1-1-1989

Object Oriented Locational Analysis

Marc P. Armstrong  
*The University of Iowa*

Paul J. Densham

David A. Bennett  
*University of Iowa*
OBJECT ORIENTED LOCATIONAL ANALYSIS

Marc P. Armstrong
Departments of Geography and Computer Science
316 Jessup Hall
The University of Iowa
Iowa City, IA 52242
BLAMGPD@UIAMVS.BITNET

Paul J. Densham
National Center for Geographic Information and Analysis
and
Department of Geography
State University of New York at Buffalo
Buffalo, NY 14260
GEOPUD@UBVMS.BITNET

David A. Bennett
Department of Geography
The University of Iowa

ABSTRACT

An object oriented approach to the analysis of spatial data is presented. We begin by describing object oriented programming, and then define a structure of spatial and analytical objects within the problem domain of locating facilities. From these objects a set of object classes and inheritance structures is created. The spatial and analytical objects are represented using frames. A Smalltalk implementation of a heuristic location-allocation algorithm is also described.

1.0 INTRODUCTION

Programmers and software engineers have often struggled to reduce the data requirements, programming effort, and amount of computation needed to solve locational analysis problems. For decades it has been a common practice to simplify and abstract reality by treating locational problems in terms of vector objects such as nodes and links (e.g. Rushton, Goodchild, and Ostresh, 1973). An additional mechanism that has been used to increase the tractibility of locational problems is to employ data structures that are designed to reduce the number of computational steps required in their solution (e.g. Hillsman, 1980). In this paper we use object oriented programming techniques to formally define spatial objects. These objects are then integrated with new ways of structuring data (analytical objects) to compute solutions to locational problems. Our general goal is to develop an object oriented locational analysis system based on the use of heuristic models. In the final section of the paper we describe how we have implemented a prototype system using an object oriented language.
2.0 OBJECT ORIENTED PROGRAMMING

Kjerne and Dueker (1988) succinctly describe two basic questions which must be addressed when conceptualizing software in object oriented form: What things (objects) exist in the problem domain? How do they behave? A program design in the object oriented environment, therefore, begins by decomposing the problem into a set of fundamental objects and their relationships. Note, however, that where procedural languages force the user to abandon these concepts at the level of implementation, object oriented programming preserves this view throughout the software development process (Thomas, 1989). In object oriented languages objects have a description in the form of private memory, a public interface, and operations (methods) which allow objects to be systematically manipulated. During the execution of an object oriented program, objects interact by message passing. Objects respond to messages sent by other objects or input through the user interface, by performing methods, and thus as messages are passed among objects their local state is altered according to the instructions that are sent. The procedural knowledge needed to respond to a message is bundled into the object with the data. By encapsulating data and methods within the same structure objects behave in a unique fashion when responding to messages. If a response to a given message can take multiple forms, however, the system is said to exhibit polymorphism.

Objects belong to semantic classes, which are defined on the basis of shared descriptive elements and methods. Classes, furthermore, support the property of inheritance wherein objects from subclasses assume the behavior and attributes of the superclass. This leads to a hierarchical representation of the problem domain which is useful in modeling spatial relationships (Gahegan and Roberts, 1988), and which lends conceptual simplicity and economy to the representation of spatial objects (Morgan and Glick, 1988).

Object oriented methodologies are being used for spatial data management (Kjerne and Dueker, 1988; Egenhofer and Frank, 1988a) because they are user-oriented and capture important relationships among real-world concepts through the use of objects, classes and inheritance (Potter and Trueblood, 1988). Object oriented techniques have been used to develop graphical interfaces to support interactive spatial query and model development (Egenhofer and Frank, 1988b; Konsynski and Sprague, 1986; Morgan and Glick, 1988; Hurrion, 1986). The object oriented approach is especially suitable for creating prototype systems because it supports a modular view of application software development and provides a programming environment in which code reusability is a key feature (Luqi, 1989; Gupta et al., 1989). In this light, the object oriented approach is useful for developing modular testbed systems for solving locational problems. Finally, object orientation plays an important role in the development and use of intelligent databases which will become increasingly common in the future (Parsaye, et al., 1989).
In the previous section the general features of object oriented languages were described. These features are useful for developing software systems for analyzing spatial data, and we illustrate this in the domain of locational analysis.

3.1 Logical Structure of Classes and Inheritance

A common set of entities and relationships exist in most locational problems (Armstrong and Densham, 1990). Locational models optimize an objective function, act upon spatial objects and their attributes, and result in the production of analytical objects (e.g. allocation tables). The superclass Locational Model in Figure 1 can be refined on the basis of the objective function chosen for the particular analysis at hand. These objective functions have been shown to exist as instances of a class (Hillsman, 1984); each variant of the Unified Linear Model is distinguished by the selection of appropriate constraints.

Each instance of a spatial object is characterized by a distinctive private state defined by location and topology, yet it remains in a general class from which it inherits properties. In their object oriented system, Morgan and Glick (1988) use spatial object classes and a hierarchical set of class descriptions based on features described in the Proposed Standard for Digital Cartographic Data (NCDCDS, 1988: 23-28). Although our class structure is more tightly focused on networks, it can be easily expanded to include other object classes. Networks are built from links which are spatial objects, and which have unique distinguishing attributes (e.g. length, direction). Nodes bound links, but they also represent the location of demand in the network, and specify the location of candidate sites in the locational model.

Figure 1. Conceptual schema of spatial and analytical objects. Analytical objects are specified by bold rectangles.
3.2 Description of OOLA

In this section, we focus attention on the kernel of the system: the shortest path algorithm (Dijkstra, 1959) and the location-allocation heuristic (Teitz and Bart, 1968). We implement these elements using several strategies developed to reduce computation (Densham, 1989); by exploiting the general structure of location-allocation models, these strategies become more effective as the size of problems increase. To illustrate the representation of objects we use frames (Winston, 1984; Kuipers, 1978) which serve as analogs of objects (Parsaye, et al., 1989: 184). The following spatial primitive objects (Armstrong, et al. 1989) are used in our analyses:

(Node
  (AKO ($Value (Spatial_object)))
  (ID ($Value (Node_Id)))
  (Candidacy ($Value (Candidate (Unit binary))))
  (Weight ($Value (Population_Values)))
  (Location ($Value (X_coordinate))
    ($Value (Y_coordinate))))

(Link
  (AKO ($Value (Spatial_object)))
  (Origin_Node ($Value (Node_ID)))
  (Number_of_Links ($Value (Valency))))
  (Destination_Node_ID ($Value (Node_ID)))
  (Distance_To_Node ($Value (Distance))))

Each frame has an identifier (e.g. node) and a nested set of attributes which comprise slots in the frame (e.g. candidacy). Each slot is assigned a value, or list of values.

3.2.1 Shortest Path Algorithm The shortest path algorithm used in our analyses generates a data structure that records all feasible allocations of demand nodes to potential facility sites (candidates) and the relevant distance, or cost, incurred in each allocation. These distances, weighted by the demand at each node, are the coefficients in the objective function of the location-allocation model. While solving a model, the Teitz and Bart heuristic requires data in two forms: the closest facility to each demand node and all the demand nodes that each facility can serve. A distance string data structure (object) supplies these data and reduces access and retrieval times at the cost of temporarily storing data to make processing more efficient.

Two forms of distance strings exist: candidate and demand strings. Candidate strings optimize retrieval of all demand nodes that can be served by the candidate site; demand strings optimize retrieval of the closest facility to a particular demand node. Each string has a base node, a list of other nodes and the distance between each of these nodes and the base node ordered by increasing distance value (Hillsman, 1980, pp. 81-90). A demand string lists all potential facility sites (candidates) that can serve a
given demand node:

(Candidate_node_name
  (AKO ($Value (Candidate_node)))
  (Weight ($Value (Weight_value)))
  (Candidacy ($Value (Candidate)))
  (String_length ($Value (Number_of_connections)))
  (Consists_of ($Value (Demand_node_ID Demand_node_name
    Weighted_Dist_to_Candidate))))

Because candidates are stored in ascending order of distance from the demand node, searching for the closest facility is minimized. String length is determined by the total number of candidates that may serve the demand node and may be restricted to include only those candidates within a specified distance of the demand node.

A candidate string contains a list of all demand nodes that can be served by a given candidate node, sorted by distance.

(Demand_node_name
  (AKO ($Value (Demand_node)))
  (Weight ($Value (Weight_value)))
  (Candidacy ($Value (Candidate)))
  (String_length ($Value (Number_of_connections)))
  (Consists_of ($Value ((Candidate_ID Candidate_name
    Weighted_Dist_to_Candidate))))

String length is determined by the number of demand nodes on the network that may be served by the candidate node. We have implemented a version of Dijkstra’s (1959) algorithm to generate each type of distance string.

3.2.2 Location-Allocation Our implementation of the Teitz and Bart heuristic (1968) requires data about: potential sites for facilities (candidates); candidates where facilities are located in the initial solution; any facilities fixed at their current location; and for each demand node, a measure of its demand and the distances to all candidate nodes that may serve it. These data are used in a solution process that systematically evaluates marginal changes to the current solution. A substitution relocates one facility temporarily and compares the resulting objective function with the current solution. Usually, P substitutions (the number of sites in the solution) are evaluated sequentially; the one leading to the biggest decrease in the objective function is adopted — termed a swap — yielding a new current solution. Termination occurs when the objective function cannot be reduced by pairwise substitutions of all candidates in the solution for all those not in the solution.

We use an allocation table (an analytical object) to calculate the change in the value of the objective function that results from a substitution (Densham, 1989). An allocation table has six rows and N columns (the number of demand nodes on the network), each containing information about the demand node. The allocations of each node to sites in the current solution are recorded in the first
four rows: respectively, the identifier of the closest facility’s site; the weighted distance from the node to this site; the second closest site’s identifier; and the weighted distance to the second site. If a node is not served by two facilities, a zero is entered in place of a site identifier, and a penalty value—an arbitrarily large positive number—is used as the weighted distance. The fifth and sixth rows of the table are used to calculate the change in the objective function resulting from a substitution.

(allocation_table
   ((allocation_demand_node_number
     ($row1_value(closest_facility_ID))
     ($row2_value(weighted_dist_to_closest_facility))
     ($row3_value(next_closest_facility_ID))
     ($row4_value(weighted_dist_to_next_closest_facility))
     ($row5_value(swap_candidate_ID))
     ($row6_value(weighted_dist_to_swap_candidate))))

The distance strings provide two sets of information to the allocation table: the first is used to calculate the change in the objective function; the second is needed to update the table after a swap is made. To determine the net change in the objective function for any substitution, rows five and six of the allocation table must contain two lists—the demand nodes that can be served by the candidate site and the weighted distances that would be incurred if these allocations were made. These data are found in the candidate string of the candidate being evaluated in the substitution. Three classes of relations, occurring between the nodes listed in rows one, three, and five, must be found and summed to calculate the net change in the objective function (Densham, 1989). For every candidate, each calculation is evaluated before a substitution is made.

After a candidate has been evaluated for substitution against all the non-fixed sites in the current solution, one or more of the substitutions may lead to a smaller objective function than that of the current solution. The substitution resulting in the biggest decrease in the objective function is made by replacing the old site’s node identifier in the current solution with that of the new site—a swap. The allocation table is updated to reflect the new allocations of demand nodes to facilities, and four classes of relations occurring between the nodes listed in rows one, three and five, must be found and processed (Densham, 1989). Sometimes, additional information is needed to update a demand node’s secondary facility allocation: a list of the candidates that can serve the node and the associated weighted distances. This information is in the node’s demand string.

4.0 A SMALLTALK IMPLEMENTATION OF OOLA

To implement the location/allocation algorithm seven new classes (Table 1) were added to the Macintosh Smalltalk environment (Digitalk, 1988). Of these classes, AllocTable, AllocString, and StringNodes hold the
procedural knowledge needed to complete spatial analyses. An instance of class AllocTable maintains the allocation table, the set of candidates in the current solution, and information pertaining to the objective function. The allocation table is implemented as a SmallTalk dictionary which is an indexed collection of objects. Each object is represented by a column in the allocation table and thus, consists of a demand node identifier (the index key) and a six element array (the object). Together the key and object are referred to as an association.

Table 1. Location/Allocation Class Structure

<table>
<thead>
<tr>
<th>Object</th>
<th>METHODS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc</td>
<td>solveFor: with:, swapFor: using:,</td>
</tr>
<tr>
<td>AllocTable</td>
<td>setForStartingSolution: forDemand:,</td>
</tr>
<tr>
<td></td>
<td>changeInObjFor: swappedWith:</td>
</tr>
<tr>
<td></td>
<td>inTableNode, getsCopyOfCandidate:,</td>
</tr>
<tr>
<td></td>
<td>initialize.</td>
</tr>
<tr>
<td>AllocString</td>
<td>atNode: addLink:, initialize,</td>
</tr>
<tr>
<td></td>
<td>returnTheString</td>
</tr>
<tr>
<td>CandidateString</td>
<td></td>
</tr>
<tr>
<td>DemandString</td>
<td>initNode, isLargerThan:, isSame:,</td>
</tr>
<tr>
<td>StringNodes</td>
<td>NodeID, LinkWt:, nodeWt, nodeID.</td>
</tr>
</tbody>
</table>

The subclasses CandidateString and DemandString inherit behavior (methods) from the AllocString class. Instances of CandidateString and DemandString are dictionaries indexed by a node identifier. Each object in these dictionaries is an association between a node identifier and a distance string which, in turn, is an instance of the class OrderedObject. The OrderedObject class accommodates a sorted collection of any object which implements the isLargerThan method (see Table 1). In particular, for each key in the CandidateString dictionary there exists an OrderedObject collection of StringNodes. Each node in this collection is a complex object comprised of a demand node and the weighted distance to the key candidate node. It is the responsibility of the class StringNodes to insure that these objects are added into the collection in ascending order of weighted distance. The class OrderedObjects relies on polymorphism to determine the relative order of
arbitrary objects.

In the object oriented environment a location allocation program is structured as a sequence of messages which request actions to be performed on spatial and analytical objects. For example, requests may be issued to:

1. initialise allocation strings
2. add individual links in the string at node N
3. set up the allocation for the starting solution using demand string D
4. and solve the problem for demand string D with candidate string C.

These phrases are easily translated into message selectors and methods. The following code segment, for example, adds a link to an allocation string:

```Smalltalk
atNode:theNode addLink:theLink
```

An analysis begins by creating global instances of the allocation table and distance strings. Candidate and demand strings are filled and, given a user-supplied starting condition, the system initializes the allocation table. The Teitz and Bart heuristic uses this information to generate a solution by calculating the change in the objective function due to substituting each candidate node not in the current solution with each candidate node that is. If a given substitution results in a reduction in the objective function a swap is made. Pseudocode for this procedure follows:

```Smalltalk
solveFor:demandstring with:candidatestring
candidatestring associationsDo:[:cNode|
  (nonSwaps=totalSwaps)
  ifFalse:[(solution includes:(cNode key))
    ifFalse:["if candidate in solution don't test swap"
      self getsCopyOfCandidate:cNode.
      solution do:[:sNode|
        allocTable do:[:tNode|
          self changeInObj:cNode swappedWith:sNode
          inTableNode:tNode.
          newObj:= newObj + deltaObj.
          ]."do allocTable"
        newObj<bestObj)
        ifTrue:[ bestObj:=newObj.
          bestSwap=sNode].
      ]. "do solution"
  (bestObj<0)
  ifTrue:[self swapFor:bestSwap
    using:demandstring.]
  ifFalse:[nonSwaps:=nonSwaps+1].
]. "if not solution includes cNode"
]. "if not nonswap=totalswaps"
]. "do candidate string"
```
5.0 CONCLUSION

Locational analysis software has used spatial objects in computing for several decades. We have developed an approach that formalizes the specification and use of spatial and analytical objects in an object oriented programming environment. The defined objects communicate by message passing through their public interfaces and create a set of new analytical objects which can be manipulated in the course of solving locational problems. The Smalltalk code developed to test the approach is modular and compact. In addition, the object oriented environment appears to be well suited for the development of prototype systems and may form the basis for future work involving the development of a model management system for spatial decision support.

6.0 REFERENCES


