National Research Council Canada | Conseil national de recherches Canada

Canada

Reprinted from

Réimpression de la

Canadian Computational Intelligence Journal

Revue canadienne de Intelligence informatique

A logic programming approach to cartographic map overlay

PETER Y. F. WU AND W. RANDOLPH FRANKLIN

Volume 6 • Number 2 • 1990

Pages 61–70

Printed in Canada / Imprimé au Canada

A logic programming approach to cartographic map overlay

PETER Y. F. WU

IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

AND

W. RANDOLPH FRANKLIN

Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, U.S.A.

Received April 4, 1989

Revision accepted March 9, 1990

Cartographic map overlay is the process of superimposing two maps into one to convey information in spatial correlation. A map refers to one in vector representation: a two-dimensional spatial data structure of nodes, chains, and polygons. We present a map overlay system developed in Prolog. The system adopts a relational approach to data structuring. We represent geometric entities and their relationships as facts, and encode geometry algorithms in the rules. Set-based operations perform data processing. To speed up the search for chain intersections, a uniform rectangular grid is imposed over the object space for spatial sorting by distribution. We sort out potentially intersecting edge segments to those occupying some common grid cell. Each bucket, if non-empty, is implemented as a Prolog fact identifying the grid cell for random access. Geometric intersections are calculated using exact rational arithmetic implemented in Prolog. Numerical accuracy is preserved and we can identify all the special cases of tangent conditions. We can then guarantee topological consistency, and stability in the process of map overlay is therefore achieved.

Key words: map overlay, polyline intersection, Prolog, rational arithmetic.

La confection de cartes géographiques nécessite la superposition de deux cartes afin que l'on puisse mettre en correspondance des informations sur le plan spatial. La référence d'une carte à une autre se fait sous forme de représentation vectorielle, c'est-à-dire une structure bidimensionnelle de données spatiales de noeuds, de chaînes et de polygones. Cet article présente un système de superposition de cartes élaboré dans Prolog. Le système favorise une approche relationnelle à la structuration des données. Des entités géométriques sont représentées ainsi que leur relation comme faits; des algorithmes de géométrie sont ensuite encodés dans les règles. Des opérations basées sur des ensembles effectuent le traitement des données. Afin d'accélérer la recherche des intersections de chaînes, une grille rectangulaire uniforme est superposée sur l'espace de l'objet pour permettre un tri spatial par répartition. On distingue les segments de contours qui font l'objet d'intersections de ceux qui occupent une cellule commune de la grille. Chaque cuvette qui n'est pas vide est traitée comme un fait Prolog identifiant la cellule en vue d'un accès aléatoire. Les intersections géométriques sont calculées à l'aide d'une arithmétique rationnelle exacte incorporée à Prolog. La précision numérique est préservée et il est possible d'identifier tous les cas spéciaux de conditions tangentes. Nous pouvons ensuite garantir la consistance topologique et assurer la stabilité du processus de superposition de cartes.

Mots clés : superposition de cartes, intersection polyligne, Prolog, arithmétique rationnelle.

[Traduit par la revue]

Comput. Intell. 6, 61–70 (1990)

Introduction

Map overlay is the process of superimposing two or more maps into one to convey spatial correlation information between the input maps. Figure 1 illustrates the traditional manual process of map overlay using transparencies over a light table. Automation of this process on the computer depends on how the map is represented. In the raster format, a map is an image. In the vector format, a map is a two-dimensional (2D) spatial data structure consisting of nodes, chains (polylines), and polygons. In the context of this paper, a map refers to one in the vector format. The map overlay problem is often also referred to as the polygon overlay problem, since it encompasses a number of geometric and topological computation problems.

Fundamental to the development of a solution for map overlay was the research effort in computational geometry during the 1970s. Solutions were reported for the basic problems in point-in-polygon test (Ferguson 1973) and polygon intersection (Eastman and Yessio 1972; Franklin 1972). Shamos and Hoey (1976) later developed a host of algorithms to efficiently solve many geometry problems. Algorithms specific to polyline intersection (Burton 1977; Little and Peucker 1979; Ballard 1981) were particularly important to map overlay. Nievergelt and Preparata (1982) formulated the plane-sweep approach for geometric intersection problems in general. Map overlay systems were not implemented until the mid 1970s (Tomlinson *et al.* 1976), but an object space geometric solution was first implemented in ODYSSEY (White 1978). The algorithm we implemented was due to Franklin (1983*a*), and to our knowledge it was not implemented elsewhere.

Using Prolog is a venture motivated by the quest for more suitable tools to implement geometry and graphics systems (Forrest 1988[1]). Swinson (1982, 1983) reported studies in using Prolog for architectural design. Gonzalez *et al.* (1984) compared Prolog to Pascal and concluded in favor of Prolog for computer-aided-design applications. Nichols (1985)

[1]Also presented in an invited address, Computational Geometry and Software Engineering, in the Second Annual Symposium on Computational Geometry, 1986, Yorktown Heights, NY.

Printed in Canada / Imprimé au Canada

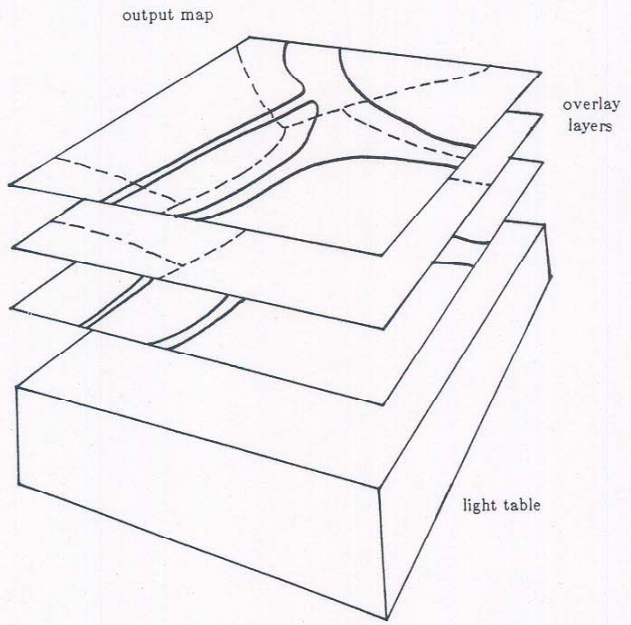
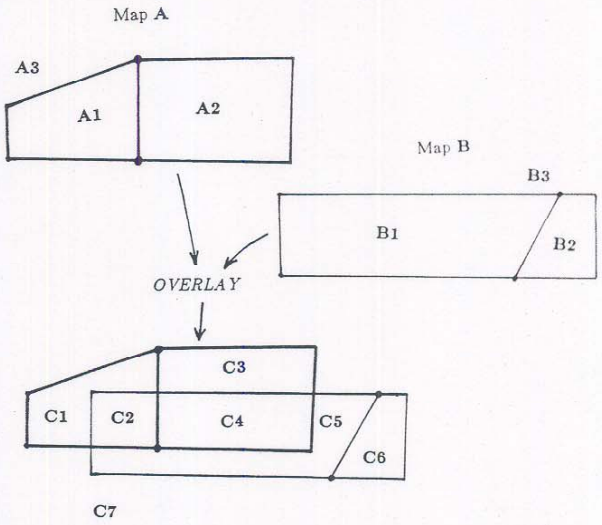


FIG. 1. The manual process of map overlay on transparencies.



Overlay Relationship:
over(C1,A1,B3).
over(C2,A1,B1).
over(C3,A2,B3).
over(C4,A2,B1).
over(C5,A3,B1).
over(C6,A3,B2).
over(C7,A3,B3).

FIG. 2. Map overlay and the overlay relationship.

implemented a subset of the graphics kernel system in Prolog. Prolog was also used in geometric modelling: solid construction by constraints (Bruderlin 1988) and octree modelling (Guerrieri and Grover 1986). We also reported some preliminary work of using Prolog in a number of geometry projects (Franklin *et al.* 1986).

In this paper, we present a system developed in Prolog to perform map overlay. The system decomposes the process of map overlay into a number of steps grouped into four major stages. Using Prolog, we naturally adopt a relational approach to data structuring. The strategy allows us to simplify the data structures involved. We represent instances of geometric entities and their relationships as "facts," and we encode geometry algorithms in "rules" to perform data processing. To speed up computing polyline intersections, we impose a uniform grid onto the object space in an approach similar to sorting by distribution. Numerical inaccuracy in computer arithmetic has bedeviled many overlay systems, since it destroys the topology. We use exact rational arithmetic implemented in Prolog. We can then preserve numerical accuracy and guarantee topological consistency, and we can properly identify and handle all special cases of tangent and coincidental conditions. Thus we have achieved a map overlay system completely stable in the process of computation. The last section will discuss some of our test results.

The map overlay problem

A map refers to a 2D spatial data structure consisting of nodes, chains, and polygons. A node is a point in the plane. It is a point of topological junction where chains begin or terminate. A chain is a directed polyline, a sequence of contiguous edge segments beginning at a node and terminating at a node (which may or may not be the same node). A chain does not intersect with any other chains in the map, nor with itself. The network of nodes and chains partitions the 2D plane into polygons. Each polygon is a connected region of

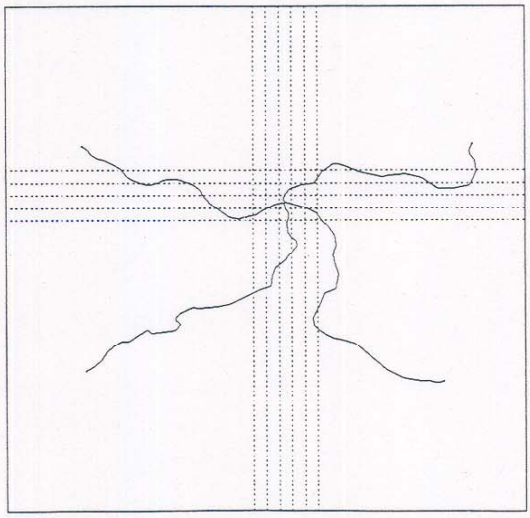


FIG. 3. A uniform grid isolates potentially intersecting edge segments. Note: When two chains intersect, they do so between two edge segments which occupy the same locality.

the 2D plane with polygonal boundaries. (A polygon with holes has more than one boundary.) The cyclic order of the chains around the boundary determines whether the interior of the polygon is to the inside or outside of the polygonal boundary.

The structure of nodes, chains, and polygons constitutes a map. However, with proper attributes, the set of chains alone suffices to carry the complete map information. Hence we can represent a map as a file of chain records, each having the following fields:

- chain identifier;
- beginning and terminating node identifiers;

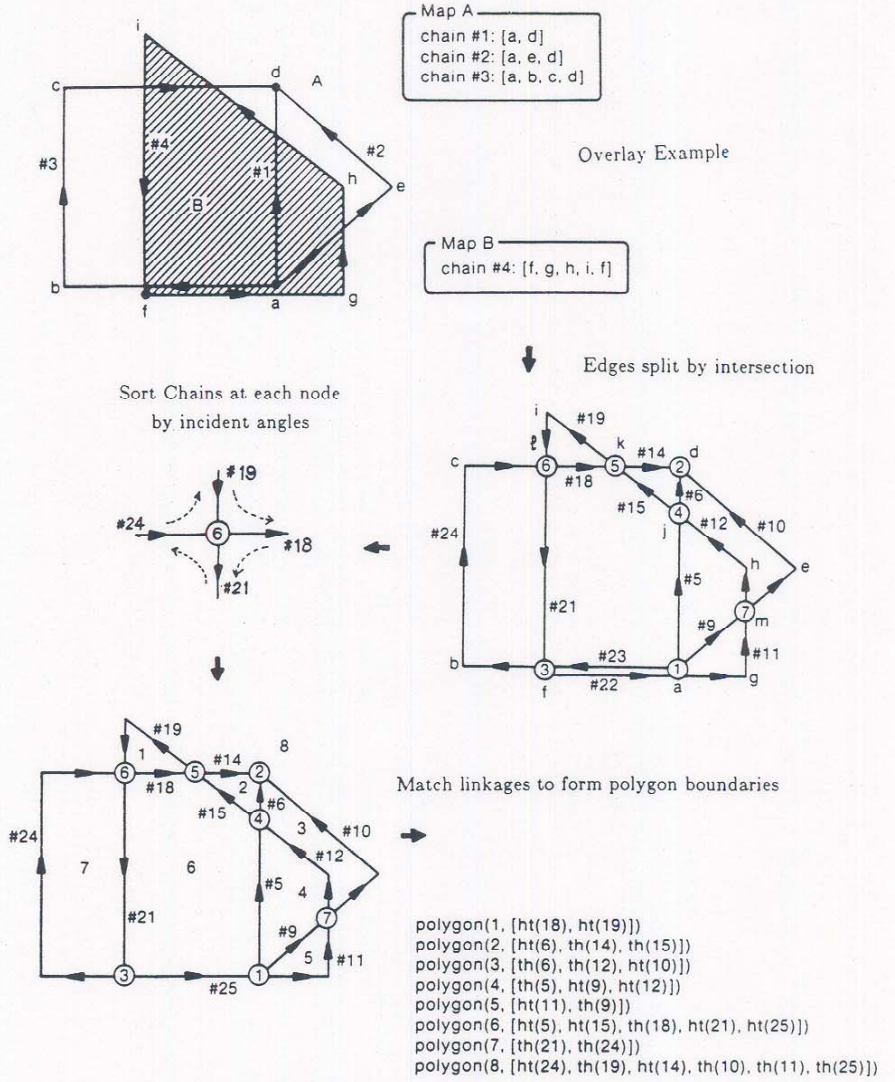


FIG. 4. Linking chains to form polygon boundaries.

- ordered list of (x, y) pairs for the polyline sequence; and
- identifiers of the polygons to the left and right.

Given this set of chains for a map, we can derive the nodes as well as the polygons.

Based on the terminology defined above, we can now describe the map overlay process in more precise terms. Given two input maps A and B, the map overlay process produces an output map C by superimposing A and B. New nodes are formed at the intersection points where chains of A intersect those of B. Map C consists of all the nodes of A and B, and the new nodes. Map C consists of all the chains of A and B except that those involved intersection are split up at the new nodes, that is, the intersection points. The network of nodes and chains in map C partitions the 2D plane into polygons, each of which is a resultant polygon from the intersection between a polygon in A and another one in B. An overlay relationship associates each polygon in C with these two polygons in A and B, respectively. To solve the map overlay problem, we have to construct the output map C and determine the overlay relationship. Figure 2 illustrates a map overlay process and the overlay relationship.

A map overlay system in Prolog

Our system divides the complicated process of map overlay into four major stages. Each is further subdivided into various steps. The following presents an overview and a brief description of the four different stages in the process. Then we will describe each stage in further details.

Stage 1. *Chain intersection*. Split intersecting chains and generate the new nodes.

Stage 2. *Polygon boundary formation*. Link up chains from stage 1 to form polygon boundaries.

Stage 3. *Overlay polygon identification*. Identify for each output polygon boundary the input polygons.

Stage 4. *Boundary containment resolution*. Resolve the containment relationship between polygon boundaries to identify polygons and holes.

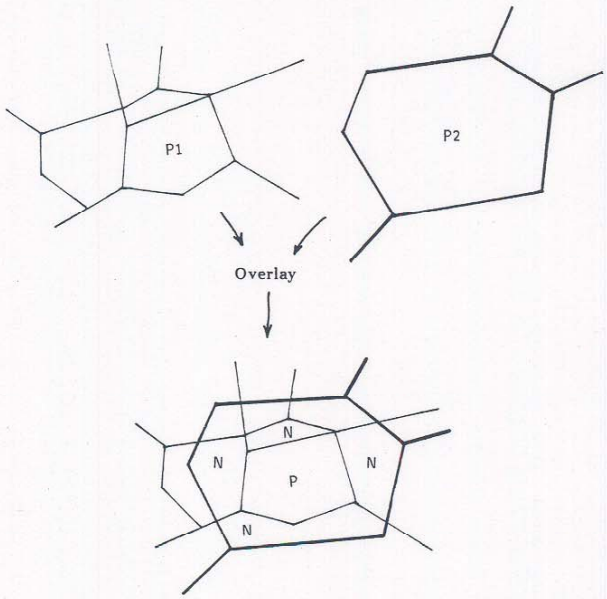


FIG. 5. When P = P1, P and all its neighbors (Ns) are inside the same P2.

Stage 1. Chain intersection

Given two sets of chains, one from each input map, we determine the intersecting chains and split them at the intersection points. In search of intersections, we decompose each chain into its edge segments and cast a uniform grid over the object space. The grid isolates potentially intersecting edge segments to those which occupy some common grid cell, as illustrated in Fig. 3. We can then determine the intersections by pairwise comparison of edge segments within each grid cell. This stage involves the following steps.

Step 1.1. Determine the appropriate cell size for the uniform grid.

Step 1.2. Cast the grid over the edge segments; determine for each edge segment the grid cells it occupies and collect the edge segments over each grid cell.

Step 1.3. Determine intersections in the set of edge segments in each grid cell; split the edge segments and generate a new node for each intersection point.

Step 1.4. Connect the edge segments back into chains, split at the intersection points.

In the section on chain intersection, we will discuss further the grid strategy and the issue of determining the appropriate cell size for the grid.

Stage 2. Polygon boundary formation

The network of nodes and chains from stage 1 should partition the plane into polygons. We can therefore link up the chains around the polygon corners at each node. Each completed cycle of chains identify a polygon boundary. This stage involves the following steps.

Step 2.1. Identify for each chain the nodes where it begins and terminates, and calculate the incident angles.

Step 2.2. Sort the chains by the incident angles around each node into proper cyclic order. Each adjacent pair of chains identify a corner of an output polygon at the node. Link up the adjacent chains in the proper cyclic direction.

Step 2.3. Match the chain linkages and connect them until a list of directed chain linkage begins and ends with the same entry; the list identifies the chains along the boundary of a polygon in its proper cyclic order.

Figure 4 illustrates the overlay process of two simple maps A and B up to identifying 8 completed lists of polygon boundaries. When we have identified all the polygonal boundaries in the output map, we need to determine for each boundary list the two original polygons in the two input maps.

Stage 3. Overlay polygons identification

Each output polygon is formed by the intersection of two input polygons, one in each input map. However, each polygon boundary may or may not be involved in intersection. If a boundary is involved in an intersection, the list of boundary chains will consist of chains from both input polygons. We can then determine the input polygons by examining the original input chains along the boundary. But if a boundary is not involved in any intersection, all the chains along the boundary come from the same input polygon in one of the input maps, the polygon being completely contained in another polygon of the other input map. We can determine one input polygon P1 form the boundary chains; the unknown P2 from the other map is the containment polygon. Observe that all the neighbors of P1 in the output map must be contained also in P2, as illustrated in Fig. 5. We can therefore search through the neighbors of P1 until we find one with the boundary intersecting the containment polygon.

The search fails only when the entire connected group of polygon boundaries are not involved in any chain intersection with the other input map. In this case the entire connected group is completely contained in the same polygon of the other input map. We will have to take a point from the group and test it for containment against polygons of the other input map. The point-in-polygon test will find the containment polygon. The following outlines the steps in this stage.

Step 3.1. For each polygon boundary, examine the input chains to determine the input polygons. If it is involved in chain intersection, we can find both input polygons.

Step 3.2. If it is not involved in chain intersection, we can find one of the input polygons. Search the neighbors to determine the containment polygon.

Step 3.3. If the search fails, identify the group of connected neighboring polygons together. For each connected group, perform point-in-polygon test to determine the containment polygon in the other input map.

Stage 4. Boundary containment resolution

Two polygons may intersect to form more than one polygon. Moreover, since we have defined a polygon to be a connected subset of the 2D plane, each polygon may have more than one polygonal boundary. In the preceding stage, we have determined the input polygons for each output polygon boundary, we must also determine which of the boundaries refer to the same connected piece of output polygon.

We examine only the polygon boundaries that are formed from the same pair of input polygons. We then divide them into two groups by the cyclic order: (*i*) positive polygon boundaries, and (*ii*) the holes which are negative. Each positive boundary identifies an output polygon. Each negative boundary identifies a hole contained in one of the positive polygons, except for the holes in the outside polygon, a region which extends to infinite on the 2D plane. The out-

side polygon contains holes only and no positive polygon boundary. The following describes the steps taken to resolve containment relationships and identify the holes from the polygon boundaries.

Step 4.1. Gather the polygon boundaries which are formed from the intersection of the same pair of input polygons.

Step 4.2. Divide into two groups according to the boundary cyclic order: positive ones are polygons and negative ones are holes.

Step 4.3. Determine containment polygon for each hole by point-in-polygon test. Update the polygon identifier for each hole to that of its containment polygon.

Once finished with this step, we have the complete sets of nodes, chains, and polygons of the output map, as well as the overlay relationship.

Data structuring and logic programming

Unlike conventional programming languages, Prolog is a declarative language. This means, at least in theory, that a Prolog program is not a prescribed set of instructions to solve a particular problem, but is instead a description of the objects and their relationships involved in the problem which provides information to solve the problem. The Prolog facts and rules constitute such a description. In our map overlay system, we represent geometric entities and their association as facts, and we encode the geometric and topological relationships into the rules.

The way Prolog maintains the facts provides us with a relational approach to data structuring. For example, a map consists of nodes, chains, and polygons, represented in the following:

```
node(Node-id, [x, y]).
chain(Chain-id, Node1, Node2, [[x, y], ...], Poly1, Poly2).
polygon(Polygon-id, [B1, B2, ...]).
```

A "node" fact comprises the Node-id and $[x, y]$, the coordinates of the node. A "chain" fact comprises the Chain-id; the beginning and ending Node-id, Node1 and Node2; the list of $[x, y]$ vertex coordinates of the polyline from Node1 to Node2; and the Polygon-id's Poly1 and Poly2 on the left and right of the chain. A "polygon" fact comprises the Polygon-id and a list, [B1, B2, ...], of the boundary chains in proper cyclic order around. Each Bi may be ht(Ci) for Chain Ci from head to tail, or th(Ci) from tail to head. Note that the Polygon-id is not necessarily unique for each "polygon" fact, but each fact identifies a polygon boundary. Similarly, the overlay relationship is represented by the set of "over" facts:

```
over(Poly-A, Poly-B, Poly-C).
```

The "over" fact states that output polygon Poly-C is formed from the intersection of input polygons, Poly-A from map A and Poly-B from map B. During the process of map overlay, our system also generates different interim data files for use in the subsequent steps. These we also keep as sets of Prolog facts. For example,

```
incident(Node-id, Chain-id, Incident-Angle).
```

represents the chain-to-node incidence relationship.

Given the map data represented as facts in the Prolog data base, the map overlay system performs data processing on the facts by set-based operations. Each step in the process applies a certain operation on the entire sets of geometric entities. Obviously, Prolog system predicates retract and assert can provide for simple set-based operations. However, we must be careful when simultaneously iterating and updating the same set of facts. Specifically, retract and assert may affect the sequence of iteration. We opt to avoid this in many cases. When we do need to have simultaneous assert and retract, we use a repeated_retract predicate.

```
repeated_retract(INSTANCE):-
  repeat,
  (not(call(INSTANCE)), !, fail;
    retract(INSTANCE)).
```

The predicate succeeds to retract an INSTANCE until no more exists, and it works independent of specific assert and retract implementations.

While the assert and retract predicates provide for access to the data base, the Prolog rules encode geometry algorithms to perform data processing. A common paradigm in our map overlay system uses pattern matching to propagate geometric and topological properties. The scheme involves retrieving facts matching in certain properties to construct new facts until no matching patterns can be found. For example, in linking chains to form polygon boundaries, we first form the corners around each polygon. Each corner is a fragment of a polygon boundary. Whenever two fragments exist and they can be connected, we retract both fragments and assert the new connected fragment. When no such fragments exist, we have all the polygon boundaries. Consider the set of "linkage" facts in the following.

```
linkage(a, b, [a, b]).
linkage(b, c, [b, c]).
linkage(c, d, [c, d]).
linkage(d, a, [d, a]).
```

These are the fragments of a quadrilateral. These fragments can be connected if they match the linkage properties. We identify a new polygon when we have a complete cycle in the linkage.

```
connect(A, A, [A|L]) :-
  !, assert(polygon(L)).      % completed cycle.
connect(A, B, L1) :-
  retract(linkage(B, C, [B|L2])),
  append(L1, L2, L3),
  assert(linkage(A, C, L3)).      % fragment matched.
```

The "connect" rule matches fragments of polygon boundaries to link them up. We use repeated_retract to perform set operation on the entire collection of linkages.

```
match_all_linkages :-
  repeated_retract(linkage(A, B, L)),
  connect(A, B, L),
  fail.
match_all_linkages.   % match until linkages exhausted.
```

When "match_all_linkages" finishes, we have the quadrilateral represented as

```
polygon([a, b, c, d]).
```

Note that the direction designators ht( ) and th( ) for the chains in the polygon boundary have been omitted for clarity

FIG. 6. The United States map of state boundaries.

FIG. 7. The United States map perturbed with new points introduced.

and ease of presentation. When linking chains in the map overlay system, chains have their specific directions in the polygon boundaries.

Chain intersection

The cost determining factor in the map overlay process is in computing chain intersections. Guevara (1983) ascertained this by asymptotic complexity analysis. In the introduction, we alluded to a number of existing algorithms for chain intersection (Burton 1977; Little and Peucker 1979; Ballard 1981; Nievergelt and Preparata 1982). Either they deal with two chains at a time or the entire set of chains to exploit spatial coherence, these algorithms all attempt various ways to search the chains to determine the intersection points. They aim at achieving a low worst case time complexity, and they approach the chain intersection problem very much like sorting by comparison.

In our map overlay system, we adopted a rather different approach. When two chains intersect, they do so between two edge segments which occupy the same locality. We impose a uniform rectangular grid over the object space and take each grid cell as a bucket in sorting by distribution. We can then avoid comparing edge segments unless they occupy some common grid cell; and we do not have to search into the chains, since we are dealing with the constituent edge segments.

For each edge segment, we can determine the grid cells it occupies in time linear to the number of grid cells occupied. The total time required to distribute all the edge segments to the buckets for the grid cells occupied is therefore proportional to the total number of (grid-cell, edge-segment) pairs. We, however, have to be careful with the data structure for the grid so that empty grid cells do not take up any storage space. In our map overlay system, the data structure for each non-empty grid cell is a Prolog fact with a unique functor name generated to identify the grid cell. For example, grid cell No. 344 is the fact

```
g344([E1, E2, ...]).
```

where [E1, E2, ...] is the set of edge segments in the bucket. Hence, each bucket is accessed directly by linear hashing, and that empty cells do not take up any storage, not do they need CPU time to check that they are empty. The grid method avoids pairwise comparison between edge segments that are not in the same vicinity, and it attempts to focus the attention directly on the intersection points by sorting out potentially intersecting edges to within those that share some common grid cell.

An interesting question in the grid method is how to determine the appropriate size for grid cells to set up the grid. Since the grid is intended to isolate cases of intersecting edge segments, it may seem desirable to use smaller grid cells so that two edge segments will less likely occupy a common cell unless they intersect. On the other hand, smaller grid cells result in a grid with more cells, and will take more preprocessing time in distributing the edge segments to the grid cells. We are therefore looking for some appropriate grid cell size to optimize the average case performance. However, formal analysis of the expected performance is difficult. Difficulties arise in part because of substantial complications in the mathematics, but also because there is hardly any consensus in an appropriate model characterizing the expected input data. Franklin *et al.* (1988, 1989) collected plentiful experimental results and showed that performance is relative insensitive to the grid-cell sizes within a range around the optimal. In our results reported in this paper, the grid cells are rectangular, and we have chosen the length and width to be the average length of the edge segments projected along the axes, rounded up to integral number of grid cells.

Exact rational arithmetic

A problem in dealing with geometry on the computer is the numerical inaccuracy of computer arithmetic. It is, however, not so much a problem with accuracy, but that of topological consistency, which can be affected by inaccurate numerical results. Early in the 1970s, Malcolm (1972) and Gentleman and Marovich (1974) identified and reported peculiarities of floating-point arithmetic. Map overlay systems are therefore prone to instability when dealing with situations such as nearly coincidental points, almost touching chains, and other tangent conditions. White (1983) called for attention to the proper use of coordinates in mapping systems. Franklin (1984) examined alternative models of arithmetic for geometry and cartography on the computer. We developed two arithmetic packages in Prolog: BIG for integers with no overflow limit and XQ for exact rational numbers built upon BIG (Wu 1986). Our map overlay system installed these packages to calculate geometric intersections.

The package BIG implements integer arithmetic with virtually no overflow limit. A BIG number is a list of integer terms in Prolog. The absolute value of each term is limited

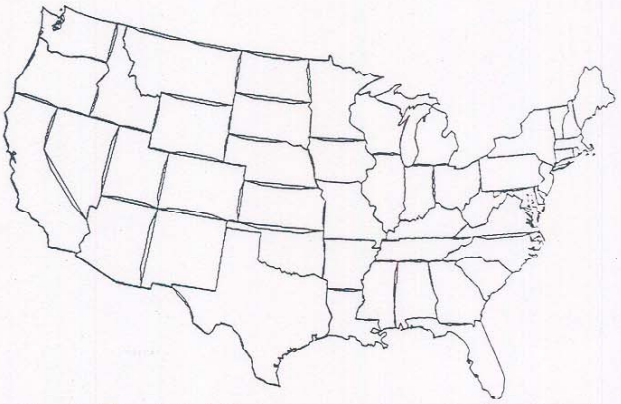


FIG. 8. The United States map overlay on itself perturbed.

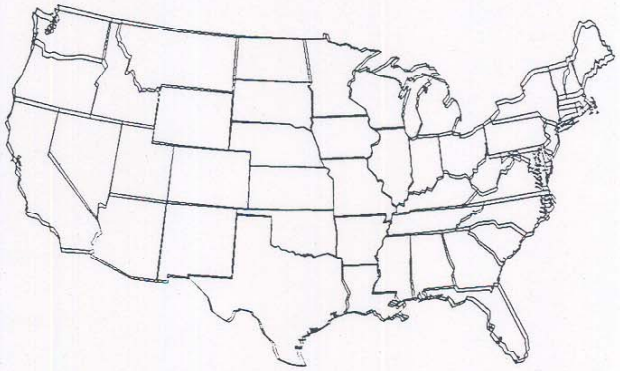


FIG. 9. The United States map overlay on itself rotated by 1°.

to less than a certain maximum, say M. The list $[a_1, a_2, \ldots, a_n]$ represents a value given by

$$a_1 + a_2M + a_3M^2 + \ldots + a_nM^{n-1}$$

Since Prolog does not impose a limit to the length of a list, the BIG number has virtually no overflow limit. The following illustrates some BIG numbers, taking $M = 100$.

0 is represented as [ ]
1 is represented as [1]
100 is represented as [0, 1]
12345 is represented as [45, 23, 1]
–10045 is represented as [–45, 0, –1]
–1 is represented as [–1]

The package XQ builds upon BIG to implement exact rational arithmetic. An XQ number is a fraction in which both denominator and numerator are BIG integers. We represent an XQ number by a division expression with BIG integer operands, and we omit the denominator when it is unity. Hence, we have compatibility, since a BIG integer is a legal XQ number with denominator equal to unity. The following illustrates some XQ numbers.

0 is represented as [ ]
33 is represented as [33]
11/310 is represented as [11] / [10, 3]
48/100 is represented as [12] / [25]
–321/100 is represented as [–21, –3] / [0, 1]
–1 is represented as [–1]

For the arithmetic operations, since Prolog allows operator overloading, the syntax for arithmetic expressions can remain unchanged. Furthermore, evaluation of XQ arithmetic can simply follow a set of rules, each of which defines one arithmetic operation on XQ operands. We merely need new operators to call for evaluation in different arithmetic domains. While Prolog commonly uses "is" for arithmetic evaluation, we added "are" for BIG and "isx" for XQ arithmetic expressions. The following examples illustrate their use.

```
X is (3 × 4) / (2 × 5).     % integer: 1.
X are (3 × 4) / (2 × 5).    % BIG integer: [1].
X isx (3 × 4) / (2 × 5).    % XQ number: [6]/[5].
```

Hence, we can install BIG and XQ arithmetic in a highly modular fashion.

In map overlay, the arithmetic in geometric intersection is based on the calculation of edge segment intersections. Given the input coordinates in rational numbers, the coordinates of an intersection point is always a rational number, since it is the solution to a linear system of equations with rational coefficients. We have closure of numerical representation in using rational arithmetic for geometric intersections. Much more important is that we can preserve numerical accuracy and guarantee topological consistency in calculating geometric intersections. We identify properly all the special cases of coincidental and tangent conditions by checking absolute equality instead of setting a tolerance value. Hence neither coincidental nor tangent conditions can cause our system to fail. Stability in the process of map overlay is therefore achieved.

On the cost for exact rational arithmetic, we contend that it does not affect the asymptotic behavior of the algorithm. Given a fixed precision of the operand values, evaluation of an arithmetic expression is only proportional to the depth of the expression tree structure. For a static arithmetic expression, as in the case of map overlay, its evaluation is a constant time operation. Hence, the total CPU time required is proportional to the size of the data volume. The multiplicative factor may be large, depending on the specific implementation. But rational arithmetic will not impose a need for CPU time more than linear to the size of the data volume. In other words, the precision level needed in the output values is limited, given that of the input values. Specifically, let *maxint* be the maximum value of an integer word. If the input coordinates are all integer values $\leq \sqrt{maxint}$, and we do arithmetic in double precision, we can express all the output values in exact rational number with both numerators and denominators within the limit of *maxint*. Since we can minimize software emulation of the arithmetic involved, we can improve the performance. Hence, we need exact rational arithmetic implemented on integers with no overflow limit only when we want to further operate on the calculated results.

A further complication occurs, since in cartographic map overlay, we rarely just overlay two input maps, but several of them, to generate an output map. We argue that the depth of the arithmetic expression tree structure does not need to be deeper than that in the case of two input maps. Since a new node in the overlay of many input maps is still only an intersection point of two edge segments from two of the input maps, we have the same arithmetic expression for the new coordinates derived from the input values. However,

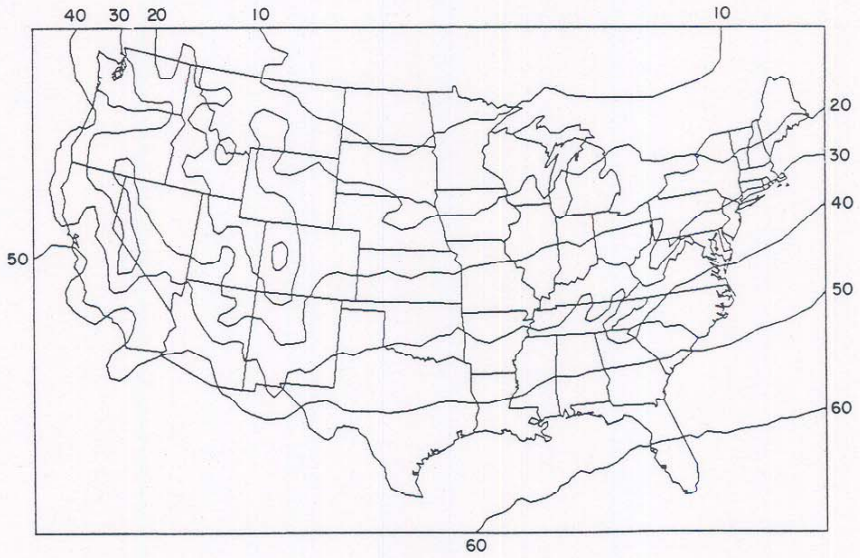


FIG. 10. The United States map with January isotherms. (Temperatures in °F.)

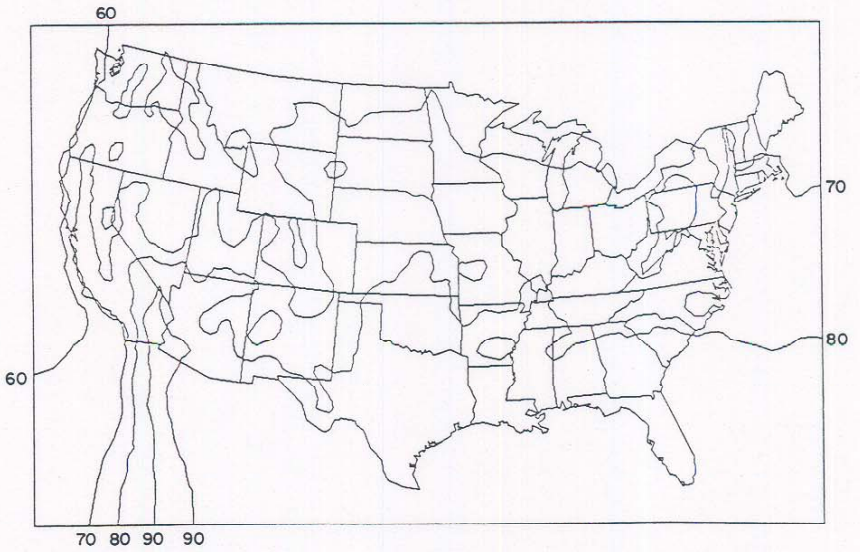


FIG. 11. The United States map with July isotherms. (Temperatures in °F.)

in the process of handling two maps at one time, we need to preserve the coefficients of the straight line equations for each edge segment involved in intersection, so that we do not recalculate these coefficients using the calculated coordinates from a previous overlay process.

Implementation and results

The map overlay system and the arithmetic packages, except the graphics output software, are implemented in C-Prolog, which is a Prolog interpreter written in C (Pereira *et al.* 1986). The system runs on a Sun/2 microsystem computer.

To test the stability of a map overlay system, there are some tests commonly known: to overlay a map on itself and on a slightly perturbed version of the same map. We have done both tests and an additional test of a map over itself rotated by a small angle. We used a United States map of the state boundaries (excluding Alaska and Hawaii). The map has 164 chains, 913 edges, and 861 nodes and vertices, and 50 polygons. Figure 6 shows the map. To overlay the map on itself, our system generated a uniform grid of 113 × 73 grid cells. The U.S. map survived the test and passed undistorted. The second test involved expanding the U.S. map complexity. We slightly shifted the midpoints of those edges that would not cross over to another polygon. We repeated the process to generate a map shown in Fig. 7. Figure 8 shows the output map from our overlay system. In the third test, we rotated the original U.S. map at about the center by approximately 1°. Figure 9 shows the output map.

For some "normal" tests, we overlay the U.S. map with the temperature maps of January and July. Figures 10 and 11 show respectively the output maps, and Tables 1 and 2 present the usage of resources, namely CPU time and storage space, in each of the stages involved in the overlay process.

TABLE 1. Resources for overlay of U.S.A. with January isotherms

Stage	CPU (s)	Storage (K words)
1. Intersect chains	12 527	889
2. Form polygons	2 105	440
3. Identify overlay	248	440
4. Resolve containment	160	522
Total	15 040	889

TABLE 2. Resources for overlay of U.S.A. with July isotherms

Stage	CPU (s)	Storage (K words)
1. Intersect chains	14 100	1070
2. Form polygons	2 162	471
3. Identify overlay	412	423
4. Resolve containment	246	543
Total	16 920	1070

In each case, the entire overlay process took more than 4 h of CPU time on the Sun/2 system. The performance of our system is not acceptable for practical use. However, we do note that this is not a necessary trade-off for the stability we have achieved.

Conclusion

We presented a map overlay system developed in Prolog. The system decomposes the map overlay process into four major stages, each into various steps. The decomposition allows us to simplify the data structures involved. Prolog provides for a relational approach to data structuring. We represent the geometric entities and relationships in Prolog facts, and we encode geometry algorithms in Prolog rules to perform data processing. To speed up searching for chain intersections, we impose a uniform rectangular grid over the object space and take each grid cell as a bucket to sort the edge segments by distribution. Thus we isolate potential intersections to only those edge segments that occupy some common grid cell. Each bucket, if non-empty, is implemented by a Prolog fact accessed by linear hashing of the identifier for the grid cell. Using exact rational arithmetic implemented in Prolog, geometric intersections are calculated without round-off errors. We can therefore identify exactly all the special cases of tangent conditions for proper handling. Topological consistency is guaranteed and we have achieved complete stability in the map overlay process. This is verified in our tests on stability. The general performance of our experimental system is not good, due both to the Prolog interpretation and to our implementation of exact rational arithmetic. However, we did contend that the asymptotic growth of CPU usage for rational arithmetic is only linear to the size of the data volume.

Acknowledgements

This work was supported in part by the National Science Foundation under grant No. ECS-8351942 and the Data Systems Division of IBM Corporation. We would like to express our gratitude toward the reviewers for their careful work and their comments.

BALLARD, D. 1981. Strip trees: A hierarchical representation for curves. Communications of the ACM, **24**(5): 310–321.

BRUDERLIN, B. 1988. Rule-based geometric modelling. Doctoral Dissertation, Swiss Federal Institute of Technology, Zurich, Switzerland.

BURTON, W. 1977. Representation of many-sided polygons and polygonal lines for rapid processing. Communications of the ACM, **20**(3): 166–171.

EASTMAN, C.M., and YESSIO, C.I. 1972. An efficient algorithm for finding the union, intersection, and differences of spatial domains. Technical Report, Computer Science Department, Carlegie-Mellon University, Pittsburgh, PA.

FERGUSON, H.R. 1973. Point-in-polygon algorithms. Technical Report, Urban Data Center, University of Washington, Seattle, WA.

FORREST, R.A. 1988. Geometric computing environments: some tentative thoughts. *In* NATO ASI Series, Vol. F-40, Theoretical foundations of computer graphics and CAD. *Edited by* R.A. Earnshaw. Springer-Verlag, Berlin, Germany. pp. 185–197.

FRANKLIN, W.R. 1972. ANOTB: routine to overlay two polygons. *In* Collected algorithms. *Edited by* D. Douglas. Laboratory for Computer Graphics and Spatial Analysis, Harvard University, Cambridge, MA. Vol. III, pp. 16–22.

———1983*a*. A simplified map overlay algorithm. Proceedings, Harvard Computer Graphics Conference, Vol. I, Cambridge, MA.

———1983*b*. Adaptive grids for geometric operations. Proceedings, Sixth International Symposium on Automated Cartography, Ottawa, Ont., Vol. II, pp. 230–239.

———1984. Cartographic errors symptomatic of underlying algebra problems. Proceedings, First International Symposium on Spatial Data Handling, Zurich, Switzerland, Vol. II, pp. 190–208.

FRANKLIN, W.R., WU, P.Y.F., SAMADDAR, S., and NICHOLS, M. 1986. Prolog and geometry projects. IEEE Computer Graphics and Applications, **6**(11): 46–55.

FRANKLIN, W.R., CHANDRASEKHAR, N., SESHAN, M., and AKMAN, V. 1988. Efficiency of uniform grids for intersection detection on serial and parallel machines. Reprint, Graphics International '88, Geneva, Switzerland.

FRANKLIN, W.R., CHANDRASEKHAR, N., KANKANHALLI, M., AKMAN, V., and WU, P.Y.F. 1989. Efficient geometric algorithms for CAD. *In* Geometric modeling for product engineering. *Edited by* M.J. Wozny *et al.* Elsevier Science Publishers, New York, NY.

GENTLEMAN, W.M., and MAROVICH, S.B. 1974. More on algorithms that reveal properties of floating point arithmetic units. Communications of the ACM, **17**(5): 276–277.

GONZALEZ, J.C., WILLIAMS, M.H., and AITCHISON, I.E. 1984. Evaluation of the effectiveness of PROLOG for a CAD application. IEEE Computer Graphics and Applications, **4**(3): 67–75.

GUERRIERI, E., and GROVER, V. 1986. Octree solid modelling with Prolog. Reprint, First International Conference on Applications of Artificial Intelligence to Engineering Problems, Southampton, United Kingdom.

GUEVARA, J.A. 1983. A framework for the analysis of geographic information system procedures: the polygon overlay problem, computational complexity and polyline intersection. Ph.D. thesis, State University of New York, Buffalo, NY.

LITTLE, J.J., and PEUCKER, T.K. 1979. A recursive procedure for finding the intersection of two digital curves. Computer Graphics and Image Processing, **10**(2): 159–171.

MALCOLM, M.A. 1972. Algorithms to reveal properties of floating point arithmetic. Communications of the ACM, **15**(11): 919–951.

NICHOLS, M. 1985. A Prolog implementation of the graphics kernel system. Master thesis, Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY.

NIEVERGELT, J., and PREPARATA, F.P. 1982. Plan-sweep algorithms for intersecting geometric figures. Communications of the ACM, **25**(10): 739–747.

PEREIRA, F., WARREN, D., BOWEN, D., BYRD, L., and PEREIRA, L. 1986. C-Prolog user's manual (version 1.5). Department of Architecture, University of Edinburgh, Edinburgh, United Kingdom.

SHAMOS, M.I., and HOEY, D. 1976. Geometric intersection problems. Proceedings, 17th IEEE Symposium on Foundations of Computer Science, Newark, NJ. pp. 208–215.

SWINSON, P.S.G. 1982. Logic programming: a computing tool for the architect of the future. Computer Aided Design, **14**(2): 97–104.

———1983. Prolog: a prelude to a new generation of CAAD. Computer Aided Design, **15**(6): 335–343.

TOMLINSON, R.F., CALKINS, H.W., and MARBLE, D.F. 1976. Computer handling of geographical data. The United Nations Educational, Scientific, and Cultural Organization, New York, NY.

WHITE, D. 1978. A new method of polygon overlay. Proceedings, Advanced Study Symposium on Topological Data Structures for Geographic Information Systems, Vol. 1, Harvard University, Cambridge, MA.

WHITE, M. 1983. Tribulations of automated cartography and how mathematics helps. Proceedings, Sixth International Symposium on Automated Cartography, Ottawa, Ont., Vol. I, pp. 408–418.

WU, P.Y.F. 1986. Two arithmetic packages in Prolog: infinite precision fixed point and exact rational numbers. Technical Report IPL-TR-082, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, NY.