" to become a topological sphere), a torus, a Klein bottle, or any topological manifold, with or without an "inside". In a subdivision, vertices and faces are *dual(T)* to each other: a dual subdivision may be created by "rotating" edges so that the cycle of edges that was a vertex becomes the cycle of edges of a face, and vice versa.

A *layer(G)* is a geometrical specialization of a subdivision: coordinates are given to the vertices to make them nodes. In geographical information systems, this is a layer of a *map*. In computer-aided design, it is a layer of a *model*. We sometimes say "map" or "model" to mean "layer", where it is clear.

A *polyhedron(G)* is a geometrical specialization of a connected subdivision in which each vertex terminates at least three different edges. All nodes of a *regular polyhedron* lie on a sphere embedded in Euclidean three-space; there are five, and the duals(T) of regular polyhedra are regular. Other polyhedra with all edges(G) of equal length include the prism and the triangular hexahedron.

A *polytope* is a polygon, polyhedron, or analogous figure in any number of dimensions.

For contrast, and to emphasize that we have not used them above, here are four further terms. A *line(G)* is a set of points in a two-dimensional space whose coordinates are related by a linear equation. It has no nodes. Its points are *ordered*. It can also be represented in a *dual(G)* two-dimensional space by the point whose coordinates are the two coefficients of the linear equation. A *plane(G)* can similarly be defined in a three-dimensional space. More generally, a *curve(G)* is a set of points in the plane whose coordinates are related by a single equation, and a *surface(G)* is such a set in three-space.

We have used *section(G)* to refer to a portion of a line or a curve with two endpoints, and *semi-infinite edge(G)* to refer to a portion of a line with one endpoint.

B.2 Glossary

This section is an index to the first use and definitions of terms in the text. "*(G)*" refers to geometrical properties and "*(T)*" refers to topological properties.

9Inter. See sections 3.5.3, 6.5.3.
Abstract Data Type. See section 10.2.
Aggregate Operation. See sections 3.2, 5.2.
Angle bisector. See sections 6.4.1, 6.4.2.
Arc(G). See section B.1.
Association Mining. See section 8.2.
Buffering Operator. See section 3.1.
Chain(G). See section B.1.
Checkout-Checkin Sharing. See section 9.
Choropleth Map. See section 3.2.
Circarc(G). See section B.1.
Class. See section 10.3.
Classification Mining. See section 8.1.
Closeness Operator. See section 3.1.
Computation, **comp**. See section 10.1.
Containment Operator. See section 3.1.
Cross-Tabs. See section 7.1.
Cube Operator. See section 7.1.
DataCube. See section 7.1.
Deadlock. See section 9.1.1.
Decision Tree. See section 8.1.
Delaunay Triangulation. See section 6.3.
Discussion Sharing. See section 9.
Divide-and-conquer. See section 6.3.
Drill-Down. See section 7.1.
Dual(G). See section B.1.