AGGREGATION AND OTHER PATTERNS OF GRAPHS IN SPACE

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy in Computer Science
in the
University of Canterbury
by
Malathy Naguleswaran

University of Canterbury
2000

Abstract

Space as perceived by humans may be described by 2-D space, 3-D space, or 4-D space, which includes the temporal dimension. Within the space that is most commonly used, the urban space, there exists an even more specialised space, the graph space. This work develops techniques for the representation of these graphs in space. The representation of geographic space needs to satisfy some recurring patterns of requirements, such as capturing topology, multiple representation and spatial indexing for fast access. These three requirements for representing the graph space have been addressed in this work.

The main contribution of this work is in the multiple representation of the graph space, which is addressed as a graph aggregation problem. This graph aggregation is based on some ordered values associated with the edges of the graph and can be defined without any other information except this edge attribute. This permits a very simple means of achieving graph aggregation and also facilitates a simple user interface for defining it. Graph aggregation is applied to transportation networks to improve any shortest path algorithm. Application of the proposal in spatial interaction modelling is also suggested. The proposal in this work is a specific interpretation of graph aggregation and is compared with similar schemes.

Capture of the topology of spatial objects in 2-D space has been investigated extensively by researchers. A simplified model for capturing the topology of extensible graph space has been developed. An Rtree based indexing scheme is also developed for indexing this extensible graph space, which can be used to implement an interactive browsable view. Two user interface software patterns for developing such browsable 2-D views are also presented.

Acknowledgements

The help received from the following people is gratefully acknowledged.

Assoc. Prof Wolfgang Kreutzer for supervising this work from its initial stages to the final writing up with suggestions and constructive criticism.

Prof. Pip Forer for assistance in familiarizing the author with some of the problems addressed in this work.

Dr Jochen Albrecht for reading sections of the thesis on graph aggregation and suggesting other related work.

Prof Tad Takaoka for his comments on graph aggregation.

Jane McKenzie for her patient and valuable help in improving the style of writing of this work.

Žarko Andič for providing the transport network data while he was working at Land Transport Safety Authority of New Zealand.

Contents

Chapter 1

Introduction

1.1 Introduction

Representation of the geographic space in Geographic Information Systems (GIS) has been evolving over the past few decades. As GIS technology matures it has been applied to solve a greater variety of problems. Since its early use in cartography, which is the management of maps in computers, GIS continues to evolve, and is used in wide ranging applications such as decision making during urban planning, to management of the environment. This growth in its applicability necessitates greater flexibility in managing the geographic data.

Geographic data describes space. Space as perceived by humans may in turn be described by 2-D space, 3-D space, or 4-D space, which includes the temporal dimension [Peu88]. The representation of space becomes progressively more difficult as its dimensionality increases. To date the greatest advances have been in the representation of 2-D space [Güt94b]. This is not surprising since most human activities occur on the Earth's surface, and therefore most GIS applications deal with 2-D space.

Within the space that is most commonly used by humans, the urban space, there exists an even more specialised space, the *graph space*. The urban environment consists of a dense collection of functioning systems. Underlying all these systems

and their associated tasks is the urban geographic space. Networks are a dominant theme in the urban geographic space [LT92]. A network is the basic structure for all facility management systems, such as electricity, telephone, gas and water. Population is dispersed over the street network, and the movement of population occurs over the transportation networks. These different kinds of networks that are found embedded in the urban geographic space represent many distinct processes and movements. *This work develops techniques for the representation of these graphs in space.*

Management of geographic data needs to represent spatial attributes and answer queries about the spatial relationship between objects with spatial attributes. An efficient means of achieving this is to capture some of these spatial relationships into the representation. This is the capture of topology. Within GIS research this approach has been developed in detail to represent 2-D space. Geographic data is also inherently large; therefore one of the efforts in GIS research is to develop improved methods for managing large volumes of data. Two threads of development within GIS have addressed this need. The first thread has developed indexing data structures such as quad-trees for fast access. Several such data structures designed to index different types of spatial data exist [Sam89]. The second thread addresses the concept of multiple representation of the geographic data as a means of reducing complexity and size of the data to be handled for a specific application [MW95]. Thus 2-D space has been the focus for evolving a model with: efficient spatial indexing, adequate capture of topology and the ability to represent multiple views. *The contribution of this work is to derive specialised techniques in these three areas for graphs in space.*

The main contribution of this work is a graph aggregation scheme. This is based on defining graph aggregation using some edge attribute. No additional information is required other than this edge attribute. This permits a very simple means of achieving graph aggregation. It also facilitates a simple user interface for defining it. Graph aggregation is applied to transportation networks to improve shortest path algorithms. It may also be used to aggregate population networks in spatial interaction modelling. A graph aggregation scheme has been used in the Artificial Intelligence (AI) field. It has also been used in GIS to build graph models into

databases. The proposed scheme is compared with these existing schemes.

Graphs in space are represented as a set of intersecting points, and polylines between these points. Representations of 2-D space need to include the area attribute in addition to points and polylines. The topological modelling of 2-D space can be specialised for graphs in space. As would be expected, this reduced dimensionality results in a simplified scheme. A recurring requirement in this graph space of reduced dimension is to access all points along the edges of the graph. Representing this graph space of reduced dimension is discussed in this work and a model is proposed. A new spatial indexing scheme for graphs in space is also suggested and this scheme was used in the implementation of the user interface to display and browse graphs in space.

Software reuse is achieved when fundamental recurring patterns are detected and encapsulated in a suitable form [Joh88]. Some well known reusable patterns are those in the graphical user interface design. Concepts such as pull down menus, radio buttons and windows are known to all computer users today. The software pattern is an emerging concept that aims to capture expert knowledge in software design. Two dominant patterns in GIS have already been mentioned above, namely the capture of topology and multiple representation. We have applied these to the reduced dimension of graphs in space. Other patterns in GIS are also discussed throughout the thesis.

1.2 Graph Aggregation

Handling very large volumes of geographic data at multiple levels of detail improves performance. Graph aggregation provides such multiple views of graphs. Graph aggregation is useful for hiding details in the graph to be viewed, and executing graph algorithms over a graph of reduced size, resulting in time efficiency. Graph aggregation as proposed in this work generalises a given graph to produce a graph of reduced size. A node in a graph, in turn, can be another graph (Figure 1.1). In this case the aggregated node and the remainder of the graph are subgraphs of the original graph. The proposed graph aggregation scheme is an extension of this

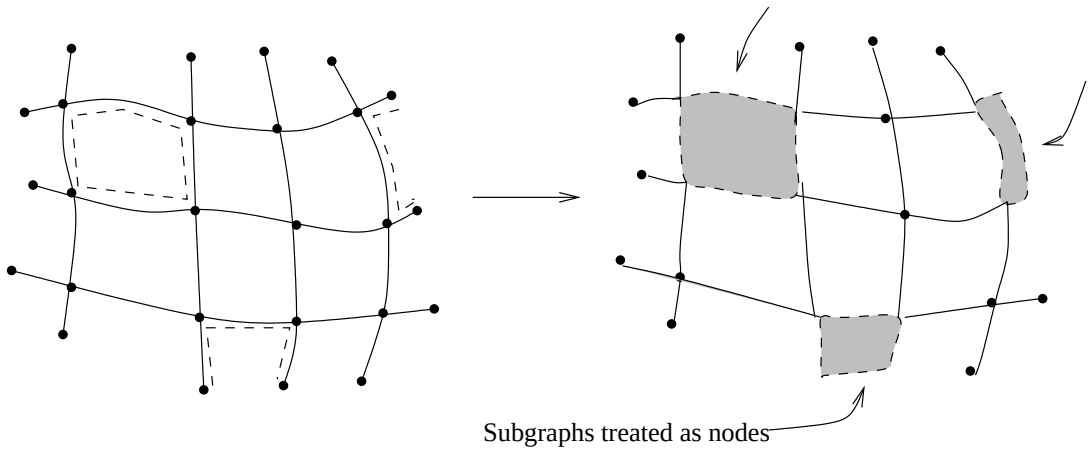


Figure 1.1: Graph Aggregation

concept.

Two distinct views can be taken of the multi-scaled view. It may be built into the representation at model building time, or it may be generated at run time through some aggregation process. The proposed scheme builds graph aggregation dynamically. Mainguenaud [Mai95] also suggests a multi-scaled graph model. Our technique and the one suggested in [Mai95] differ in two fundamental aspects. Firstly, our technique can be applied at any stage to a given graph. Therefore it can produce several different aggregations over the same graph. Secondly, the representation of our scheme is very simple. *This simplicity is achieved by making an assumption about the connectedness property of the aggregated part of the graph.*

The proposed graph aggregation is referred to as *graph nodeflex*. A simple means of defining nodeflex is derived, based on an edge attribute of a graph. *We demonstrate that any edge attribute that represents a set of ordered values can be translated into a nodeflex.*

Graph nodeflex can be applied to several problems. The use of graph nodeflex is demonstrated over a transport graph. The edges of the transport graph may be assigned some scalar quantity based on its importance in the transportation. The transport graph can be aggregated based on this value. This aggregated graph is then used to improve shortest path algorithms. Any shortest path algorithm will execute more efficiently based on this approach.

Another application of graph nodeflex is in aggregating population data. An aerial view of an urban settlement will reveal that it can be viewed as separate pockets of dense population connected by roads. When the scale of the population density is varied, different sets of population pockets will result. This type of aggregation can be useful in applications that need to model population distribution and movement.

1.3 Topology of Graphs in Space

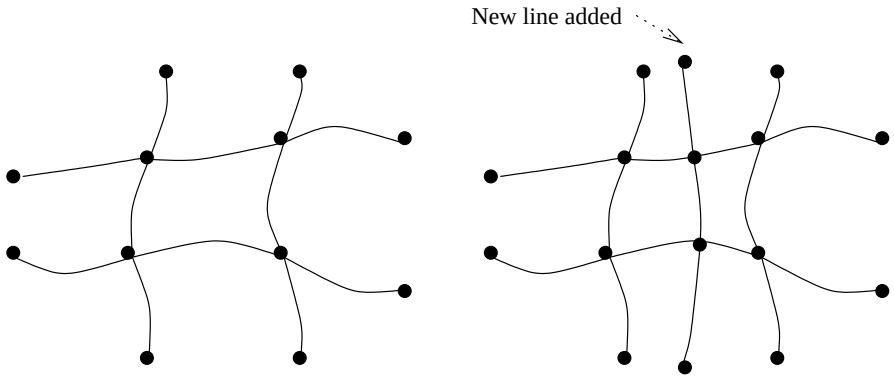


Figure 1.2: Graph Topology

The most commonly represented geographic space is the two dimensional (2-D) space. Point (0-cell), polyline (1-cell) and polygon (2-cell) are assumed to be the primitive spatial elements of 2-D space [Mar90]. More complex spatial objects are built using these three primitive elements. Developments in representing this space have concentrated on the complete representation of 2-D space. Specialising the representation of 2-D space for graphs in space has several benefits. These are: the simplified model permits more efficient implementation, efficient indexing schemes specific to graphs in space can be applied, and graph aggregation, as mentioned above, can also be applied.

The graph in Figure 1.2(b) is a new graph resulting from the addition of another line to the graph in Figure 1.2(a). One approach to sharing the lines among different, but overlapping, graphs in space is to overlay all the lines involved. This will result in a new planar graph. Other planar graphs can be defined with respect to this fully overlaid graph. This approach is the same as the topological model [EFJ89] for

representing 2-D space. The topological model extends it to the entire 2-D space and thus considers point, line and triangle. When the topological model is concerned with points and lines only it captures the graph topology. A hybrid topological model for graphs is developed, exploiting this reduced dimension, and resulting in a simplified model.

This scheme can be used to study several developmental scenarios during planning for the addition of new roads and residential areas.

1.4 Indexing Graphs in Space

There is an extensive collection of indexing data structures for spatial data that affords efficient, selective accessing. Many of them focus on representing 2-D space. Some are specifically designed to represent points sets and line sets in 2-D space. Point quad-tree for points and PM quad-tree for lines are two examples. Rtree is another scheme which is an extension of the Btree indexing scheme [Sed88]. Rtree uses the bounding rectangle as the key of an object being indexed.

A variation of the Rtree scheme is used in this work to index the polyline set that represents a graph in space. Its use is demonstrated in the implementation of a user interface to display and interact with graphs in space.

1.5 Thesis Outline

The thesis surveys some generic aspects, namely spatial data models for GIS and graph aggregation, and concepts very specific to graphs in space, namely, the topology of graphs in space and its spatial indexing. These are presented in five broad sections in the following order: GIS and computer science, graph concepts, graph aggregation, topology of graphs in space, and spatial indexing.

GIS and Computer Science: Chapters 2 and 3 survey the background to this work in GIS and computer science. Chapter 2 discusses the intersection of GIS and computer science. Database technology applied to the capture of the topology of 2-D

space, object orientation applied to GIS, and software patterns and its relevance to GIS, are all discussed. Chapter 3 pays more attention to the subject of generalisation in GIS.

Graph Concepts for GIS: Chapter 4 covers wide ranging topics related to graphs. The field of graph theory develops several concepts about graphs, based on a graph's properties. This field is distinct from the study of representation and operations on graphs in computer science. Several issues in relation to graph representation and operations are discussed in this chapter. These are: *graph representation in databases, data structures for graphs, some graph associations, and graph algorithm specification.*

Within relational technology there are no specialised mechanisms for representing explicit graphs. Thus graphs are not stored for efficient graph traversal, and maintaining paths over such graphs is not possible. Improvements on the relational technology for modelling graphs is one generic graph issue that is addressed by recent research developments. This is noted in this chapter.

Many data structures are used to represent graphs, e.g. adjacency matrices. Other graph data structures, and their advantages and disadvantages are also discussed in this chapter. An optimal data structure to represent explicit graphs is selected.

When an underlying base graph is assumed, subgraphs of this base graph can represent different features. *Graph containment, graph overlap, graph adjacency, graph partition and graph coverage* have straightforward semantics with respect to the base graph and its subgraphs. These are defined in this chapter.

Another area related to graphs with application to computing is the specification of graph algorithms [MS90], [Erw93]. Developments here can be used for providing generic programming interfaces to specify algorithms over graphs. A specific application is in specifying queries over graphs in databases [RHDM86]. A style of algorithm specification is described in this chapter and used in this thesis.

Graph Aggregation: The graph aggregation concept is developed in Chapters 5 and 6. Chapter 5 introduces the basic idea behind graph hierarchy, based on node-flex. Aggregation can be repeated to produce a hierarchical view of the original

graph. Several such independent hierarchical views can be generated for the same graph. Aggregation of edges is also possible, with some restrictions.

Another multi-scaled view of graphs proposed in the literature is also described in Chapter 5 and contrasted with the proposed approach. This existing approach builds the multi-scaled view into the representation at model building time.

Chapter 6 develops a simple representation scheme for nodeflex. The simplification is based on an assumption that the subgraph representing an aggregated part is *connected* (Section 4.1). This is a valid assumption in most practical cases where this type of aggregation is useful.

In Chapter 7 some basic algorithms related to graph nodeflex are presented. Most importantly the algorithm to derive the aggregation data structure based on an edge attribute which has an implied order is presented. It also develops an application of nodeflex in improving shortest path algorithms for transportation graphs.

Graph Topology: Chapter 8 outlines a method for maintaining graph topology under updates. A brief outline of the topological model for point and line is given and contrasted with the proposed method. A common requirement of models of graphs in space is access to all points along the edges. This is required when information needs to be recorded about these locations. Accidents, traffic signs and address locations are some examples of locating data along the edges. This pattern is also discussed.

Graph Display and Interaction: The graph nodeflex was tested using custom built software. This uses a spatial indexing scheme based on Rtree for graphs in space. This scheme is used for interaction with the displayed graph. This is described in Chapter 9. This chapter also describes further user interface software patterns for display and interaction of 2-D maps of geographic data.

Chapter 2

GIS and Computer Science

What is GIS? This question has received a lot of attention from Geographers. Geographic Information System (GISystem) refers to, "a computer based information system that enables: capture, modelling, manipulation, retrieval, analysis and presentation, of geographically referenced data [Wor95]." Goodchild coined the phrase Geographic Information Science (GIScience)[Goo92a], to refer to the geographic concepts required for building GISystem. The debate reported in [WGP97], concludes that the practitioners and researchers of GIS regard the activity of "doing GIS" to mean a continuum from "using GIS the tool", to "making GIS the tool", to "research in the GIScience".

The tool making aspect of GIS exploits many developments within computer science. Developments in database technology, data structures and algorithms, software engineering, computer graphics have all contributed to GIS tool making. Some of these contributions are described in this chapter. These are spatial data representation, object orientation, and the concept of software pattern.

2.1 Representing Spatial Data

Space is a difficult concept[CG86]. Its digital representations are difficult resolve within a uniform framework [Peu88], [Cou92]. One established dichotomy of spatial

representation is the raster/vector models of representing space. The raster model (field view) treats space as a continuum and the vector model(object view) defines geometric objects in the Euclidean space. Couclelis [Cou92] suggests that this dichotomy between a field view and an object view represents the empirical issue of how the geographic world is categorised and acted upon.

In the 'space as a continuum view', values are assigned to each point in space and are used to describe its properties. Examples of this are: soil quality, and wild life distribution. In computer representation the space is discretised into cells and values are assigned to these cells. This takes up a large storage space. Computer science has developed hierarchical data structures, such as quad trees, to handle spatial data. These data structures permit the amalgamation of adjacent cells into single units, thus reducing the storage requirement. Quad tree structures also enhance the accessing of stored space, based on spatial keys. Therefore, unlike compression schemes, which are used to store regular images, the data structures used to represent raster spatial data also index the space to allow selective access.

In the 'objects in space view' objects are represented using points, lines and polygons. The management of this data presents several challenging problems for the computer scientist.

The main requirements of a vector based spatial data model are:

1. Capturing or deriving the topological relationships among geometric objects from which we can analyse spatial data and answer queries.

2. Query languages that incorporate spatial aspects.

3. Efficient access to objects based on the spatial range.

4. Spatial data with multiple representations, so that the correct levels of detail can be used for the problem at hand.

The most notable successes of computer science's contribution to "GIS tool making" have been in some of the above areas.

Firstly, the development of spatial data structures, such as Quad tree, Rtree and other related hierarchical data structures has contributed to the efficient management of spatial data. These permit geometric objects to be indexed based on spatial keys to facilitate fast access [Sam89]. Spatially indexed data structures continue to be improved by researchers. In Chapter 9, a modified Rtree indexing scheme is described and is used in the implementation of a user interface for the display and interaction of graphs in space.

Secondly, geometric algorithms, such as point in polygon algorithms and polygon overlay algorithms [PS85], have been developed for deriving topological relationships from a model; a computation intensive task.

Thirdly database technology has been the backbone of GISystem, [Mol98], [Wor95], and [LT92]. GISystem databases were first implemented using Relational Database Management System (RDBMS). RDBMS together with the associated query language, SQL, is able to manage a limited set of entities that can be represented as rows in a table. Its adoption for GIS, however, has limitations.

2.1.1 RDBMS and Spatial Data

Traditional RDBMSs typically do not provide the functions needed for handling raster and vector based spatial data. These functions include: incorporation of new data types, operations on these new data types, extension to query languages to handle the new data types, and new access structures for efficient spatial range searches.

The mature technology behind RDBMS, however, makes it an attractive option to implement GIS and it has been employed with some success in both commercial and experimental GISs [HC91]. ARCINFO, a commercial GIS [ESR90], implements a vector model directly in an RDBMS and provides additional custom built operations to handle the data types. It thus provides the three primitive data types for handling spatial data: point, polyline, and polygon, but new data types cannot be defined. Spatial objects are stored as traditional RDBMS tables, allowing direct SQL queries. However, these must explicitly state the relational join operations needed to retrieve

the desired data. Spatial relationships, such as *touch, overlap*, etc. (Section 4.5.1) cannot be directly used to query data. The SmallWorld GIS system [Sma92] provides an object oriented GIS front end, but uses an RDBMS to store the objects in the user's object model as tables in the RDBMS. A number of experimental systems, e.g. GRAL [Güt89] and POSTGRES [SRH90], also adopt an RDBMS for handling spatial data.

2.1.2 Capturing Topology for Vector Based Spatial Data

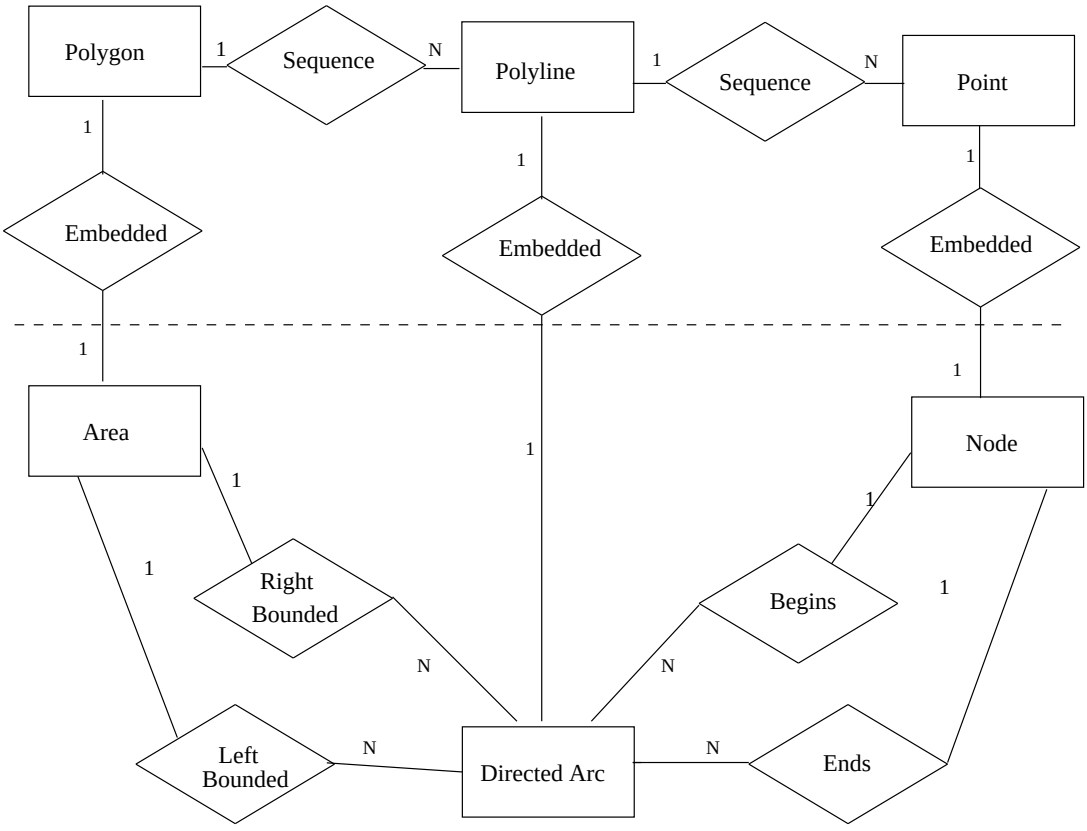


Figure 2.1: Vector Based Data Model

A standard Entity Relationship model of vector based data is shown in Figure 2.1 [Wor95]. The lower half of the figure records the topological relationships among the spatial entities, node, directed arc and area. The top half of the figure records the geometry of the vector based model. The two halves are connected by one-to-one correspondence between point–node (0-cell), polyline–directed arc (1-cell) and

polygon–area entities (2-cell).

Placing progressive constraints on the relationship between 0-cell and 1-cell objects in a vector based model results in different cell-topologies. These in turn capture an increasing level of topological relationships among the 0, 1, and 2-cells.

The first constraint is that no two 1-cells are permitted to intersect. This constraint ensures that all intersections of 1-cells are already recorded as part of the model. Thus queries that require the derivation of intersections can be evaluated by querying for common nodes. This is the approach developed for the TIGER file system [Mar90] and later employed in many GIS databases.

The second constraint is that every 1-cell has at most one 2-cell defined to the left and one 2-cell to the right. This ensures that, for a given set of 1-cells that intersects only at its end points (first constraint above), the 2-cells defined in the model are the smallest possible polygons. Larger polygons are then defined as an aggregation of these smallest 2-cells. This approach eliminates the need for polygon overlay for answering queries.

The third constraint is that every 0-cell in the model is part of a 1-cell; in other words there are no isolated 0-cells. The largest homogeneous cell concept suggested in [BK91] is based on these three constraints. The combination of these constraints results in a *complete topological representation* [BK91].

The fourth constraint ensures that the 1-cells in the model are straight lines and the 2-cells are triangles. All polylines in the system are then defined as a series of straight line 1-cells and all polygons are defined as a collection of triangle 2-cells. This is based on simplicial complexes [EFJ89].

A vector based data model could maintain the above mentioned constraints at the first, second, third, or fourth level. As new points or lines are added to the model, the model must be modified to maintain the constraints. Existing geometric elements may be deleted and new elements created. Spatial objects whose attributes are defined with respect to old geometric elements that are deleted must be modified appropriately. Therefore maintaining a vector based model that satisfies the constraints under updates is a non-trivial task. This is discussed further in Chapters 8

and 9. In Chapter 8, maintaining graph topology under updates is developed and its implementation is described in Chapter 9.

2.1.3 Summary

Spatial queries over a data model maintained with the above mentioned constraints and resulting cell topology can be evaluated readily using matching keys, a standard database approach. However, it is not always practical to maintain these constraints. If the database includes geometric elements forming widely different spatial phenomena, fragmentation occurs as too many cells are created. On the other hand, spatial queries over databases that do not maintain these constraints cannot be formulated using matching keys because such queries will require geometric algorithms to be executed. The optimal approach is to use both techniques, thus enforcing the above constraints on spatial object sets that are frequently treated together, while separating others that are not frequently treated together into different groups.

A generic spatial database management system able to handle raster based and vector based data has not yet materialised. The promise of object orientation in software development and database technology has further infused the GIS field with new developments and delayed the emergence of standards in GIS databases. The requirements of multiple representation in GIS also complicates the matter.

2.2 Object Orientation and GIS Tool Making

Object orientation initially evolved as a programming language paradigm. Linguists have theorised that language influences a speaker's thought processes, and that language, in turn, develops to match the speakers' view of the world. Thus both evolve as a result of one influencing the other. This can be equally applied to programming languages. A programming language influences the design, and the thought process involved in a design influences the further development of programming languages. When designing, people think in terms of objects and their interactions.

Object orientation therefore matches the thought processes of a designer much better than more traditional program structuring styles. The use of the object oriented paradigm has also been extended to database technology and other areas of software engineering such as requirement analysis.

Object orientation permits user defined types in a programming language and database. Thus it is a promising paradigm for representing spatial data. For example, vector based spatial data are typically used to represent discrete objects in space. Application dependent objects such as 'house' or 'corn field' can be created as user defined types. The inheritance concept of object orientation can then be used to unify related objects. Thus 'house', 'school' and 'office' are all related to 'building' through inheritance. The standard interface between a database and a programming language is the query language defined over the database. However, a query language that recognises user defined types has not become widely accepted; a drawback to increasing the popularity of object oriented GIS.

Commercial object oriented GISs, such as the SmallWorld system [Sma92] and LaserScan [Las94] have mainly been successful in handling vector based spatial data. The SmallWorld system is widely used in network based applications in facilities management. These systems provide an integrated programming and database environment, and use proprietary query languages. However, with the independent development of systems, inter-operability remains difficult.

2.3 Patterns in GIS Tool Making

The use of high level programming languages and faster computers makes it easy to build large software. However, it is still a resource and human intensive task, requiring a team of people working closely together. Gabriel [Gab96] likens building large software to an attempt at building a 'flying-buttress construction cathedral'. Inevitably several prototypes will collapse before the team figures out how to build one. Brooks [Bro87] separates developments in software engineering into "essence and accidents" and attributes most successes to date to the "accidents". Coming to terms with the "essence" remains difficult.

Patterns are an evolving concept in software engineering and it is envisaged that their wider use will lead to improved software development environments. Building software has much in common with other design disciplines such as engineering and architecture. The design process has been explored more thoroughly in the writings of some architects, particularly Christopher Alexander, who challenged the traditional mould of architectural design which separates architect and builder [AIS77]. Similarly, software designers study requirements for software and pass the design on to programmers for coding. Alexander proposed a user centred approach to architecture in which the inhabitants design their own buildings and the architect-builder would blend the activities of the design and construction. He proposed *patterns* as a way of accomplishing this. The pattern states a problem, the constraints placed on its solution, and a method for solving it. The example below demonstrates the concept and the style of recording of a pattern.

Consider the choice a worker should make about the mode of transport to his or her work place.

Problem: Should one use public or private transport to undertake a particular trip?

Forces: If the weather is pleasant, a few minutes walk to and from the bus stop could be good for one's health.

If the shortest possible time is to be spent in travelling, then private transport may be a better choice, although only if a parking space is readily available at the destination and traffic congestion is low.

Will there be any further trips once the destination has been reached? In this case the availability of private transport may be an advantage.

Removal of the private transport at the source point could disadvantage others who may also need the same vehicle.

Solution: Choose public transport unless mitigated by the other forces.

Rationale: This is beneficial for the traveller's health and the environment. It will also eliminate traffic congestion.

Patterns can address problems at varying levels of granularity. A collection of patterns documents the problems and solutions in a particular application domain. This collection is then used to solve a larger and more complex concrete problem in that domain. Alexander generated over 200 building design patterns. In spite of this, his notion of user centred design is still a distant vision.

The software community has borrowed this concept to develop a collection of common software design patterns [GHJV95], [Pre95]. A software pattern describes an abstraction used by experts, and documentation is similar to the example above. Although software patterns are most commonly used in object oriented programming, they can, in principle, be used to design software in any style.

Object oriented languages offer the class/subclass construct, where subclasses inherit the properties of their parents. This pattern can be used to directly represent the generalisation–specialisation abstractions that frequently occur in software. Two other patterns of large granularity that recur in software designs are the whole–part association and the one-object–many-roles concept. These have no equivalent constructs in existing languages or database management systems. The whole–part association pattern is related to the concepts of aggregation and multiple representation. This pattern is the main theme of this work. These, and other patterns, will be cited throughout the remainder of this thesis, with reference to GIS. In Chapter 9, two user interface patterns are described for display and interaction with 2-D maps. These two patterns were used in the implementation to test the ideas presented in this work.

2.3.1 The Roles Pattern

The *Role* pattern documents the one-object-many-role concept. This is useful when the an object takes different roles. By implementing the roles separately from the object these can evolve independently. In general the same role may be taken by objects from different classes.

The geographic space represented by GIS holds a rich variety of information. For example, it may be used to represent population density, land classification, and

voting districts. Thus the same area takes on different roles when representing different data. The layers concept, as implemented in many GIS packages, is an example of this multiple roles. However, each layer is implemented as an independent entity and the idea of *Role* is not used. A *Layer* pattern can be viewed as a special case of the *Role* pattern applied to geographic space. This is described in more detail in [GBdN97].

Another example of the *Role* pattern occurs in transportation systems [ZF97]. A point in a transportation system may represent a junction, a stop point, a parking point, etc.

2.3.2 The Proxy Pattern

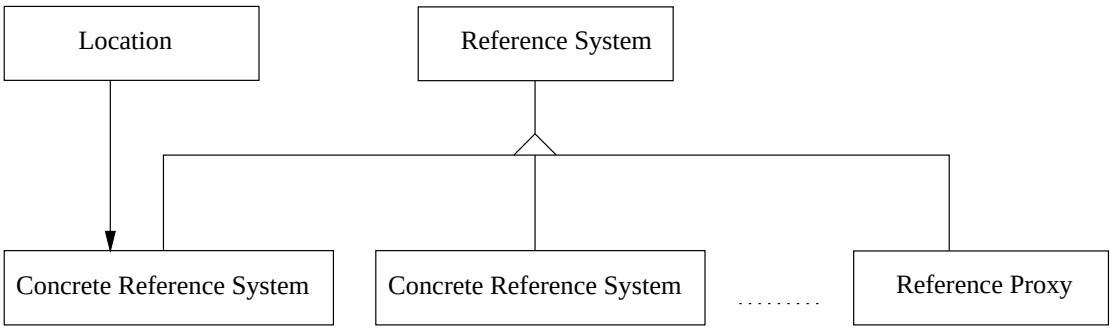


Figure 2.2: Reference-Proxy Pattern

The intent of a *Proxy* pattern is to provide a place-holder that controls access to another object or subsystem. Essentially the *Proxy* pattern consists of two objects of the same type, where one is the gate-keeper and the other a concrete object to which access must be controlled. Several different types of proxies have been recorded. A *Remote-proxy* provides a local representative for an object somewhere else. A *Virtual proxy* creates expensive objects on demand. A *Protection-proxy* mediates access rights.

Another example of the proxy pattern is the Reference proxy. Location on the Earth's surface is represented using a variety of reference systems. Therefore, a location object must know the particular reference system on which its measure is based. For example, some reference systems use elliptic coordinates: two angular

measures and one scalar. The angles represent longitude and latitude, and the scalar measure represents altitude. The reference system must be taken into account when calculating other measures, e.g. distance. To achieve this a *Reference-proxy* can be defined. The Reference-proxy will mediate among the different concrete reference systems when calculating measures that depend on locations based on different concrete reference systems. Figure 2.2 shows this.

2.3.3 The Chainage Pattern

In some spatial applications graphs need to be treated as continuous rather than consisting of node and edge objects. To record relevant data an infinite number of points along the edges of the graph need to be defined. For example, the road in Figure 2.3 shows themes describing its properties, which need to be recorded. The approach used is to specify the points in the form $< route, offset >$.

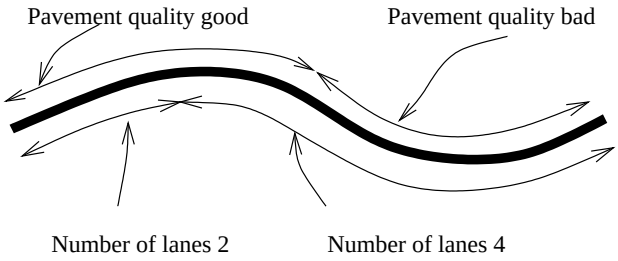


Figure 2.3: Overlapping themes over a road

Route

A *Route* pattern in transportation systems is described in [ZF97]. This pattern addresses the problem of defining routes in a transportation network in several ways. A route over a graph can be defined at three levels: the 1-cell representing a graph edge, a path representing an ordered set of edges, and routes. Routes are a specialisation of the path concept (Figure 2.4a). In general, it should be possible to define a route whose start and end points may not coincide with the nodes of the graph.

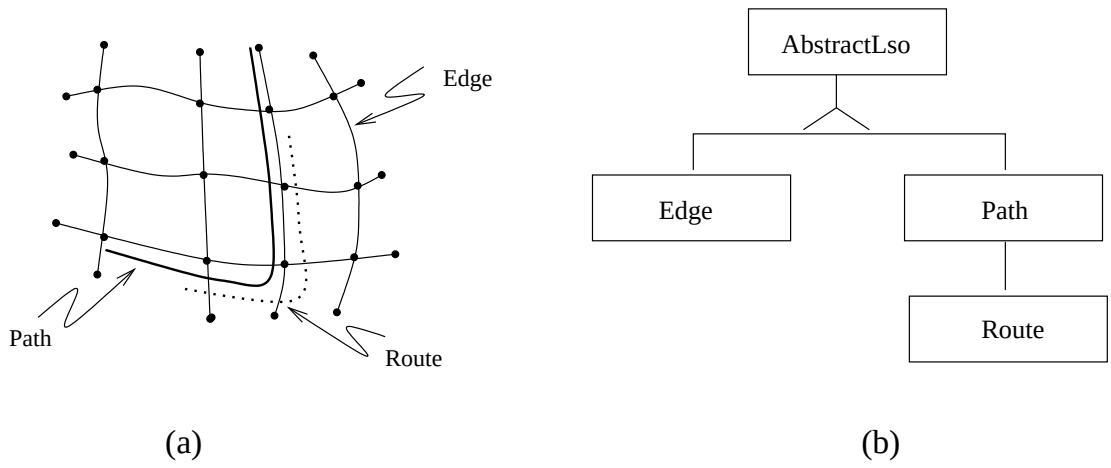


Figure 2.4: A Route

1-D Addressing Systems

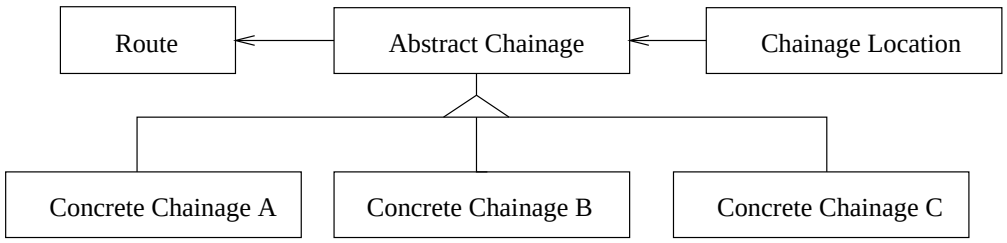
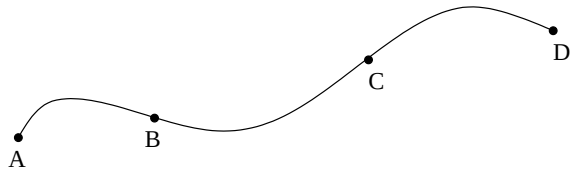


Figure 2.5: Chainage Pattern

Given a route defined as above, points may be specified on it. The $< route, offset >$ method of specifying points mentioned above is referred to as *dynamic segmentation* [DV92, ESR92]. Dynamic segmentation is just a special case of the many independent 1-D addressing systems that are used. Mile-post measure [DV92] is used to locate transportation data such as traffic signs and road markings. Street address location is another example of 1-D addressing. These 1-D addressing systems are modified versions of $< route, offset >$ to define points along the route. In these systems visible objects along a route with some marked values are used as anchor points. Locations between marked values are determined by interpolation using the geometric distance of the route and the marked values.

The *Chainage* pattern represents any 1-D addressing system that takes a route defined by some means and applies 1-D addressing to locate points along the route.

Figure 2.5 shows this pattern. A `Chainage` object knows how to interpolate intermediate locations based on the defined locations and a given route. An example `Address-Chainage` object for a route is shown in Figure 2.6. It consists of the route and a table that defines locations along the route with corresponding address values. A given address, say within the range 55–150, will be located by interpolating over the route using the offsets for address 55 and 150.



Offset	Address
0 (A)	55
650 (B)	100
2500 (C)	101
3000 (D)	150

Route Table

Figure 2.6: Address Chainage Example

2.4 Summary

GIScience is the science behind the technology used to make GISystems. It is a multi-disciplinary field to which computer science has made major contributions, such as database management systems and associated query languages, data structures for spatial indexing, multiple representation models, object orientation, and software patterns. This chapter described the topological model of 2-D space as a basis for its representation in DBMSs, object orientation, and software patterns. The following chapter will discuss multiple representation models.

Chapter 3

Generalisation Hierarchies in GIS

Hierarchy is a partial ordering of a set of components, in other words the components can be placed in a tree structure. Many complex systems can be decomposed into a hierarchy of components. For example the earth can viewed as consisting of land and sea. Both land are sea can be further decomposed into regions and so on. Many of the elegant hierarchies observed in complex systems are only approximations. Thus there will be a faint presence of the components of the lower levels of the hierarchy in the components of the higher levels of the hierarchy. Simon observes this as follows: complex systems are nearly decomposable and not completely decomposable [Sim73].

In attempting to understand complex systems, science and society has employed two main strategies [AA96]. The first strategy reduces the complexity by gaining tight control over the system's behaviour. This is the mechanical style as in setting up mathematical equations to solve a complex problem. The second strategy includes the observer as well as the observed in analysing complex systems. Hierarchy theory falls in this second category.

In building a hierarchy to represent a complex system, the observer determines what is represented at each level. This in turn is determined by the grain and extent of the observations made by the observer. [AA96] compares grain and extend to a fishing net metaphor. Grain is the hole size of the net which determines the smallest size of the fish that will be caught. Some fish such as whales will be too large to be caught in the net. This is compared to the extent. The levels in a hierarchy are thus related

to the grain and extent of the observations.

In a hierarchy the upper level components offer context and therefore constraint lower level components. Hierarchies can be nested or non-nested. In a nested hierarchy, in addition to offering context, the upper level components contain the lower level components. Nested hierarchies are more readily distinguishable than non-nested hierarchies [AA96].

Within GISystems scale dependent views of spatial data can be organised as a hierarchy and this concept is loosely referred to as 'generalisation'. A host of processes are used within GIS to reduce the size and complexity of a real world representation. Muller and Weibel [MW95] refers to two types of generalisation as cartographic generalisation and model oriented generalisation. The internal representation (model) should be distinguished from visual representation (cartographic). Cartographic generalisation is aimed at display efficiency and improved legibility. In contrast, model generalisation generalises 2-D space by manipulating the underlying database.

In this Chapter some hierarchies in GISytems are discussed and the main contribution of this thesis is also outlined.

3.1 Cartographic Generalisation

Cartographic generalisation is a computation intensive process used for transforming some graphical representation of spatial information to improve legibility. Fully automated, as well as human assisted cartographic generalisations have been attempted by researchers.

By zooming 'in' and 'out', scale changing can be achieved within a limited range during display of a geographic space stored at a single scale. More complex methods are needed to effect scale changes over a greater range.

Three types of parallel hierarchies can be observed during the cartographic generalisation process of geographic phenomenon [Tim98], [DH97]: aggregation hierarchy, filter hierarchy, and abstraction hierarchy. Aggregation hierarchy takes several concrete objects and produces a fewer number of concrete objects. In an aggregation

hierarchy a higher level object contains a collection of lower level object. It is thus a nested hierarchy. Filter hierarchy eliminates some objects from the lower levels and therefore it is a non nested hierarchy. This occurs when the scale is increased and very small objects disappear. Abstraction hierarchy is the same as the inheritance concept of the object oriented paradigm. For example, houses, schools, hospitals, etc. at a detailed level may be represented as buildings in a less detailed level. Building is an abstraction generalisation of houses, schools, hospitals, etc. Similarly farm land is an abstraction of corn field, wheat field, vegetable field, etc.

During cartographic display, as the display scale is decreased so that the level of detail is increased, the following changes may occur to the display of an object [Tim97]: a simple scale change, a change of the symbol that depicts an object, more detailed display of the object such as a greater number of lines used to draw the boundary, a change in the dimension of the object, a symbol being replaced by a spatial representation, and the splitting of the objects into several other objects.

Some of the changes listed above may be related to the three hierarchies, namely, aggregation hierarchy, filter hierarchy, abstraction hierarchy. The last of the above changes, that is the splitting of an object into several objects, relates to an aggregation hierarchy. When symbol changes occur, it reflects an abstraction hierarchy. A more detailed drawing either by using greater number of lines or by changing the dimension of the displayed object also reflects an aggregation hierarchy and is related to the topological hierarchy described below.

3.2 Topological Hierarchy

In both cartographic and model generalisation in GIS context the resulting hierarchy is associated with scale change (grain of the hierarchy level). Each level models the phenomena at a different scale. The spatial dimension of the components may also be different at different levels. For example, a polygonal area at a detailed lower level may change to a point at a less detailed level. Thus each level will represent the spatial objects at a particular scale using some topology. The levels of the hierarchy thus contains different topologies that can be related to the topologies at other levels

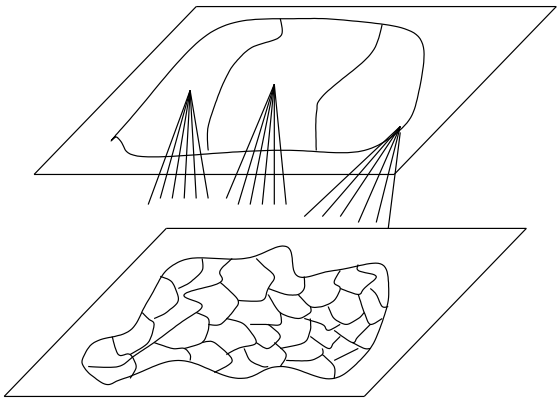


Figure 3.1: Hierarchy in Multiple Topology Representation

of the hierarchy.

Bruegger and Kuhn [BK91] analyse this and present a topological hierarchy of 2-D spatial data. This hierarchy reflects the aggregation hierarchy above.

When a spatial phenomenon is to be represented at multiple levels of detail, each level could be represented by a separate topological model of the 2-D space. If the topological model constrains its cells to be the largest homogeneous cells of Section 2.1.2, this gives rise to a one-to-many relationship between a cell at the larger scale to the cells at the more detailed level, yielding an aggregate hierarchy of cells at the various levels of representation [BK91](Figure 3.1).

For example, each object of the local government hierarchy in Figure 3.3 could be represented using a different spatial representation of the 2-D space. At the `District` level the scale used could be more detailed than the scale used at the `Country` level. These different 2-D space representations are then connected by the topological hierarchy described above.

3.3 The Composite Pattern

Hierarchies are captured in a software using a tree data structure that connects the objects at different levels using pointers. These pointers used to represent the hierarchy implies a containment of the lower level objects by the higher level object

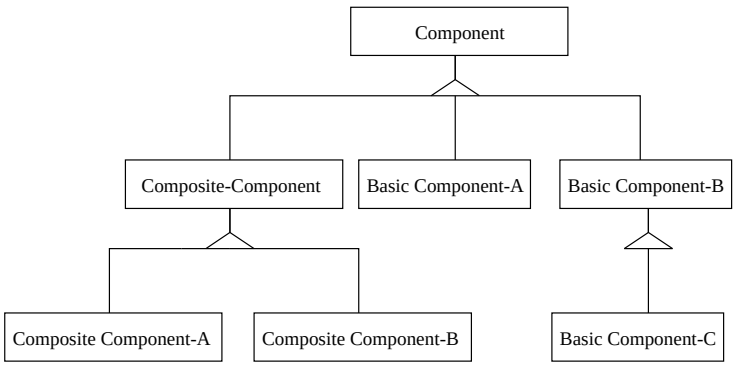


Figure 3.2: Composite Pattern Classes

in the tree, resulting in a nested hierarchy. Non-nested hierarchies in the real world are captured by assigning context related meaning to the pointers in the tree data structure.

The *Composite* pattern describes the issues involved in building a hierarchy. An abstract class defines the common properties for all the components of the hierarchy (Figure 3.2). Both `Basic-Component`s and `Composite-Component`s are subclasses of this `Component` class. The `Basic-Component`s are the lowest level objects in the hierarchy. `Composite-Component`s represent objects at higher level and they are made up of `Basic-Component`s and `Composite-Component`s. Through this mechanism the *Composite* pattern is able to export a uniform interface to its clients that use this hierarchy.

Two examples of the use of *Composite* pattern for modelling geographic entities is given below.

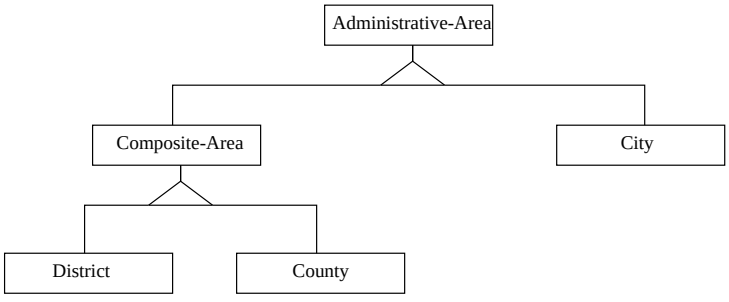


Figure 3.3: Local Government Areas Classes

The *Composite* pattern is used to implement the administrative structures of a

local government area in [GBdN97] (Figure 3.3). The `Administrative-Area` is the abstract component class. The `Composite-Area` is another abstract class that specialises the `Administrative-Area` to implement a composite component class. `City` is a `Basic-Component` class and objects of this class are the leaves of the whole–part relationship.

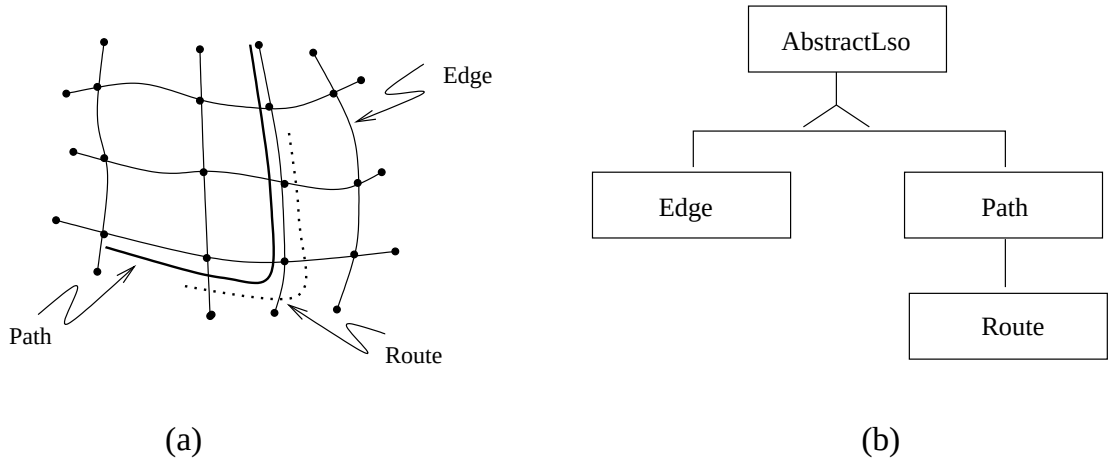


Figure 3.4: A Route

The edge, path and route in Section 2.3.3 take part in a *Composite* pattern, where the abstract component class is `AbstractLso` representing any 1-cell object (Figure 3.4(b). The `Edge` defines the leaf class for the pattern and represents the polylines defined in the model. `Path` is the composite class of the pattern and is the concatenation of several other `Edges`. `Route` is defined with respect to `Path`.

3.4 Graph Based Model Generalisation

Model oriented generalisation for the full 2-D space is too complex. To begin with they need to capture the multiple hierarchies as described in Sections 3.1 and 3.2. A simple tree data structure that connects a set of individual objects cannot represent these hierarchies simultaneously. Graph is a generic concept that connects a set of objects. Generalisation based on graphs provides an approach that is able to capture more information than a simple tree. Graph based hierarchies have been studied in relation to "GIS tool making" as well as in other fields.

Spatial aggregation based on neighbourhood graphs was first developed in Artificial Intelligence (AI) to solve certain classes of automated reasoning. Yip and Zhao [YZ96] state that it has applications in *"dynamical systems, nonlinear controllers, kinematic mechanisms and fluid motion"*. They propose a theory of spatial aggregation based on the neighbourhood graph data structure as a common interface to modularise computations. Some implemented problem solvers are redescribed in terms of spatial aggregation theory. Yip and Zhao state in [YZ96]: *"neighbourhood graphs are used to transform information rich input to abstractions well-suited for concise structural and behavioural description"*.

Within the GIS field, knowledge discovery from a large volume of spatial data has received attention from researchers. Knowledge discovery is the extraction of implicit, previously unknown, and potentially useful information in a data set [EKS97]. For example, in a geographic data set that records the pollution levels at different locations, a correlation between the pollution level and the existence of various activities such as manufacturing, traffic, etc could be extracted. The application of spatial aggregation theory for knowledge discovery within the GIS field is currently being investigated [Alb99].

Stell and Worboys [SW99] describe a graph simplification operation based on selection and aggregation of its elements. The selection omits some graph nodes and edges, while aggregation collapses the edges in a path into one with fewer edges. These reflect the filter and aggregation hierarchies respectively.

A variation to the above approach involves the aggregation of the graph elements without selection. Thus the filter hierarchy is absent in this. This is more accurately referred to as graph aggregation and is developed further in this work.

The proposed graph aggregation is referred to as *graph nodeflex*. Graph nodeflex permits aggregating a set of nodes and edges that form a *connected* subgraph into a node at the next higher level. A simple means of defining nodeflex is derived, based on an edge attribute of a graph. *We demonstrate that any edge attribute that represents a set of ordered values can be translated into a nodeflex.* The simplification resulting from this permits dynamic aggregation as well as aggregation built into the model.

In the case of graphs in space, the adjacency relation coincides with the spatial topology of the graph elements. Thus when graph based aggregation is applied to graphs in space, it will result in a specialised GIS generalisation process where a graph with large number of nodes and edges are reduced to a graph with lesser number of nodes and edges. This corresponds naturally to real world situations. This is demonstrated with two real world examples.

The use of graph nodeflex is demonstrated over a transport graph. The edges of the transport graph may be assigned some scalar quantity based on its importance in the transportation. The transport graph can be aggregated based on this value. This aggregated graph is then used to improve shortest path algorithms. Another application of graph nodeflex is in aggregating population data (Section 7.3).

3.5 Summary

Complex systems can be represented by hierarchy. Hierarchical views of the world, commonly used by humans, are also useful in its representations in GISystem. This is loosely referred to as generalisation in the GIS field. Three types of hierarchies are noted. These are: aggregation, filter, and abstraction hierarchies. An important aggregation hierarchy in GISystem is the topological hierarchy. The *Composite* software design pattern can also be related to the aggregation hierarchy.

A simple tree data structure is inadequate to represent the complexities of generalisation in GISytem. Graph based generalisation can represent more information than a simple tree data structure. Graph can be generalised using filter and aggregation hierarchies. It can also be generalised using simply an aggregation hierarchy. The latter case is equivalent to graph aggregation, which will be considered in this thesis.

The next chapter introduces some graph theoretic concepts that are used for development of a graph aggregation scheme. The theory is introduced in Chapter 5. Chapter 6 develops a representation scheme and Chapter 7 demonstrates its applications. Chapter 9 describes the implementation of the proposal and one of the applications.

Chapter 4

Some Graph Concepts

This chapter describes several concepts relating to graph representation and operations, such as representation in databases, data structures, graph relationships and a graph algorithm specification scheme.

4.1 Definitions

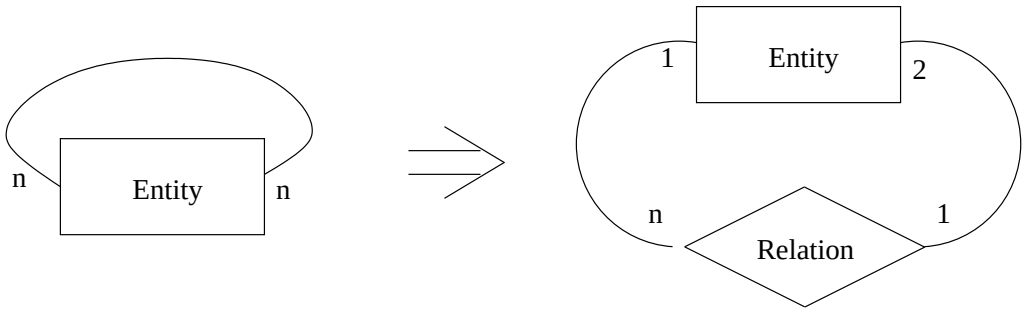


Figure 4.1: Cyclic Relation

In the most generic sense, a graph is a relation that relates a set to itself. A graph is defined through a domain of values {N}, which are interpreted as the 'nodes', and a set of 'ordered node pairs' from the domain {N} called 'edges', and denoted by {E}. Thus {E} defines a cyclic relationship over the entity represented by the domain {N}, and has two attributes from the domain {N} (Figure 4.1). This is the standard 'directed graph'.

Directed graph can be specialised to denote graphs with specific properties (Figure 4.2). These and other relevant definitions are given below. For a more exhaustive list of graph related definitions see Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness, [Meh84] or Discrete Mathematics with Graph Theory [Goo98]

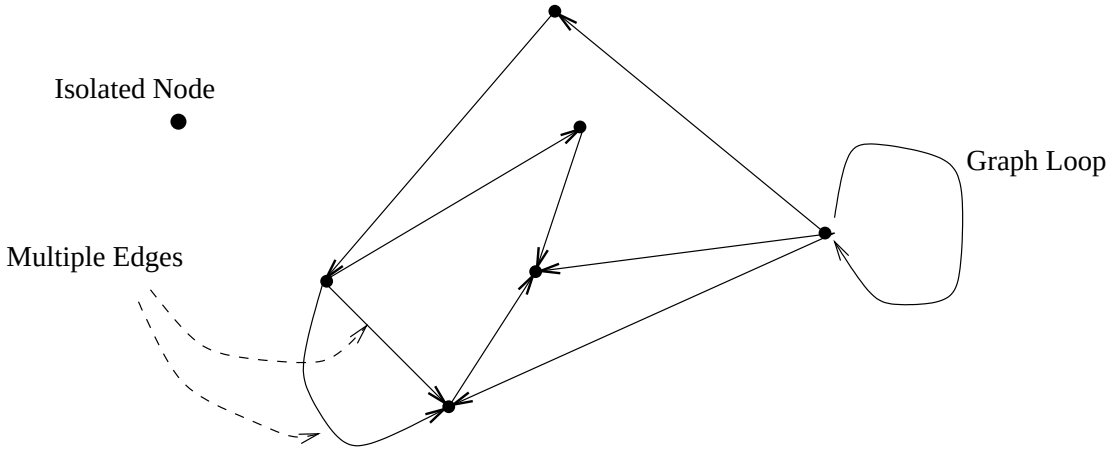


Figure 4.2: Graph Properties

When each edge in a graph can be traversed in both directions, the graph is referred to as an *undirected graph*.

When every node in {N} is a member of at least one node pair (edge) in {E}, the graph is said to have no *isolated node*s.

When an ordered node pair does not uniquely denote an edge, the graph is said to have *multiple edges*.

When the graph contains node pairs of the form (n, n), it is said to have *graph loops*.

Graph element is a generic term that refers to both nodes and edges.

A *path* over a graph is a sequentially connected set of edges.

A *subgraph* is a sub-set of the edges from an underlying graph. (The dotted lines in Figure 4.3 define a subgraph of the whole graph.)

A *labelled graph* is one where each edge is assigned some value. Labelled graph can be formally defined based on path algebra [Car79]. For the purpose of this thesis it is sufficient to restrict the edge values to be non negative numbers. A path can

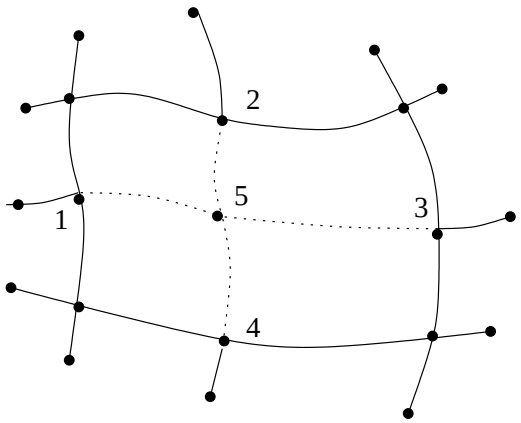


Figure 4.3: Subgraph and Boundary Nodes

be assigned a value based on the labels of its edges. Many real world problems are based on finding the shortest or longest paths with respect to these path values, known as the 'extremal' path problems.

A *spatial graph* is a graph whose nodes and edges are mapped into the plane with points and simple curves respectively.

A *planar graph* is a spatial graph whose edges touch only at their end points. Euler proved that if n is the number of nodes and e is the number of edges and f is the number of faces of a planar graphs, then, $n + f = M + 2$. A corollary of this is $e \leq 3n - 6$ [Goo98]. This means that planar graphs are sparse. Specialised algorithms can be devised for sparse graphs with a reduction in the computational effort required.

The *boundary nodes* of a subgraph are those nodes of the subgraph for which some of their incident edges may not be part of that subgraph. (Nodes 1, 2, 3 and 4 of the subgraph in Figure 4.3 are *boundary node*s of the subgraph. Node 5 is not a *boundary node* of this subgraph.)

A *connected graph* is a graph where each node in the graph can be reached from any other node in the graph when all the edges are treated as *undirected*.

4.2 Graphs in Databases

A graph is an abstract view of a set of objects and the binary relationship between pairs of objects in the set. A database derived from an entity relationship representation can therefore be viewed as a graph where the nodes of the graph are instances of the entities and the edges are instances of the relationships (Figure 4.4). The nodes and edges of the resulting database graph need not be homogeneous elements. In the same entity relationship representation, a cyclic many-to-many relationship over the same entity (Relation B-B) represents an explicit graph whose nodes and edges will be homogeneous.

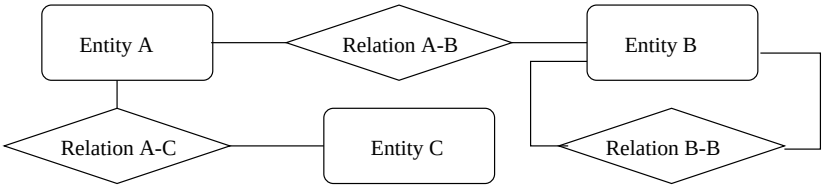


Figure 4.4: Entity Relationship Graph

In traditional database management systems a relation that represents an explicit graph is treated just as any other relation in the entity relationship representation. Graph related queries can still be directed at these relations to extract graph based information. Extensions to database query languages have been proposed for this purpose [MS90].

Capturing explicit graph semantics in the data model can lead to possibilities for capturing further information over the graph. GraphDB [Güt94a] provides the ability to define and store paths over graphs.

4.2.1 GraphDB

GraphDB is a database management system able to represent explicit graphs. Its strength lies in its provision for defining paths over explicit graphs. These graph concepts are incorporated into its query language.

In GraphDB the class concept of the object oriented paradigm is specialised to provide three distinct types of classes for defining the database schema. They are,

`Simple` classes , `Link` classes and `Path` classes. `Simple` classes represent the usual class–objects semantics, and may be subclassed. `Link` classes define a relationship between two `Simple` classes. When the relationship is over the same `Simple` class, the `Link` class will represent an explicit graph. A `Path` class may then be defined in the schema with respect to the `Link` classes.

For example, if a `Path` class is defined over a single `Link` class, then an instance of this `Path` class will be the concatenation of several instances of the corresponding `Link` class. In general a `Path` class may be defined over several `Link` classes as follows:

```
Path class aPath = aLink (bLink)* (cLink dLink)*
```

where the * indicates that several instances of that `Link` class may appear in `aPath`. Thus the definition of paths in GraphDB permits concatenation of links from different `Link` classes.

A GraphDB Example

The public transportation network example given to illustrate the use of Path classes in GraphDB is reproduced in this section [Güt94a]. The network is modelled using three layers of graphs, representing the physical network of routes and stations, the connection service between stations, and scheduled timetables (Figure 4.5).

The physical network is modelled with a graph in space. `Arc` is a subclass of the `Link` class and models line geometry of its edges. Nodes are instances of class (`Vertex`) which is a subclass of the `Simple` class.

`Phys-route` is a subclass of `Path` class, that is defined as the concatenation of several edges of the physical network (`Arc`). It forms a section of a regular service line that connects two stations.

At the second layer these connection services (`Connection`) between two stations are the edges of the graph whose nodes are `Station`. Thus `Connection` is a subclass of `Link` class and `Station` is a subclass of `Simple` class. A subclass of `Path` class, `Line`, defined as the concatenation of several `Connections`, represents a complete

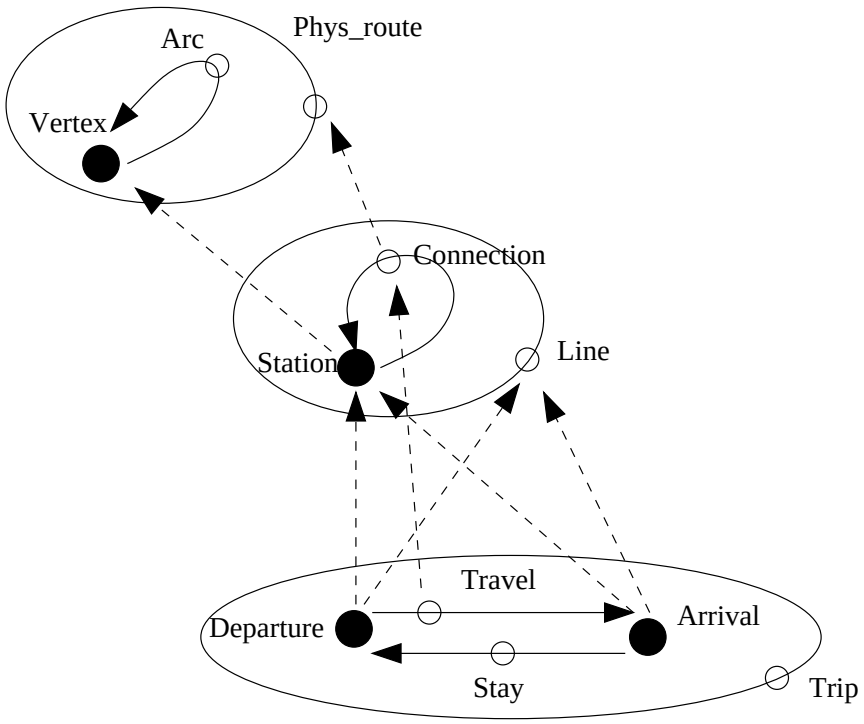


Figure 4.5: Transportation modelling with GraphDB

service line.

The third layer models the scheduled time tables. This consists of two subclasses of `Simple` class, `Arrival` and `Departure`, as nodes. Both the `Arrival` and `Departure` classes model events in time at a given station and relate arrival or departure for a service `Line` defined in the previous layer. Two subclasses of `Link` class, `Travel` and `Stay`, defined between the `Arrival` and `Departure` nodes, completely describe a particular service. `Travel` class represents a section of a particular service in terms of `Connection` defined in the second layer. `Stay` class represents the stop time of the service in a station (i.e. between `Arrival` and `Departure`). A subclass of `Path` class, `Trip`, defined as a concatenation of several instances of the two event classes, `Travel` and `Stay`, completely defines a particular service.

The example above illustrates how to store explicit graphs and different instantiations of the `Path` class at different levels of abstraction in a database.

4.2.2 Summary

Representation of explicit graphs within databases has received only limited attention, yet such schemas can allow paths to be defined over these graphs. This concept has many real world applications, such as in transportation networks. The following two chapters discuss multi-scale views of explicit graphs and dynamic graph aggregation. These concepts can also be applied to explicit graphs in databases.

4.3 Three Data Structures for Graphs

Three common data structures for representing the node pairs of a graph are the adjacency matrix, the edge list and the node-edge list. See Figure 4.6(b)–(d).

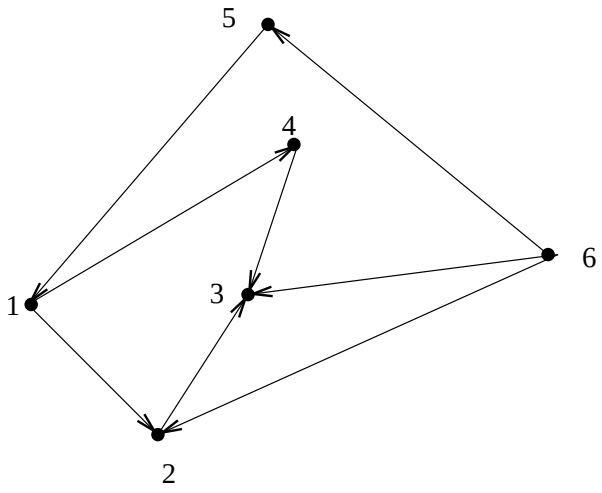
An entry in the adjacency matrix is set to a non-zero value if and only if there is an edge between the nodes corresponding to the column and row of that element.

The edge list data structure holds pairs of values or identifiers associated with the nodes, and each entry in the edge list represents an edge of the graph.

A node-list data structure consists of the list of nodes belonging to the graph. Each node in the list holds another list of nodes reachable from that node. If the graph can have 'multiple edges' between a given pair of nodes, then the list of nodes reachable from a given node must sometimes hold duplicate values. This then becomes ambiguous. Therefore, to represent graphs with 'multiple edges' each node must hold an edge list rather than a node list.

4.3.1 A Comparison of the Three Data Structures

Adjacency Matrix: All three data structures for the graph are capable of representing directed and undirected graphs. The adjacency matrix representation, however, can only represent a restricted class of graphs because it constrains the number of entries to the product of the number of nodes in the graph. Therefore an adjacency matrix holding binary entries cannot represent graphs with 'multiple edges'. In addition, when it is used to represent undirected



(a) A Graph

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	1	0	0	1
3	0	0	0	0	0	0
4	0	0	1	0	0	0
5	1	0	0	0	0	1
6	0	0	1	0	0	0

(b) Adjacency Matrix

1 - {2,6}
2 - {3,6}
3 - {}
4 - {3}
5 - {1,6}
6 - {3}

(c) Node-Edge List

(1 2)
(1 4)
(2 3)
(2 6)
(4 3)
(5 1)
(5 6)
(6 3)

(d) Edge List

Figure 4.6: Graph Data Structures

graphs half of its entries will be redundant. For not fully connected graphs this representation also tends to be sparse. While there are many strategies for compression of sparse matrices, updates to graphs in compressed matrix form may be inefficient. Within these limitations the matrix representation has all the advantages derived from a wide range of techniques developed for matrix manipulation.

Edge List: The edge list is a minimal representation scheme. With the exception of graphs with 'isolated nodes' the edge list permits the representation of all types of graphs. Use of additional identification values for edges permits 'multiple edges' to be recorded. It is compact and its size grows linearly with the number

of edges in the graph, as opposed to the size of an adjacency matrix, which is proportional to the square of the number of nodes in the graph. Undirected graphs may be stored twice as efficiently as directed graphs; by recording the edge only once and allowing traversal in both directions. Traversal efficiency must be achieved by indexing the edge list with nodes. In the case of a directed graph that is always traversed in one direction, the edge list could be ordered using the origin node so that all edges with the same origin are listed together. Graph updates are efficient in this representation and are identical to updates in a relational table in relational information theory.

Node-Edge List: The node-edge list representation may be viewed as the adjacency matrix representation without null elements. As such, the list attached to each node must hold identifiers of all nodes which are reachable by paths of length one. The node-edge list could, however, be modified to hold an additional edge identifier and thus be able to represent 'multiple edges' and 'graph loops'. The node-edge list representing a directed graph can also be viewed as the edge list representation indexed by nodes. The node-edge list therefore, is time efficient for graph traversal. When representing an undirected graph the node indexing scheme indexes only half of the nodes associated with the edges. It is therefore necessary to represent each edge in an undirected graph by two directed edges in a directed graph. This not only doubles the space requirement, but also loses the inherent semantics of undirected graphs. Edge list and node-edge list data structures have comparable space requirements, but the edge list representation is twice as efficient when representing undirected graphs. In terms of traversal efficiency, a node indexed edge list and a node-edge list are comparable.

The choice of data structure depends on the graph's properties, i.e. whether they are: directed graphs, undirected graphs, graphs with isolated nodes, or graphs with 'multiple edges'. It also depends on how it is used. Thus in addition to the graph type, the size of the graph to be represented, the efficiency requirement of graph traversal, and the frequency of updates to the graph itself must all be considered. The choice of data structure will also be influenced by space–time trade-offs.

4.3.2 Recording Graph Element Attributes

Graph algorithms require access to the attributes of the graph nodes and edges. It is assumed that these are represented by their identifiers, which allow their attributes to be recorded and accessed in some manner. The efficiency of this access depends on the choice of data structure. The edge list representation permits the recording of edge attributes as part of the edge list structure, and accessing them as part of the traversal operation. Node attributes, however, need to be recorded separately and accessed through a separate operation. In the case of a node-edge list representation, the recording of node attributes can be part of this same structure. Attributes of the current node during a traversal operation can, therefore, be accessed as part of that operation, but attributes of reached nodes and traversed edges must be accessed by separate operations.

4.3.3 Choice of Graph Data Structure

Based on these observations, it can be concluded that, for the purpose of information representation, space efficiency, traversal efficiency and flexibility in representing all types of graphs, the edge-list data structure offers several benefits over the other two representations. Its inability to record 'isolated nodes' is not a serious disadvantage since most useful graphs in space will not have any 'isolated nodes'.

Graph aggregation, the focus of this work, requires the representation of subgraphs. Subgraphs of a graph stored as an edge list can be represented as a subset of the edge list. Adjacency matrix and node-edge list representation do not allow such a straightforward representation of subgraphs.

The edge list data structure has additional advantages when used to represent explicit graphs within a database. Güting [Güt94a] suggests a representation scheme for the objects of a class hierarchy in a database. The class hierarchy tree is divided into sub-trees (Figure 4.7). Each sub-tree is called a kernel class. All objects belonging to all classes in a kernel class are stored as a single list. The appropriate selection of kernel classes can enhance the efficiency of access to objects. Based on this scheme, an explicit graph represented by a kernel class of `Link` classes in

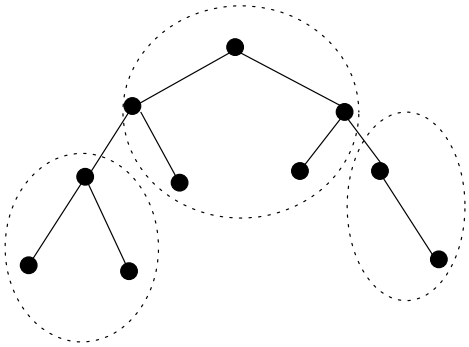


Figure 4.7: Kernel Classes

GraphDB can be maintained as a list. This, then, is the same as the edge list data structure discussed above.

A graph represented in an edge list structure is well suited to the proposed graph generalisation scheme. The implementation of the proposals in this work is described in Chapter 9. The data model used in the implementation uses an edge list representation of graphs in space.

4.4 Graph Algorithm Specification

Graph algorithms, in general, share many common characteristics. Their execution requires a mechanism to get from a given graph element to its neighbours. Erwig [Erw93] develops a classification for graph algorithms. He suggests capturing the processing order of graph elements by placing the out-edges from a node into a list type data structure. Stack, queue and sorted list are the three types of *order enforcing lists* used for this purpose.

In addition to the standard list behaviour of the above three types of data structures, additional constraints can be placed on retrieving elements from these lists. For most algorithms, a valid constraint is to allow all edges to be traversed only once. Thus, this will be assumed for all *order enforcing lists*. A second constraint is to allow a node to be reached only once. Restricting the node access to either once or multiple times gives the lists the *single get* or *multiple get* behaviour.

The stack, queue or sorted list determine the order in which edges are traversed,

while the *single get* and *multiple get* behaviours determine the pattern resulting from a traversal. A *single get* behaviour will yield a tree, while *multiple get* behaviours will yield a graph.

At any stage of the execution, a graph algorithm is focused on one particular graph element, which will be referred to as the current element. Such graph elements are either nodes or edges of the graph. For the sake of uniformity the graph element may be specified by an edge of the graph. The current node is then specified as the start node of the current edge.

Based on the above observations, the following two data structures provide the flexibility to specify the input, output and execution of graph algorithms.

Current edge: This is the current edge being traversed. Its start node is the current node of traversal. Its end node is the next current node of traversal.

Order enforcing list: This is the order enforcing data structure; stack, queue or sorted list and the associated *single get* or *multiple get* behaviour.

This kind of algorithm specification is a complex instance of the *Iterator* pattern of [GHJV95] for iterating over graphs. The style of description, derived from [Erw93], is used in this thesis to describe the algorithm in Section 7.1 and Section 7.3.1.

4.5 Graph Based Relationships

Formal definitions of spatial relationship among objects have been developed [Güt94b]. These are presented below followed by parallel definitions for graph based relationships.

4.5.1 Spatial Relationships

Any spatial object consists of a boundary and an interior, and by implication everything else is its exterior. In the vector based 2-D spatial data model, the boundary and interior can be represented using the 0, 1 and 2-cells.

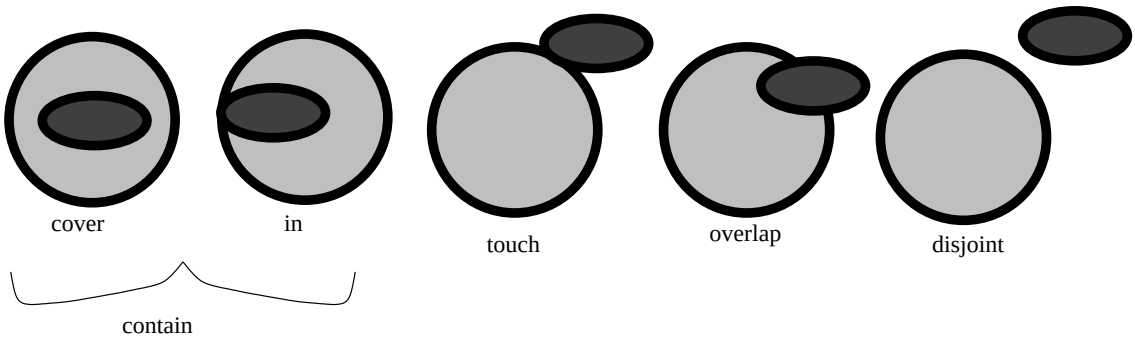


Figure 4.8: Topological Relations

In Figure 4.8 the black border area represents the boundary of the objects and the interior is represented by other shades of grey. There are four possible relationships between two objects, X and Y: boundary intersection ($X_B \cup Y_B$), interior intersection ($X_I \cup Y_I$), intersection of the boundary of X with the interior of Y ($X_B \cup Y_I$), and intersection of the interior of X with the boundary of Y ($X_I \cup Y_B$). Each of these relationships can yield an empty or a non-empty set, leading to 16 combinations, although several are not valid. For example, if the intersection of the two interiors is empty and the intersection of the two boundaries is also empty, then the two objects are disjoint irrespective of the intersection of the boundary of one with the interior of the other. Thus the three relationships between one boundary and the other interior, namely:

1. $(X_B \cup Y_I) \neq \phi$ and $(X_I \cup Y_B) \neq \phi$
2. $(X_B \cup Y_I) = \phi$ and $(X_I \cup Y_B) \neq \phi$
3. $(X_B \cup Y_I) \neq \phi$ and $(X_I \cup Y_B) = \phi$

do not represent distinct relations. Six valid relationships have been identified: *cover, in, touch, overlap, disjoint and equal* (Figure 4.8).

A *cover* relationship exists when one object is completely contained within another object and the boundaries of the two do not intersect. A point within a polygon and a city within a country are both *cover* relationships. A special case of the *cover* relationship is the *in* relationship, where the boundaries of the two objects do intersect. A generic term for both these relationships is the *contain* relationship.

The other four relationships are similarly defined. These relationships are adapted for graphs and subgraphs and used for the development of graph nodeflex.

4.5.2 Graph Relationships

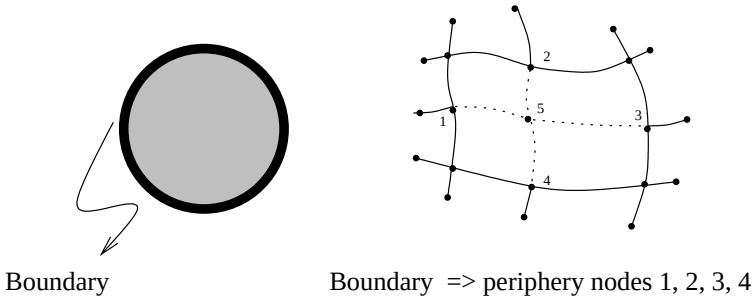


Figure 4.9: Object and Subgraph Boundaries

Graph relationships defined below are extensions of the generic spatial relations described in the last section. These are defined with respect to the edge sets and node sets of graphs.

A *contain* relation between two graphs implies that the edges of one graph is a subset of the edges of the other graph. In other words one is a subgraph of the other. A graph as an edge set does not have a boundary. Therefore the distinction between the *cover* and *in* relations disappears.

An *overlap* relation between two graphs implies that some edges are shared by the two graphs.

A *touch* relation between graphs implies that the two graphs share some nodes but do not share any edges.

Disjoint graphs imply that the graphs do not share edges or nodes.

A graph does not have a boundary, however, in the case of subgraph, the *boundary nodes* define its boundary. Figure 4.9 relates the object boundary concept to the *boundary nodes* of a subgraph (Section 4.1). Using this, the *cover* and *in* relations can be defined with respect to subgraphs. Note that the nodes involved in a *touch* relation between two subgraphs will be their *boundary nodes*.

A *cover* relation between two subgraphs is a specialisation of the *contain* relation between two graphs. Therefore the edge set of one subgraph is a subset of the edge set of the other subgraph. In addition a *cover* relation implies that the sets of *boundary node*s of the two subgraphs do not overlap.

An *in* relation between two subgraphs is a specialisation of the *contain* relation between two graphs. Therefore the edge set of one subgraph is a subset of the edge set of the other subgraph. In addition an *in* relation implies that the sets of *boundary node*s of the two subgraphs overlap.

The *boundary nodes* of subgraphs, and the *in* and *cover* relationships defined above, are used in the development of graph aggregation in the following chapters.

4.5.3 Graph Whole–Part Associations

B, C, D, E partitions A B, C, D, E covers A

Figure 4.10: Graph Whole–Part Associations

Two types of whole–part topological relations are: *partition* and *coverage* [KPS97] (Figure 4.10). In a *partition* relation all partitioned objects have a *contain* relation with the whole object and a *touch* relation with the other partitioned objects. In a *coverage* relation the partitioned objects again have a *contain* relation with the whole, but some of them may have an *overlap* relation with the other partitioned objects. These are also defined below with respect to graphs.

Graph partitioning implies that graph edges are partitioned into subgraphs. This concept can be extended to an allocation of nodes as follows. The *touch* relation exists between the partition subgraphs implying that they share some

of their *boundary nodes*. These shared nodes can be allocated to the partition corresponding to one of the subgraphs sharing it, and removed from the partition corresponding to all other subgraphs sharing it. All other nodes are assumed to belong to the partition corresponding to the only subgraph that refers to it through its edges.

Graph coverage divides the set of graph edges into overlapping subgraphs. As in partitions, graph coverage can be extended to include allocation of nodes. In this case the *boundary nodes* of each subgraph could be either allocated to, or removed from, the corresponding partition.

Graph associations can be applied to any graphs. When applied to graphs in space they have spatial semantics. Labelled spatial graphs can be used to model population data. Traditionally zones in urban space are represented as polygonal partitions. Zone overlap from two or more zonings is determined by computing polygon overlays. When population data is represented as a graph, zones based on population distribution can be represented as subgraphs. Representing zones as subgraphs is much more space efficient than defining polygons for each zone. When zones are represented as subgraphs of the population graph, zone overlap can be computed by matching subgraphs. This is a strong argument for representing population data as graphs.

4.6 Summary

This chapter outlines some graph concepts to be applied to develop the graph aggregation scheme in the following three chapters. These are: the edge-list structure as the most appropriate data structure for representing spatial graphs, a graph algorithm specification style, and some graph associations. Explicit graph semantics in databases is also discussed. This can be used for representing spatial graphs, as demonstrated by GraphDB. Graph aggregation can then be used to provide multi-scaled views of the graph in the database.

Chapter 5

Graph Aggregation

An aggregation hierarchy aggregates several elements into a single element and repeats this to many levels. What is the semantics of such an aggregation applied to a graph? In the following two chapters a theory of graph aggregation and its representation is developed. This takes a given graph and generates another graph with a reduced number of graph elements. This process can be continued to multiple levels, resulting in a hierarchy over a graph.

Graph aggregation may be given differing interpretations based on the treatment of its nodes and edges during the aggregation process. A graph aggregation hierarchy can be built into the model when it is constructed. The multi-scaled graph model of Mainguenaud is such a model. It is also useful to generate graph aggregation hierarchies on an ad hoc basis. The aggregation scheme, referred to as graph nodeflex, and developed in the following chapters, is such a scheme.

Nodeflex is based on node based graph aggregation. In this Chapter node based graph aggregation is discussed and graph nodeflex is then defined based on this. Nodeflex interpretation is then compared with the model given in [Mai95].

5.1 Node Based Graph Aggregation

A particular interpretation for graph aggregation, namely node based graph aggregation, is introduced in this section. It defines the structural (nodes and edges) organisation of a node based graph hierarchy. The rules for defining this are not considered here. Graph nodeflex is then defined in Section 5.1.4 as node based graph aggregation with additional constraints. Rules for defining graph nodeflex are considered in detail in Chapter 6.

5.1.1 Single Level Aggregation

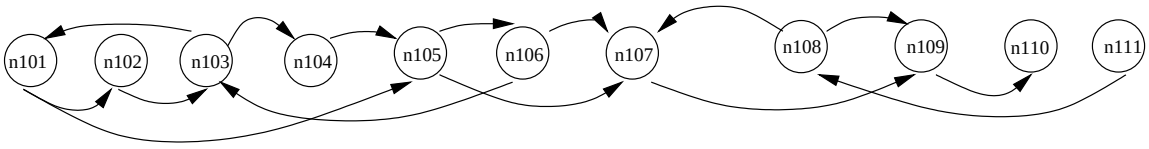


Figure 5.1: Graph

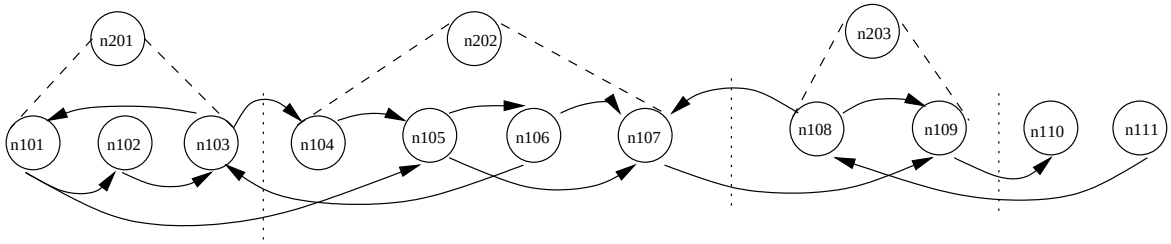


Figure 5.2: Node Hierarchy

Consider the graph in Figure 5.1 where the circles represent nodes and arrows represent edges. A set of nodes and associated edges in this graph can be treated as a node at a higher level. Figure 5.2 shows a node aggregation of the graph in Figure 5.1. Nodes n201, n202 and n203 are an aggregation of a set of nodes and associated edges of the original graph.

The incoming and outgoing edges of an aggregate node are those edges that have one of its nodes inside the aggregate node and the other node outside of it. For the aggregate node n202, the edge n103–n104 is one of its incoming edges and edge n106–n103 is one of its outgoing edges.

Based on this aggregation, a graph can be defined as consisting of nodes n201, n202, n203, n110 and n111. For the purpose of traversing the aggregated graph, each aggregate node is considered to have the incoming and outgoing edges as described above. If the nodes at the other end of these edges are inside another aggregate node, these edges are equivalent to edges between the two aggregate nodes. For example the edge n107–n109 in Figure 5.2 can also be regarded as an edge between the aggregate nodes n202 and n203.

5.1.2 Multi Level Aggregation

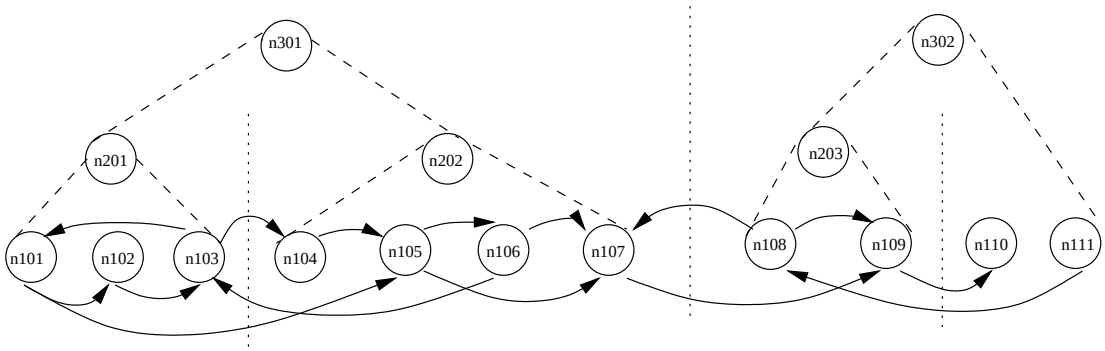


Figure 5.3: Multi Level Node Hierarchy

Node based aggregation can be taken to many levels, as shown in Figure 5.3. The meaning of incoming edges and outgoing edges of the aggregate nodes at each level of aggregation will be the same as that of single level aggregation. We will use the term *base level* to refer to the base graph and the refer to every other aggregated level as first level, second level and so on.

An aggregated graph may be viewed at any given level of aggregation. Alternatively each aggregate node could be selectively expanded to the desired level of detail, thus resulting in a particular view of the aggregated graph. For example, a particular view of the aggregated graph in Figure 5.3 could include the following nodes and edges: n302, n201 and n202. In this case node n301 is expanded into n201 and n202, while node n302 is not.

5.1.3 Edge Aggregation

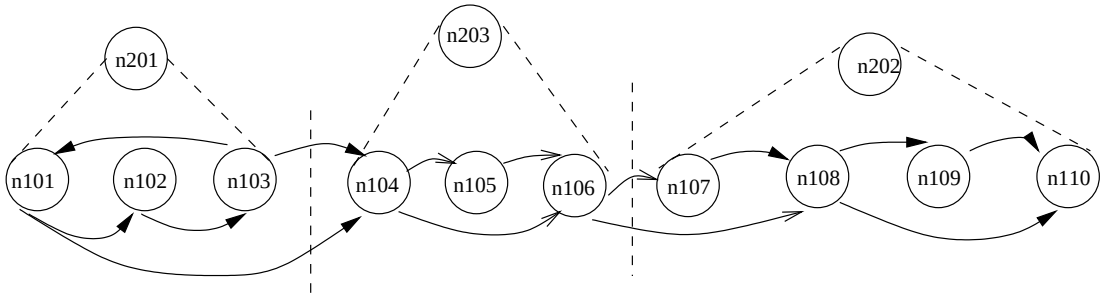


Figure 5.4: Edges Between Aggregate Nodes

All the edges in a node based aggregation are those that connect the nodes at the bottom-most level. These same edges are visible at higher levels as incoming and outgoing edges of the aggregate nodes.

An aggregate element could also be interpreted as an aggregate edge when it satisfies one condition. In Figure 5.4, the aggregate element n203 could be viewed as an aggregate edge between nodes n104 and n108. In this case there must be exactly two nodes within the aggregate element that have edges leading to outside nodes. The representation scheme developed in the next chapter ignores the aggregate edge interpretation for reasons discussed there.

5.1.4 Graph Nodeflex

In practice, ad hoc graph aggregation is often based on the adjacency property of graph elements. This is especially true for graphs in geographic space. Therefore it is reasonable to assume that the subgraphs representing the aggregate nodes are *connected* (Section 4.1). By implication each *connected* subgraph representing an aggregate node is *disjoint* (Section 4.5) with respect to all other subgraphs in the node based aggregation.

A node based aggregation where the set of edges which constitute an aggregate node is a *connected* subgraph, and each such subgraph is *disjoint* with respect to all other subgraphs, is termed a *nodeflex* of a graph.

All the concepts derived from here onwards are only applicable to nodeflex where all subgraphs are connected. This connectedness property permits a simple representation scheme for graph nodeflex.

The graph nodeflex is one interpretation of graph aggregation. For purposes of comparison, the next section describes the graph aggregation semantics of [Mai95], which is defined during the model building phase.

5.2 A Multi-scaled Graph Model

The multi-scaled model of [Mai95] consists of a hierarchy of nodes maintained in a set of layers. A parallel and similar hierarchy of edges is also maintained. The graph is defined using four types of objects. These are base-node, base-edge, master-node and master-edge. The base-nodes and master-nodes form an aggregation hierarchy. Similarly the base-edges and master-edges form another aggregation hierarchy. A node or an edge at a higher level may represent an aggregation of nodes or edges at a lower level (Figures 5.5 and 5.6).

Every graph element at a higher level is assumed to have a shadow presence at all the lower levels. These shadow elements are also aggregated into the corresponding elements at higher levels. The shaded elements in Figure 5.5 are the shadow elements.

In addition to the aggregate nodes and edges, a new node or an edge may be added at a higher level independently of the nodes and edges at the lower levels. These independent elements are then repeated at the lower levels as shadow elements. Node S22 is a node added at level one and repeated at the level below, the base level, as shadow node -S22. Thus new graph elements added at higher levels are also converted into aggregate elements whereby they aggregate only their shadow in the lower levels. In this way a very strict layer hierarchy is maintained and the nodes and edges at the lowest level are the base-nodes and base-edges, while those at all the other levels are the master-nodes and master-edges.

Nodes or edges at a given layer which are not aggregated into a node or edge at

the layer above it, are aggregated into a shadow element in that layer. Thus in Figure 5.5, node S17 is aggregated into the shadow element -S31.

The shadow elements are not shown in Figure 5.6.

A brief description of the representation of this graph aggregation follows.

5.2.1 Representation of Hierarchy

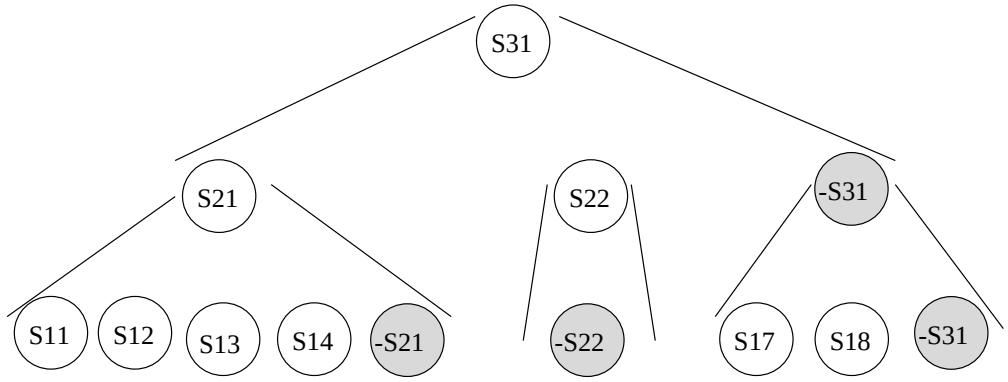


Figure 5.5: Node Hierarchy in a Multi-scaled Model

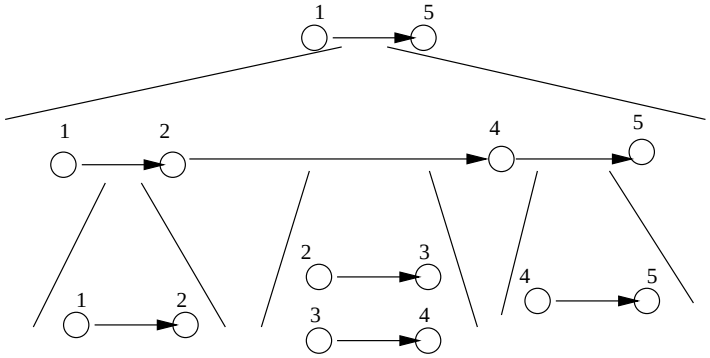


Figure 5.6: Edge Hierarchy in a Multi-scaled Model

The node hierarchy and the edge hierarchy are maintained by associating layers to each level in the hierarchy. Layering allows nodes and edges at a given level of abstraction to be retrieved.

Each element in the multi-scale model is assigned a unique identifier. In addition, each element is assigned an identifier list. There are as many identifiers in this list

as there are layers in the hierarchy. These identifier lists encode the hierarchy. An element's unique identifier appears in its identifier list in the corresponding place of the layer to which the element belongs. For example, in Figure 5.5 node S11 has the object identifier list: $< S31, S21, S11 >$. Note that its unique identifier S11 appears as the third element which corresponds to the third layer, the layer to which S11 belongs. Similarly node S31, S22, and S17 have the object identifier lists: $< S31, null, null >$, $< S31, S22, null >$ and $< S31, -S31, S17 >$ respectively. These identifiers and identifier lists are specific to the graph model representing the hierarchy and are independent of the object identifiers in the regular database that stores the alphanumeric data.

As mentioned above all elements are assumed to have a shadow presence in all the layers below it. The identifiers for these shadow elements are denoted by negative values of their own unique identifier.

5.3 Summary

Graph aggregation aims to represent a graph in multiple levels of detail. A node based graph aggregation semantics was introduced, referred to as graph nodeflex, and compared with another multi-scaled model.

The multi-scaled graph model of [Mai95] provides a means of building a set of hierarchically structured nodes and edges. The scheme assumes a single hierarchy, which allows many application dependent views to be derived from it - although the underlying hierarchy is fixed. Multiple aggregations with differing node and edge hierarchies are not possible. The object identifier system of maintaining the hierarchy, described in Section 5.2.1, requires a large amount of storage for a large graph. It also permits the addition of new graph elements at any level independently of elements at lower levels. This enables the model to capture a rich aggregation semantics.

Another requirement of graph aggregation arises when a graph needs to be aggregated into a graph of smaller size, so that algorithms can be executed over the reduced graph. Spatial analysis can make use of this type of aggregation. This

requirement necessitates ad hoc aggregation of graphs, rather than an aggregation that is built into a data model. The multi-scaled model's built-in encoding scheme makes it difficult to support such ad hoc aggregation.

In the next chapter an ad hoc graph aggregation scheme based on graph nodeflex is developed. Graph nodeflex appears to be a true subset of the aggregation semantics of the multi-scaled model. Graph nodeflex also permits aggregation hierarchies. By restricting the edge in a graph nodeflex to be at a single level, the representation scheme is simplified. The storage intensive object identifier scheme of [Mai95] is also eliminated.

The representation scheme is the major difference between the two approaches. The multi-scaled model is more space intensive and requires a manual model building phase, while the ad hoc approach proposed is more space efficient and can be built from the data associated with graphs and algorithms (Chapter 7). Both models permit a similar degree of flexibility in accessing the graph at different levels of detail.

Spatial aggregation (Section 3.4) is also based on graphs. Here each level of aggregation detects subgraphs and converts them into nodes. In this approach, no interpretation is given to the existing adjacency relation between aggregate nodes. Rather, once aggregate nodes are detected and converted to simple ones, new neighbourhood graphs are built using these newly created nodes. In this respect this approach differs from graph nodeflex, which promotes all existing adjacency relations to the next level.

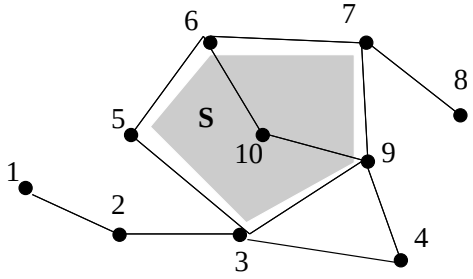
Chapter 6

Graph Nodeflex

Representation of the multi-scaled model of [Mai95]was surveyed in the last chapter. Each node and edge in the aggregated graph of this model is assigned a set of logical identifiers associated with a fixed set of layers. This set of logical identifiers together with the set of fixed layers structure the graph aggregation hierarchy.

A much simpler representation of graph hierarchy based on graph nodeflex is developed in this chapter. Due to its simplicity the scheme is suitable for dynamic aggregation. Applications are discussed in the next chapter. The representation also lends itself to user interfaces for creating and editing the associated data structures as described in Chapter 9.

Graph nodeflex is defined based on *connected* subgraphs, where connectedness is defined assuming graph edges are undirected (Section 4.1). Thus the definition of graph nodeflex is primarily applicable to undirected graphs. However, nodeflex can also be defined over directed graphs by treating each of its edges as undirected for the purpose of defining the subgraphs of the nodeflex. Application of such an interpretation of graph nodeflex over directed graph is discussed in Section 7.2.3. All the examples used in the following chapters are undirected graphs.



(a) Subgraph

Graph		Map
1	2	
2	3	
3	5	s
5	6	s
3	4	
3	9	s
4	9	
9	7	s
6	7	s
9	10	s
6	10	s
7	8	

(b) Edge-map Showing the Subgraph

Figure 6.1: A Graph

6.1 Subgraphs and Edge-map

In the last chapter graph nodeflex was defined as follows:

- All the edges of graph nodeflex are added to connect the nodes at the base level.
- Aggregate nodes are *connected* subgraphs of the original graphs.

Subgraphs can be maintained as a subset of the edge set of a graph. The most flexible and simplest method for maintaining subsets is to keep another collection of identifiers of the objects in the main collection to represent the subset. Any subset maintained as a collection of identifiers will be valid as long as the main collection is not subject to modifications. When the objects in the main collection are deleted, the subset collection becomes invalid if the deleted object also belongs to the subset.

Another approach to define a subset is to use a *mapping*.

A *mapping* is defined via object–value pairs for a collection of objects.

A subset can be defined using a mapping in conjunction with a predicate based on the values recorded in the mapping.

Thus *subset* is given by a *mapping* and a *predicate* on the mapping values.

This means that the inclusion/exclusion of an object in {M} into the subset in {S} is indicated by the true/false value of the predicate over the values in the mapping.

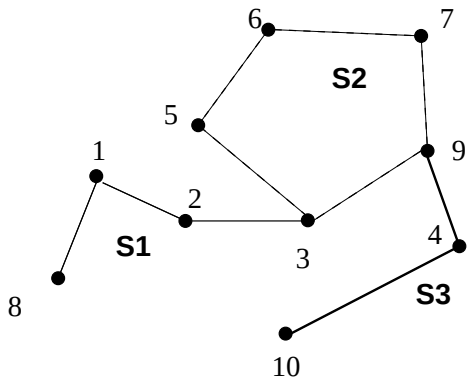
A mapping over the edge set {E} of a graph is referred to as an *edge-map*. A *subgraph* of the graph can be viewed as an *edge-map* and a *predicate* on the edge-map values.

Tables, as in Figure 6.1(b); will be used to display the values in an edge-map. The first column of the table named 'graph' will represent the edge list of the graph; and the second column will represent an edge-map. In all the tables used here subgraphs are defined by the predicate ($= value$). The table in Figure 6.1(b) above shows an edge-map that will represent the shaded subgraph S in Figure 6.1, with the predicate ($= s$).

Traversing a subgraph, defined using an edge-map, is achieved by traversing the main graph. The predicate is used to select only the edges of the subgraph. Since the subgraphs are *connected*, the entire subgraph can be traversed in this manner, starting from a node.

6.1.1 Multiple Subgraphs

A collection of non-overlapping subgraphs can be maintained in a single edge-map by using a distinct predicate for each subgraph. The table in Figure 6.2 shows an edge-map that represents the set of non-overlapping subgraphs, S1, S2, S3, marked on the graph in the figure. Another relation associating each subgraph with its edge-map value will complete the definition.



(a) Subgraphs

Edge list		Map
1	2	s1
2	3	s1
1	8	s1
4	10	s3
3	5	s2
5	6	s2
3	9	s2
9	4	s3
6	7	s2
7	9	s2

(b) Edge-map

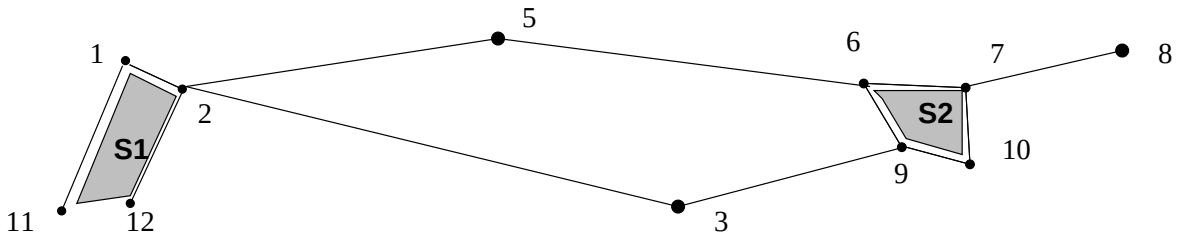
Figure 6.2: Multiple Subgraphs

6.2 Single Level Graph Aggregation

The graph in Figure 6.3(a) will be used as an example to develop a representation scheme for graph nodeflex, using edge-maps. S1 and S2 are two *connected* subgraphs. They are also *disjoint*. Therefore they can be treated as two aggregate nodes of a nodeflex. The edge-map in Figure 6.3(b) and the relation in Figure 6.3(c) represents these two subgraphs.

6.2.1 Optimising an Edge-map

Consider the table in Figure 6.3(d) where all edge-map values in the table in Figure 6.3(b) are replaced with 1 or 0. In this table, all aggregated edges are given the value 0 and others are given the value 1. Since the subgraphs are *connected* and *disjoint*, given any node of the subgraph they can be extracted using this edge-map and the predicate $(= 0)$. Thus to complete the definition another relation that associates the subgraphs with any one of its node is needed.



(a) Aggregate nodes

Edge list		Map
1	2	s1
2	12	s1
2	3	
3	5	
1	11	s1
3	9	
5	6	
6	9	s2
6	7	s2
7	10	s2
7	8	
9	10	s2

(b) Nodeflex Map

Aggregate Node	Edge-map Value
S1	s1
S2	s2

(c) Edge-map Values

Edge List		Map
1	2	0
2	12	0
2	3	1
2	5	1
1	11	0
3	9	1
5	6	1
6	10	0
6	7	0
7	9	0
7	8	1
9	10	0

(d) Alternate Map

Aggregate Node	Boundary Node
S1	2
S2	9
S2	6
S2	7

(e) Boundary Nodes

Figure 6.3: Graph Nodeflex

The table in Figure 6.3(e) is such a table that associates each subgraph with not one, but a specific set, of its nodes. This specific node set consists of the *boundary nodes* (Section 4.1) of the subgraph as described below.

The edges from the aggregate nodes, S1 and S2, are those edges from a node in the corresponding subgraph to a node that is not in that subgraph. The edges incident to the aggregate node S1 in the above graph are edge 2–3 and edge 2–5. The edges emanating from the aggregate node S2 are edges 7–8, 9–3 and 6–5. Thus node 2 is a *boundary node* of S1, and nodes 6, 7 and 9 are the *boundary node*s of S2.

6.2.2 Pseudo-Edges

The *boundary node* table in Figure 6.3(e) shows the relation between *boundary node*s and their corresponding aggregate nodes. Each entry in this table can be treated as an additional edge, with no associated cost, of the aggregate graph. These will be referred to as the pseudo-edges. Traversing an aggregate graph can now be simulated by using the initial graph, the edge-map and the pseudo-edges. The edge-map is used during traversal to eliminate all edges that are part of a subgraph representing an aggregate node. Pseudo-edges provide the links to traverse across the aggregate nodes. An example traversal using pseudo-edges is presented in Section 7.3.1.

6.2.3 Equivalence of Edge-map and Nodeflex

The alternative representation of a graph nodeflex in Figure 6.3(d) suggests that any edge-map consisting of values, 0 and 1, can be viewed as a graph nodeflex of two levels. In other words, from any such edge-map, a set of disjoint connected subgraphs and its boundary nodes can be derived so that all the edges in each subgraph have a value of 0. This observation about the reduced edge-map can now be applied to develop a representation for a hierarchy of graph aggregation.

6.3 Aggregation Hierarchy

The following two assertions have been made so far:

- All aggregate nodes, at any level of the aggregation, can be viewed as *connected* subgraphs of the original graphs.
- An edge-map with values 0 and 1 can be viewed as a single level graph aggregation hierarchy.

If the graph aggregation is taken to several levels, then an edge set of a *connected* subgraph representing an aggregate node may be contained in another edge set of an aggregate node at a higher level. Figure 6.4(a) shows the sets of edges of a graph aggregation hierarchy as a Venn diagram. Each *connected* subgraph is represented as a set in this Venn diagram. The *contain* relation among the edge sets is equivalent to a *contain* relation (Section 4.5) between the corresponding subgraphs. The Venn diagram is also represented as a tree in Figure 6.4(b), where the link between a parent and child in the tree represents a *contain* relation.

At the first level of aggregation is the set of subgraphs $L1N1$ to $L1N11$, denoting a set of aggregate nodes. These, together with the original graph, represent a new aggregate graph whose aggregate nodes are $L1N1$ to $L1N11$.

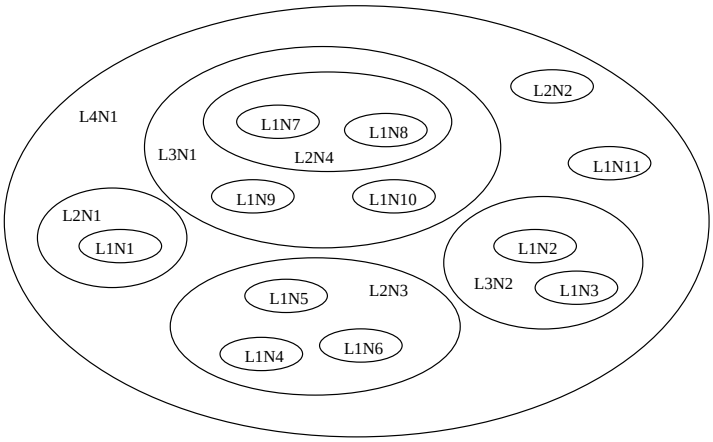
The second level of aggregation consists of aggregate nodes $L2N1$ to $L2N4$. Note that the aggregate node $L2N2$ does not contain any aggregate node of the first level.

At the third level are the aggregate nodes $L3N1$ and $L3N2$.

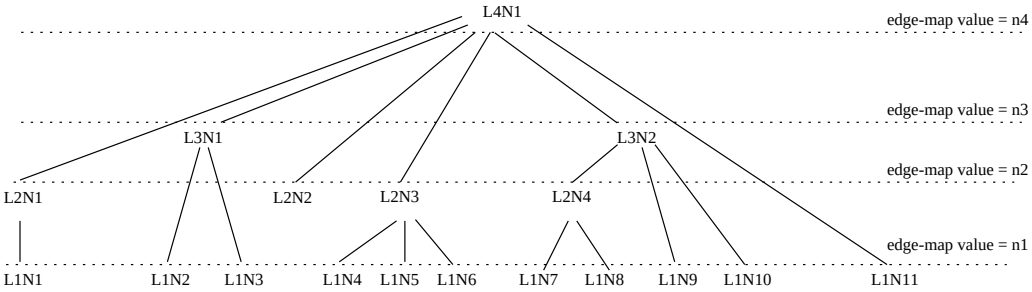
The edge set of the main graph, $L4N1$, *contains* a set of disjoint subgraphs, $L2N1$, $L3N1$, $L2N2$, $L2N3$, $L3N2$, and $L1N11$. Each of these disjoint subgraphs may *contain* another set of disjoint subgraphs.

Two additional assertions can now be made about the subgraphs of a graph hierarchy based on the above observation:

- The subgraphs associated with the aggregate nodes impose a subdivision of the single level edge set into non-overlapping and contained sets as shown in



(a)



(b)

Figure 6.4: Edge Sets of Aggregate Nodes

the Venn diagram in Figure 6.4. The *contain* relations among these subgraphs form a tree. This tree will be referred to as the *aggregate node tree*.

- All pairs of non-overlapping subgraphs of a graph nodeflex are disjoint. In other words these subgraphs do not share edges of the first level graph or nodes at any level in the hierarchy.

The edge-map representation of nodeflex of Section 6.2 can now be extended for multiple levels.

Let $n_1 < n_2 < n_3 ... < n_p$ be a set of ordered values in the edge-map.

A set of disjoint connected subgraphs can be derived from the edge-map so that the edge-map values of all the edges in these subgraphs is equal to n_1. Let this set of

subgraphs be $\{Sn_1\}$. In terms of the subgraphs in Figure 6.4(b), this set is:

$\{Sn_1\} = \{L1N1, L1N2, L1N3, L1N4, ...L1N11\}$.

Another set of disjoint connected subgraphs can be derived from the edge-map so that the edge-map values of edges in these subgraphs is $\leq n_2$. Let this set of subgraphs be $\{Sn_2\}$. Again in terms of the subgraphs in Figure 6.4, this set is:

$\{Sn_2\} = \{L2N1, L2N2, L2N3, L2N4, L1N9, L1N10, L1N11\}$

The selection criteria $\leq n_2$ ensures that all subgraphs in {Sn1} are part of some subgraph in {Sn2}.

Case (1): some subgraphs in $\{Sn_2\}$ may have all edge values $= n_1$. For the example used these are: $\{L1N9, L1N10, L1N11\}$.

Case (2): some subgraphs in $\{Sn_2\}$ may have all edge values $= n_2$. For the example used this set is empty: $\{L2N2\}$.

Case (3): some subgraphs in $\{Sn_2\}$ may have all edge values $\leq n_2$. For the example used these are: $\{L2N1, L2N2, L2N3\}$.

Subgraphs of Case (1) represent aggregate nodes at level 1. Subgraphs of Case (2) can be interpreted as aggregate nodes at level 2. These aggregate nodes do not include any aggregate nodes at level 1. This is because subgraphs in this set do not contain any edges with values n_1.

The subgraphs in Case (3) can be interpreted as aggregate nodes at level two, which includes some aggregate nodes at level one. A set of such subgraphs whose edge values are n_1 can be extracted from the subgraphs in Case (3). A point to note in this case is that the subgraphs in Case (3) have a *contain* relation (Section 4.5) with the corresponding subgraphs of level one extracted from it. However, this may be either a *cover* relation or an *in* relation . This point is relevant when designing algorithms to extract the *boundary node*s of the subgraphs in Case (3), as described in Chapter 7. In general, the *boundary node*s of a lower level subgraph will also be the *boundary node*s of the higher level subgraph when it is an *in* relation.

Thus p number of ordered values $n_1 - n_p$ in an edge-map can be viewed as aggregations to level p. In other words, edge-map of a graph with ordered values $n_1 - n_p$

completely defines a nodeflex of the graph to level p.

Thus any edge-map whose values have an implied order can be interpreted as a nodeflex of the graph, and therefore a particular aggregation of the original graph.

The term nodeflex is also used to refer to such an edge-map with ordered values.

6.3.1 Pseudo-Edges

To enable graph traversal at a selected level the *boundary node* table can be created as in single level aggregation. An additional column in this table is also added to hold the corresponding level of the aggregate node. The *boundary node* table for the aggregation hierarchy in Figure 7.8(a) is shown in Figure 7.8(c). Figure 7.8(d) shows the aggregate node tree for this graph nodeflex. An algorithm for the extraction of pseudo-edges and the aggregate node tree from a nodeflex is given in Section 7.1.

6.4 Aggregate Edges

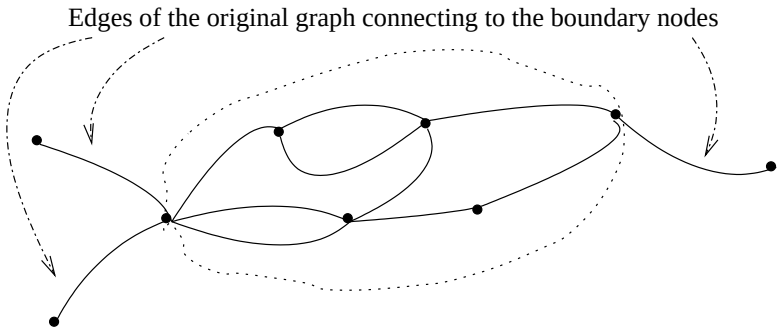


Figure 6.5: Aggregate Element With Two Boundary Nodes

The aggregation scheme we have described does not consider edge aggregation and assumes that all edges are at a single level. The following observations can be made regarding edge aggregation.

In general any aggregate element may be considered as an aggregate edge if it has only two *boundary node*s (Figure 6.5). While maintaining the representation scheme as described above, the 'aggregate edge' semantics could be enforced during graph

traversal. However, since the representation scheme is based on the assumption that all aggregate elements are disjoint, *an aggregate edge cannot directly connect two aggregate nodes*. Additional intermediary edges of the original graph must connect the *boundary nodes* of the aggregate edge to other graph nodes. Therefore, the aggregate edge concept within this representation is not flexible and has not been developed further.

6.5 Summary

The proposed graph nodeflex concept provides a very simple and efficient means of viewing a graph at a higher level. Any edge-map whose values have an implied order can be interpreted as a nodeflex of the graph, and therefore a particular aggregation of the original graph. The edge-map view of graph aggregation gives some insight into the inherent hierarchy present in a graph with respect to some values associated with each graph edge. These values may represent the length of the edge of a Euclidean graph, or some other value which may be meaningful in the resulting aggregation.

Each aggregate graph element can be selectively decomposed into its constituent nodes and edges independently of the other graph elements in the current development of the graph. This provides an easy interface for viewing the graph at different levels of aggregation. The same base graph may also be aggregated in several different hierarchies. These options of selective development of a given graph aggregation hierarchy and multiple independent aggregations are useful in spatial analysis where different aggregations may be used for studying behaviour of populations in different contexts. This application is discussed in the next chapter (Section 7.3).

Graph nodeflex was tested in an implementation using Smalltalk. This is described in detail in Chapter 9. This implementation was used to apply graph nodeflex to find shortest paths in a transport graph. The results of these tests are also reported in the next chapter.

Chapter 7

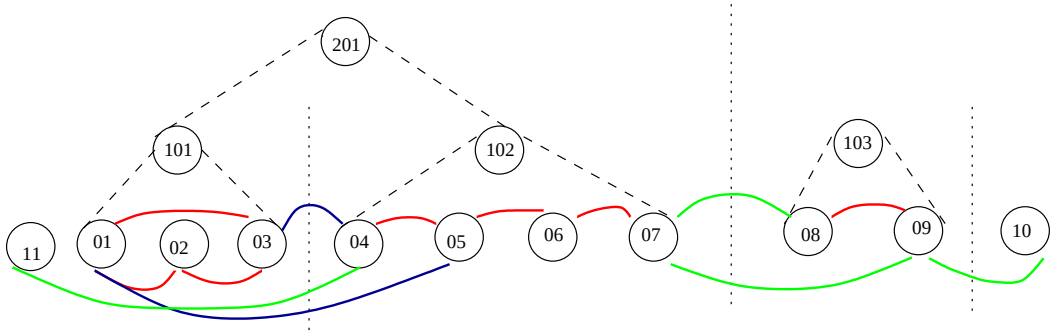
Application of Graph Nodeflex

As noted previously, any edge-map whose values have an implied order can be interpreted as a nodeflex of the graph, and therefore a particular aggregation of the original graph. Such an edge-map may simply be based on some value (label) associated with an edge. Alternatively, an aggregate graph element may be defined by explicitly selecting a set of connected subgraphs, which in turn will define a corresponding edge-map/nodeflex. The algorithm for extracting the aggregate node tree, the *boundary node* table and the aggregate map of a nodeflex is given here. This algorithm is specified using *order enforcing list* of Section refalg-specification.

black > yellow > green > blue > red

Two applications of nodeflex are also given. Firstly, graph nodeflex is applied to a transportation network and then used to find shortest paths. The aggregation is used to efficiently eliminate irrelevant edges during a search. The transportation network in Figure 7.2 is used to demonstrate this application. Secondly, it is suggested that nodeflex can help to aggregate population data over a graph. This can be used as an alternative approach to aggregation of population data in spatial analysis. Some additional operations for handling population data are also given. These are: an aggregated view of a graph at a fixed level relative to the values in the nodeflex edge-map, and deriving a user defined view based on nodeflex.

In this chapter colours are used to show edge-map values of line graphs; these and their meaning in terms of relative magnitudes are shown above.



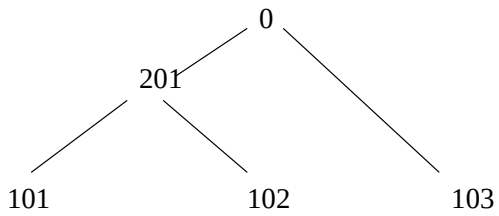
(a) Node Hierarchy

Edge list		Map	Aggregate Node
01	02	0	101
01	05	1	201
02	03	0	101
03	04	1	201
04	05	0	102
05	06	0	102
06	07	0	102
07	09	2	0
08	09	0	103
08	07	2	0
09	10	2	0
03	01	0	101
11	04	2	0

(b) Nodeflex

Aggregate Node	Boundary Node	Level
101	03	1
101	01	1
102	04	1
102	05	1
102	07	1
103	08	1
103	09	1
201	07	2
201	04	2

(c) Boundary Node Table



(d) Aggregate Node Tree

Figure 7.1: Extracting Aggregation from Nodeflex

7.1 Algorithm for Extracting Aggregation from Nodeflex

Figure 7.1(a) shows a line graph with relative edge-map values depicted by the colour of the edge. First three columns of Figure 7.1(b) also shows the same graph with corresponding edge-map values 0–2 representing red–green. This is therefore a nodeflex of the graph shown. The fourth column of Figure 7.1(b) and Figures 7.1(c) and 7.1(d) represent data structures that could be extracted from the nodeflex. Figure 7.1(d) is the aggregate node tree of the nodeflex. Figure 7.1(c) represents the boundary node table for the aggregate nodes in the nodeflex. The fourth column of Figure 7.1(b) assigns to the edge the lowest level aggregate node to which it belongs. This will be referred to as the aggregate map.

The algorithm given below extracts the aggregate node tree, the boundary node table and the aggregate map from a given nodeflex. Two applications of these data structures are given in the next two sections.

This algorithm uses the specification style of Section 4.4. It uses an order enforcing list, where the order is based on the nodeflex value of the graph edge. It also imposes the single get restriction on the graph nodes .

Initialisation:

1. Set up the order enforcing list to hold edges whose order is based on their nodeflex (edge-map) values.

2. Set up the single get restriction for the graph nodes.

3. Determine the maximum and minimum values in the edge-map:

   `min-map, max-map`.

4. Set up three collections to hold start-node, end-node, and edge-map value triplets corresponding to: boundary node table, aggregate node tree and another temporary table for boundary node entries:

   `pseudo-edges`, `aggregate-tree`, and `temporary-pseudo-edges`.

5. Set up a collection to hold the currently processing identifiers of the aggregate nodes. This will hold identifiers corresponding to the edge-map values from `min-map` to `(max-map - 1)`. Assign new identifiers to this collection. Set up a parallel collection to hold a `true-false` flag to indicate if an aggregate node at the corresponding edge-map value has been encountered. Set all the flags to `false`:

   `aggregate-node-IDs`, `aggregate-node-flags`.

6. Set up an edge-map for the graph to hold the aggregate map of the nodeflex:

   `aggregate-map`.

7. Select any edge in the graph: `current-edge`. Set `current-map` to the edge-map value of this edge.

Main algorithm: Steps to be repeated until processing is complete.

1. Set the `cur-node` to be the end node of the `current edge`. Get all the edges from the `current-node` and set their edge-map values. Let the minimum and maximum edge-map values of all these edges be `emin, emax`.

2. For each edge from the previous step, check the `aggregate-node-flags` collection corresponding to its edge-map value. If this is `false` then set it to `true`.

3. Add all the edges from Step (1) to the order enforcing list.

4. If `emin` is not equal to `emax` from Step (1) above, then this node is a *boundary node*. In this case add pseudo-edges to the `temporary-pseudo-edges` corresponding to all edge-map values from `emin` to `(emax - 1)`. The start-node of the pseudo-edge is the entry in the `aggregate-node-IDs` corresponding to the edge-map value, and the end-node is the `current-node`.

5. Pop an edge from the order enforcing list. From the `aggregate-nodeIDs` list determine the aggregate node corresponding to this edge's nodeflex value and write this into the `aggregate-map`.

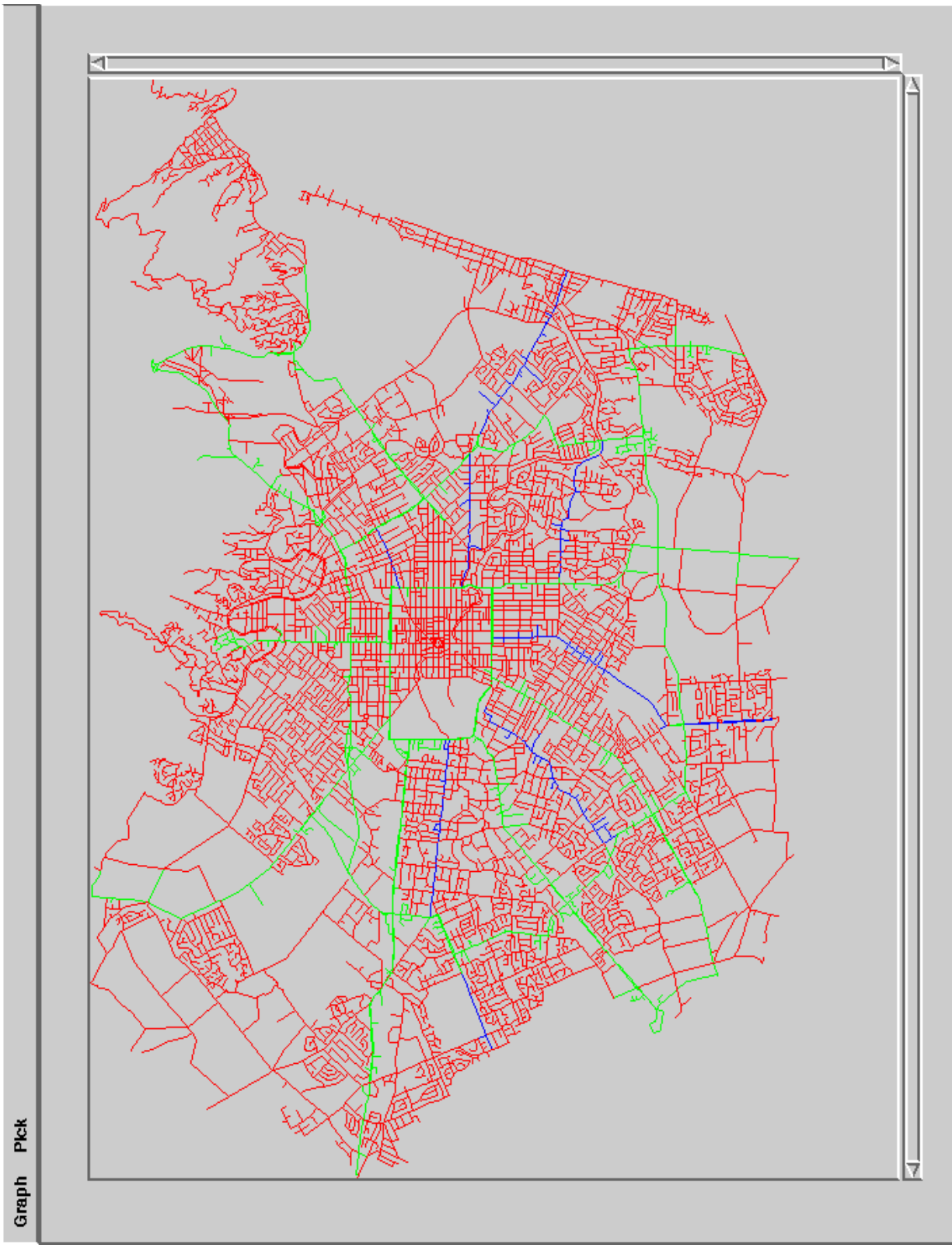


Figure 7.2: Nodeflex of a Transportation Graph

6. Continue popping until an edge to an unprocessed node is popped (single get). Set this to `cur-edge`. *If the order enforcing list becomes empty then processing is complete.* If the edge-map value of the popped edge (`popped-map`) is greater than the `current-map` then the traversal of an aggregate node has been completed; in this case perform the next step.

7. - Select all the pseudo-edges from the `temporary-pseudo-edges` for each edge-map value from `current-map` to `(popped-map - 1)`. Remove these from the `temporary-pseudo-edges`. For each selected pseudo-edge, test if the `aggregate-node-flags` contains a `true` value for the corresponding edge-map value. If so, add it to the `pseudo-edges` after incrementing its nodeflex value by 1.
   - By searching from top down in the `aggregate-node-IDs`, locate the parent aggregate node of the aggregate node whose traversal was just completed and add an entry to the `aggregate-tree` collection representing the corresponding edge in the aggregate node tree.
   - Create a new identifier and enter it into the `aggregate-node-IDs` corresponding to `cur-map`. Also set the corresponding `aggregate-node-flags` to `false`. Set `current-map` to the edge-map value of the popped edge.

Computational Effort:

The computational effort of the algorithm can be measured in terms of the number of times the edges of the graph are accessed. Let the number of edges be n. Steps 1–2 and 4–6 require $O(n)$ computations. Each execution of Step 3 enforces an order and thus it takes $O(n)$ computations. Each step of Step 7 also takes $O(n)$ computations for transferring the pseudo-edge. Thus the entire algorithm can be executed in $O(n^2)$ computations.

7.2 Shortest Path in Transportation Network

One application of graph nodeflex concept is given in this section. It uses the algorithm of the last section to extract the aggregate node tree and the aggregate map from a nodeflex and uses these two data structures to improve the computational time required to find shortest path in a transportation network.

The classical algorithm for finding shortest path between a pair of nodes is the Dijkstra's algorithm which requires $O(n^2)$ computations, where n is the number of edges in the graph. Improvements to Dijkstra's algorithm for special cases of graphs have been proposed. For a class of sparse networks including planar graphs, Johnson [Joh77] shows that, by using binary tree instead of a sorted list to store the weighted nodes during the execution of Dijkstra's algorithm, the computational time can be bounded by $O(n)$. A^* algorithms are another approach to finding shortest paths [Nil82]. These are heuristic algorithms that uses a heuristic estimator function to estimate shortest path cost and uses this estimate to determine the next node that should be examined. In the worst case scenario it executes in $O(n^2)$ time. However, good heuristic estimators can yield better time. Another approach to shortest path searching was suggested by Frederickson [Fre91]. This preprocesses the graph to build data structures, called compact routing tables, which capture information related to shortest paths. Another approach known as hierarchical shortest path algorithms partitions the graph into two levels, a boundary graph and fragment graphs [SFG97]. In this approach some selected shortest paths based on this partition are pre computed. The cost of these pre computed values are stored without storing the actual path. Hierarchical shortest path algorithms exploit these pre-computed values to determine the exact shortest path between any two nodes.

A transportation graph consists of streets with a varying capacity for carrying traffic. Therefore the edges have graded values [Eur95]. van Eck and de Jong [vEdJ90] suggest an algorithm that exploits this grading to eliminate irrelevant edges while path finding. This method consistently selects edges with greater values. However it does not guarantee the correct shortest path at all times.

Car suggests Hierarchical Spatial Reasoning (HSR) approach to path finding over

transportation network. Car states that HSR is a method of spatial problem solving that uses hierarchy to infer spatial information and draw conclusions [Car98]. Essentially HSR consist of a hierarchical structure that is derived from a non hierarchical structure using a given method and a set of rules on how to reason about them. The road classification criterion of transportation network is used to extract a subgraph from a transportation network. This gives the hierarchical structure. Dijkstra's algorithm is then used to find the path over this structural hierarchy with some additional rules. These rules specify which part of the graph to use, when to switch between the levels, and how to combine the partial results into a complete solution.

An approach based on graph nodeflex is developed here for improving shortest path algorithms over transportation graph. The grading of the edges of a transportation graph can be used as edge-map values that define a nodeflex over the transportation graph. From this an aggregate node tree and an aggregate map can be extracted using the algorithm of the last section. *A path over a graph, from a source node to a destination node, can now be translated into a path over the aggregate node tree.*

For example in Figure 7.1: to determine the shortest path from node 01 to node 10, first determine the aggregate node for these two nodes as follows. Determine the edge from the node with the lowest nodeflex value and then determine this edge's aggregate node using the aggregate-map. The aggregate node of the node 01 is 101, and that of the node 10 is the aggregate node 0, which is the root of the aggregate node tree. Thus a path from 01 to 10 is translated into the aggregate node path; 101–201–0.

The aggregate node path gives a set of aggregate nodes. During path finding all edges whose aggregate-map value is not in this set are ignored. Typically this is a small set because the number of levels in a transportation graph nodeflex is less than seven. In this case the maximum number of aggregate nodes in any aggregate node path is fourteen. This approach thus avoids examining irrelevant edges while determining the shortest path.

This method guarantees the correct shortest path determination under the following assumption. *The approach makes the assumption that the shortest path between two*

base graph nodes within an aggregate node can be found by searching the subgraph corresponding to that aggregate node. This assumption may not produce the exact shortest path at all times. It will, however, produce one of the best paths at all times. Any shortest path finding algorithm can be adopted for use with this scheme.

7.2.1 Splitting Subgraphs

Nodeflex defined by the graded values of the edges of a transportation graph will not result in the optimum graph aggregation for use in the scheme. Some data preparation is needed on the nodeflex values so that the subgraphs defined in the nodeflex are optimal. This will involve splitting the subgraphs further by shifting some edges to subgraphs at higher levels. Figure 7.3 demonstrates this, where, green coloured edges are higher than the red coloured edges.

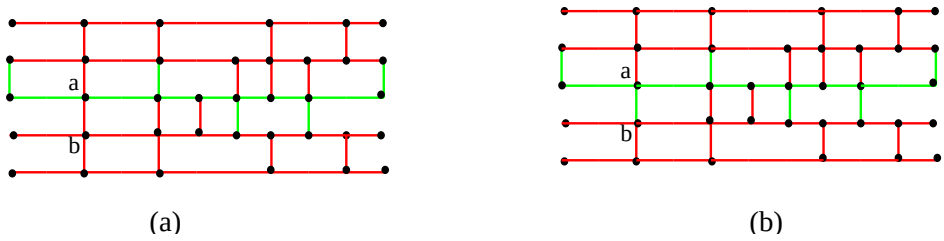


Figure 7.3: Splitting Subgraphs

It may be appropriate to consider Figure 7.3(a) as consisting of three subgraphs; the green subgraph, a red one above it, and another red one below it, resulting in an optimal aggregation for the shortest path problem. However the two red subgraphs form a single connected subgraph. In Figure 7.3(b), the edge a-b is changed into a green coloured edge, thus the red subgraph in Figure 7.3(a), has been divided into two connected red subgraphs. Manual preparation is therefore needed to achieve optimum hierarchy for path finding. This investment in manual preparation can be justified by the efficiency gained in finding shortest paths. Failure to remove connectedness where possible will only result in a reduced level of efficiency during path finding. Thus the transportation graph data could be incrementally improved as resources become available. Every time the nodeflex is modified, the algorithm of Section 7.1 must be executed to extract the altered aggregation data structures.

7.2.2 Results

Figure 7.2 shows the edge-map values of a transportation graph in different colours. The main transportation graph consists of nearly 1000 edges. Red coloured edges have the lowest edge-map value. The values of blue coloured edges are greater and green coloured edges have the highest edge-map values. A connected subgraph of red coloured edges that are bounded by blue or green coloured edges forms an aggregate node at the first level. Similarly, a connected set of red or blue coloured edges bounded by green coloured edges form an aggregate node at the next level. (Note: these edge-map values were manually assigned using an interactive graphical interface which is described in Chapter 9.) This generated an aggregate node tree with 51 nodes. Transportation graphs with similar graded values are commercially available.

Figures 7.4 and 7.5 show the edges traversed (white coloured edges) during the execution of Dijkstra's shortest path finding algorithm using the nodeflex in Figure 7.2. In Figure 7.4 the source and destination aggregate nodes are both at the first level of aggregation. These two aggregate nodes were searched, as well as the top level node which consists of the green coloured edges (Figure 7.2). Thus the erroneous search indicated by the white coloured edges in the lower part of Figure 7.4 is due to searching the top level node indicated by the green coloured edges in Figure 7.2. In Figure 7.5 some blue edges (Figure 7.2) were also traversed because the source and destination aggregate nodes were *contained* by an aggregate node at this level.

A greater number of aggregate nodes will reduce the number of edges in each node, thus improving the elimination process. However, when all the aggregate nodes are generated at a single level, the number of edges at the top level will increase. This in turn increases the number of edges that must be traversed. Thus a limiting condition exists on the number of aggregate nodes that can be introduced at a single level. By increasing the number of levels a more effective elimination will result, but the number of aggregate nodes in the aggregate node path also increases. As these increase, more computational effort is required in the edge selection process that uses the aggregate path. This, again, creates a limiting condition on the number of levels that can be introduced.

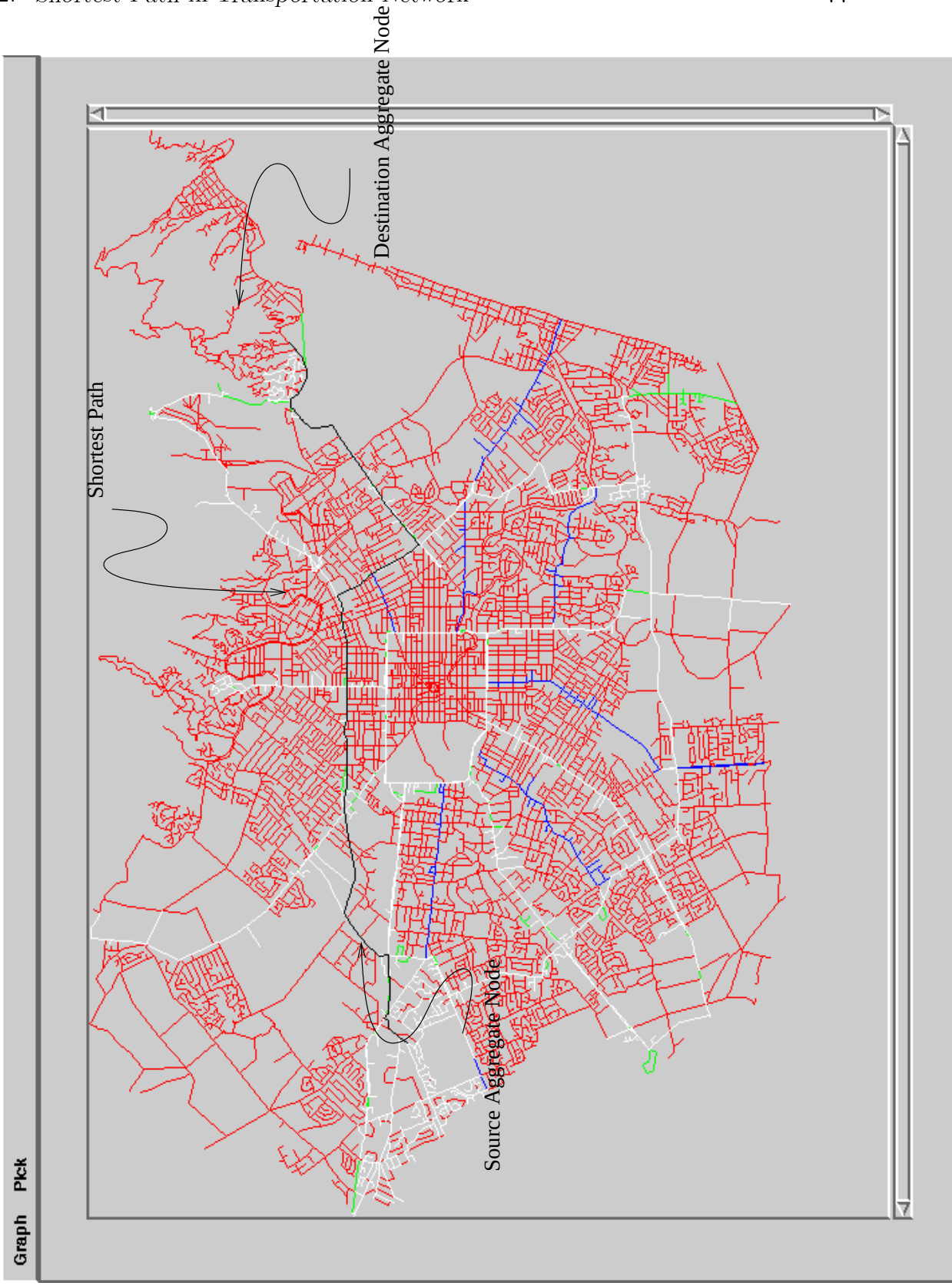


Figure 7.4: Result of Shortest Path Search

Computational Effort

The computational effort can be estimated based on the number times graph edges are accessed.

Let n_g be the number of edges in the base graph.

Any shortest path algorithm will execute in $O(n_g^2)$ time.

Let T_a be the time required to test each edge for inclusion against the aggregate node path.

Let T_p be the time required to determine the aggregate path over the aggregate node tree.

Let n_i be the number of edges examined in the nodeflex based approach.

Any shortest path algorithm based on the nodeflex approach will execute in:

$O(n_i^2) + n_g \times T_a + T_p$.

If the base graph is planar, then based on the estimate given in [Joh77], it will execute in:

$O(n_i) + n_g \times T_a + T_p$.

7.2.3 Nodeflex Over Directed Transport Graph

The above transport graph example is an undirected graph. The same approach can be used with directed transport graphs. In this case the nodeflex will interpret the graph as undirected for the purpose of creating the aggregate tree and the aggregate map. However, the algorithm used to determine the shortest path will treat the transport graph as directed. As long as the assumption observed in Section 7.2 holds, that is the shortest path between two base graph nodes within an aggregate node can be found by searching the corresponding subgraph alone, this approach will yield correct result.

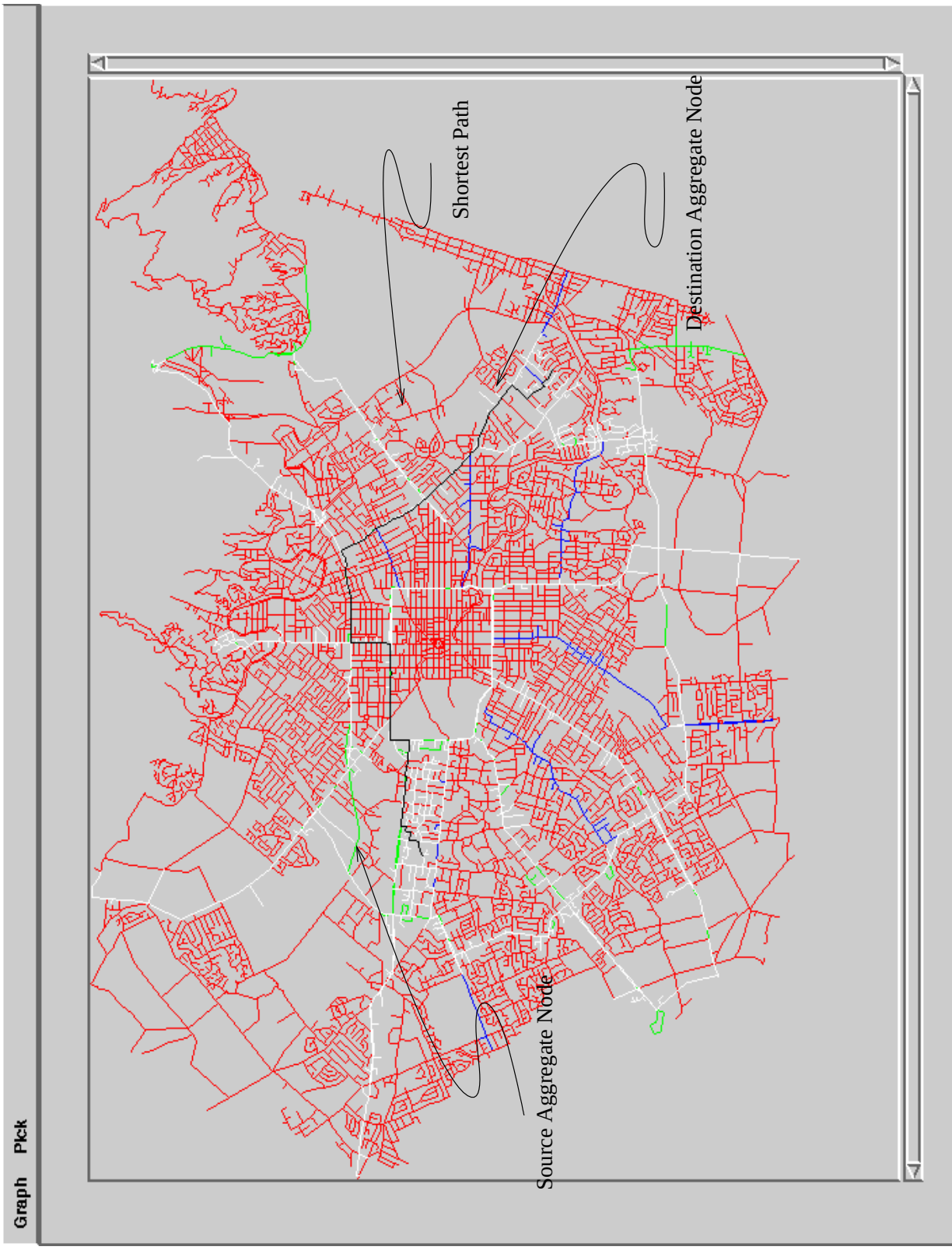


Figure 7.5: Result of Shortest Path Search

7.3 Aggregation of Population Data

The street network contains the population and service distribution. One advantage of representing this distribution as a graph was mentioned in Section 4.5. In this representation, the edges of the graph will be labelled with a population value. In Chapter 4 it was suggested that zones based on population could be represented as subgraphs of the population graphs. In this case, zone overlap can be more efficiently computed through subgraph overlap, thus avoiding the computation intensive polygon overlays.

Nodeflex can also be applied to a population distribution represented as a graph. This has applications in spatial interaction modelling, which requires flexible aggregation of the population data. Typically, population distribution represented as polygonal data is aggregated by merging adjacent polygons [FDC95]. Nodeflex can be used as an alternative approach. Multiple levels of a nodeflex could be used to evaluate interaction models at different levels of aggregation. This approach is unusual, possibly due to a lack of adequate functionality in existing commercial software to facilitate such a view, or perhaps because this application domain is not as commercially profitable as other graph based GIS application domains such as facility management.

Each aggregate node of a nodeflex aggregates the population data along its edges. A particular view of the graph then consists of the aggregated population data at these nodes and the remaining edges connecting these nodes. Population data along those edges that are not aggregated into a node at a given level could be transferred to an appropriate aggregate node. A traversal of such a graph can be simulated by combining the main graph, nodeflex and the pseudo-edges represented by the *boundary node* table (Section 6.3.1). This approach will therefore depend on the resolution of the following issues.

- Firstly, the population data must be made available as data distributed along the edges.
- Secondly a nodeflex that could be used for the aggregation of population data must be created. This could be manually created through the provision of a

user interface. Alternatively it could be generated based on some rules.

- Thirdly, a scheme must be developed to aggregate the data along those edges that are not part of an aggregate node.

Modifiable Aerial Unit Problem (MAUP) in spatial interaction modelling has been well documented [FDC95]. MAUP is caused by the errors introduced by aggregating adjacent polygonal zones into larger zones which are then considered as nodes of a graph. Algorithms over such graphs are used to deduce the spatial behaviour of the population. The approach to population representation and aggregation described above, and its effect on MAUP, may be worth investigating. Participation of an expert in interaction modelling is needed to develop this proposal fully and test its potential.

The following two sections outline two forms of traversals over an aggregate graph. These are traversal at a fixed nodeflex level and traversal over a user defined view. Both could be used to simulate spatial interaction.

7.3.1 Aggregated View at a Fixed Level

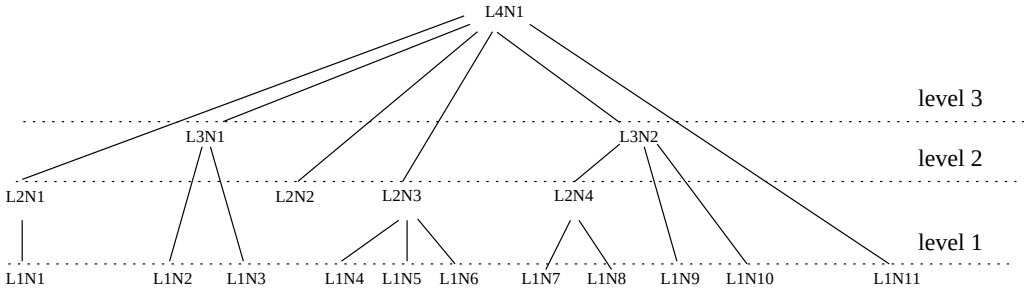


Figure 7.6: Aggregate Node Tree

A nodeflex can be viewed as a graph at a specific aggregation level n_i. In this case the graph will consist of: all the edges from the base graph with nodeflex value $\geq n_i$, aggregate nodes at level n_i, and aggregate nodes at lower levels that are not aggregated into a node at level n_i. For example, see Figure 7.6, which represents an aggregate node tree. When traversing at level two the aggregate nodes that will be seen are (L2N1, L3N1, L2N2, L2N3, L3N2, L1N11).

Traversal of this graph can be simulated using the base graph, nodeflex edge-map, and the pseudo-edges from the boundary node table. Nodeflex edge-map is used to select the edges of the base graph. Boundary node table is used to select the aggregate nodes. Note that the level of each aggregate node is also stored in the boundary node table.

If a pseudo-edge from the boundary node table is returned as an out-edge from a node, this implies that the node is a boundary node to some aggregate node. The following test will select the appropriate aggregate node for the current traversal level.

Let agg_{level} be the level of each aggregate node with a given node as one of its boundary. Level of the selected aggregate node is given by:

$max(agg_{level} \leq n_i)$.

That is the level of the selected aggregate node is the maximum level of all aggregate nodes of this node that are less than or equal to the level being traversed (n_i). This ensures that the aggregate node seen by a traversal is as described above.

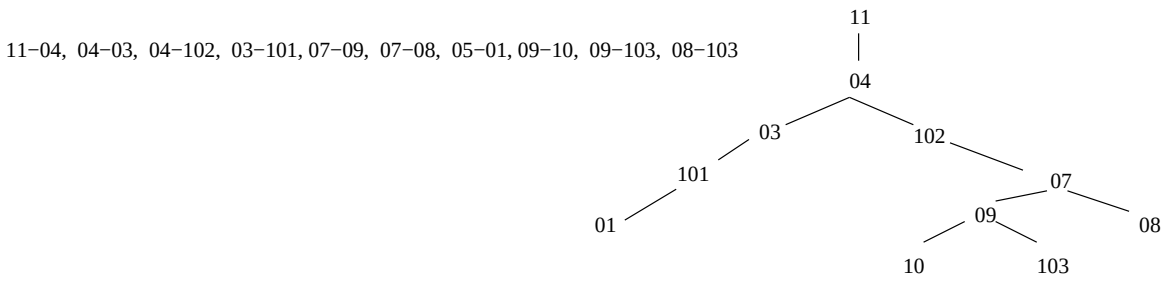
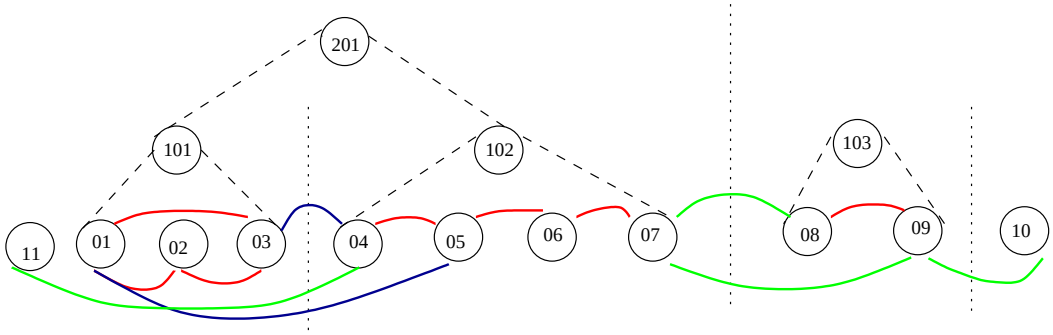


Figure 7.7: Order Enforcing Queue and Traversed Tree

A graph traversal at level one for the graph in Figure 7.8 is shown in Figure 7.7. Traversal uses queue as the order enforcing list with single get restriction on nodes. Therefore the resulting traversed structure will be a tree.



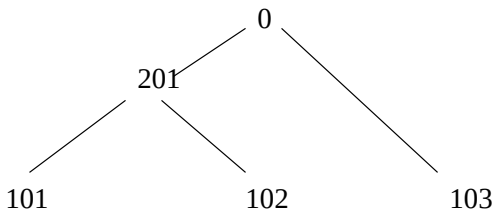
(a) Node Hierarchy

Edge list		Map	Aggregate Node
01	02	0	101
01	05	1	201
02	03	0	101
03	04	1	201
04	05	0	102
05	06	0	102
06	07	0	102
07	09	2	0
08	09	0	103
08	07	2	0
09	10	2	0
03	01	0	101
11	04	2	0

(b) Nodeflex

Aggregate Node	Boundary Node	Level
101	03	1
101	01	1
102	04	1
102	05	1
102	07	1
103	08	1
103	09	1
201	07	2
201	04	2

(c) Boundary Node Table



(d) Aggregate Node Tree

Figure 7.8: Aggregation Hierarchy

Traversal starts at node 11. Edges from node 11 are 11–04. The level of this edge in the nodeflex edge-map is 2 and therefore added to the queue. Traversal proceeds by popping edges from the queue and adding out-edges from the reached node to the queue.

Edge 11-04 is popped from the queue. Edges from node 04 are: 04-03 from base graph, and 04-102 from boundary node table. Nodeflex value of edge 04-03 is 2 which is greater than the traversal level one and therefore accepted. Pseudo-edge 04-102 is of level one and therefore accepted based on the test above. Both these edges are added to the queue. Edge 04-03 from the top of the queue is traversed next and edges from node 03 are pseudo-edge 03-101, which is added to the queue. Pseudo-edge 04-102 is now at the top of the queue. Traversal across an aggregate node is enforced through pseudo-edges as follows.

When a pseudo-edge is traversed, all other pseudo-edges of that aggregate node are also traversed simultaneously. Thus when pseudo-edge 04-102 is traversed, pseudo-edge 102-07 is also traversed and the edges from the boundary node 07 are added to the queue.

Compared to the original graph, the number of edges of an aggregate graph is reduced by the number of edges aggregated into nodes. Some additional edges are introduced, however, as pseudo-edges. When the number of edges aggregated into nodes is substantially greater than the resulting number of pseudo-edges, the computational effort needed to traverse the aggregate graph will be much less than that required by the original graph.

7.3.2 Creating a User Defined View

In addition to the fixed level view as above, a more flexible user defined view of the nodeflex can be selected. This is captured in a copy of the nodeflex which is then modified to reflect the desired view. The aggregate node tree and the *boundary node* table, together with the user defined nodeflex copy, will represent this view.

Figure 7.9 (a) is a graph with nodeflex values shown in colours. An initial nodeflex for a user defined view is shown in Figure 7.9 (b). This view presents the immediate

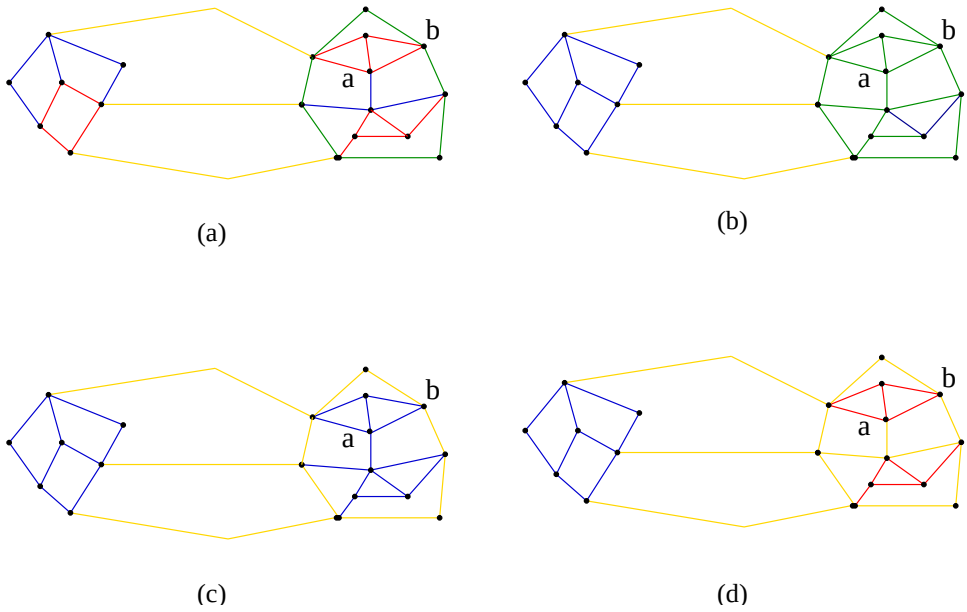


Figure 7.9: Nodeflex and User Defined Views

children of the root of the aggregate node tree. In the standard nodeflex a subgraph at a particular level includes all edges with nodeflex values less than or equal to the given level. In a user defined view all subgraphs are defined with a single nodeflex value. Only those edges with the highest nodeflex value in this nodeflex copy are used for traversal.

An aggregate node in this view can be expanded or condensed by selecting an edge of that node. The following steps will modify the user defined view:

1. Let the edge-map value of the selected edge in the nodeflex copy be e_s.
2. Let the traversal edge-map value be e_m.
3. When expanding, set $e_m = e_s$.

   When condensing, set $e_m = (e_s + 1)$.
4. The nodeflex copy representing the user defined view is now modified by traversing the original graph starting from the selected edge. The traversal will be restricted to those edges whose edge-map value is $\leq e_m$ in the original nodeflex. The corresponding edge-map values in the nodeflex copy are modified as follows.

5. When *expanding*, replace all those edge-maps whose value is equal to e_m in the original nodeflex with the highest edge-map value. Replace all other edge-map values in the copy with $(e_s - 1)$. Figure 7.9(c) shows the nodeflex values that result when *expanding* by selecting edge a–b in Figure 7.9(b). Figure 7.9(d) shows the nodeflex values that result when *expanding* again by selecting edge a–b in Figure 7.9(c).

   When *condensing*, replace all the traversed edges with e_m. If edge a–b in Figure 7.9(c) is selected for *condensing* then the nodeflex values will revert to those in Figure 7.9(b).

Traversal of the user defined view is based on the nodeflex copy corresponding to the view developed by the user. The original nodeflex edge-map will not be used. Since the level of all edges of each subgraph in the user defined view is the same, this is used to select the correct pseudo-edges from the *boundary node* table.

7.4 Summary

In the last two chapters a theory of nodeflex and a method of defining it were developed. This chapter has presented an algorithm for deriving the data structures of graph aggregation from a nodeflex definition. These data structures are: nodeflex, boundary node table, aggregate node tree and aggregate map.

- One application of nodeflex was demonstrated on transportation graphs, where path finding algorithms are modified to use the aggregate-tree and the aggregate-map to effectively eliminate irrelevant edges.
- Another application of nodeflex for aggregating population data was also suggested. Two approaches are possible: the nodeflex and the *boundary node* table can be used to traverse the aggregate graph; and a nodeflex copy can be used to develop a user defined view and it can be traversed by using this copy and the *boundary node*.

Chapter 8

Topology of Graphs in Space

The nodeflex theory developed in the last three chapters has applications for graphs in space. This chapter will examine the representation of these graphs in space.

The model of vector based spatial data in Section 2.1.2 captures 0-cell, 1-cell and 2-cell spatial objects. However, graphs in space are a subset of this model, being formed by 0-cell and 1-cell objects only. Considering only this subset, thus excluding the representation of 2-cell spatial objects, results in some useful simplifications. The requirements of the representation are:

- A representation that can be extended with the addition of 1-cell spatial objects.
- Selection of graphs that consist of a subset of the entire graph formed by all the 1-cell spatial objects.
  - It should be possible to define paths over a selected graph and apply a *Chainage* pattern on this selected graph.
  - Create nodeflex over a selected graph.

This type of extension to graphs in space is useful for evaluating new developments in an urban context. However such an application has not been developed as part of this work due to time restrictions.

8.1 Representation with Topological Model

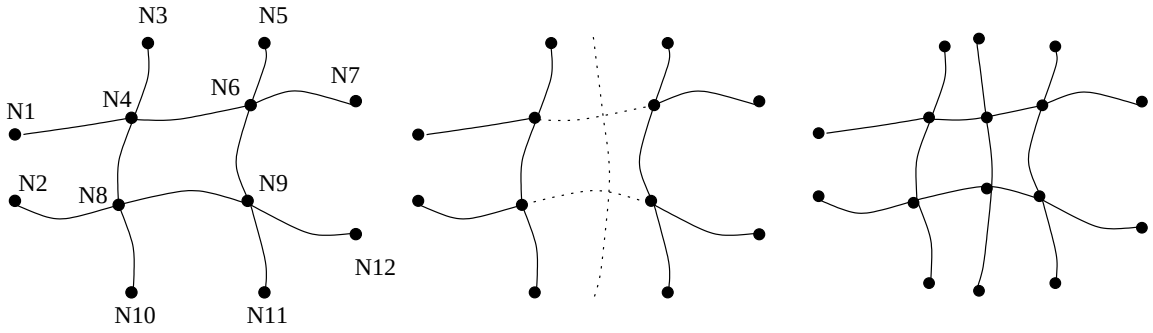


Figure 8.1: Adding a Polyline

Spatial data models are affected by the finite representation of numbers. The topological model of Egenhofer, Frank and Jackson [EFJ89] and the Realm model of Güting and Schneider [GS93] are attempts to eliminate the finite representation problem. In both cases this is achieved by determining all points of interest in space at the time of model building. All other spatial entities are built on these predetermined points. This ensures that no further computation is needed to determine spatial relations.

The model proposed in [EFJ89] for representing vector based spatial data is based on the theory of topology. Spatial objects have a spatial dimension. A point is a 0-D spatial object. A line is a 1-D spatial object, and so on. For each dimension, the minimal object in that dimension is called an n-simplex. A point is 0-simplex, a straight line is 1-simplex, a triangle is 2-simplex, and so on. Each n-simplex has (n+1) (n - 1)-simplex faces. Thus a line which is 1-simplex has two end points which are 0-simplex as its faces. A triangle which is 2-simplex has three bounding lines which are 1-simplex as its faces and so on. A simplicial complex is a collection of simplices and their faces that satisfy the following condition: *If the intersection between two simplices in this collection is not empty, then the intersection is a simplex which is a face of the two intersecting simplices.*

For example, in the case of a 2-D simplicial complex, this says that all 0-simplices in the complex are an end point of one or more 1-simplices in the complex. The 1-simplices touch each other at their end points and otherwise do not intersect. All 1-simplices in the complex are faces of either one or two 2-simplices and the two

simplices touch each other at these faces and otherwise do not overlap.

To represent graphs in space, 2-simplices and the representation of regions are not relevant. Thus the representation is reduced to 0-simplices (0-D) and the polylines between them. Adding a polyline to the representation is shown in Figure 8.1.

8.1.1 Extension and New Graphs

Egenhofer, Frank and Jackson [EFJ89] notes that to maintain a consistent simplicial complex and to maintain the definition of new simplices, database capabilities are required. A database system could also be used to maintain the representation of the extensible graph in space. When a new polyline is added, some existing polylines will be split. The database that maintains the extensible graph in space therefore needs to update these dependent definitions. The old polylines need to be retained in the database if there are graph edges that refer to it. As the edges (polylines) are split by the addition of polylines to create new graphs with new edges, the relationship between the old edges and the new edges will be represented by a tree structure (*Composite* pattern), which must be maintained so that the old graphs and the new graphs can be spatially related.

The simplicial complex approach for representing 2-D space, and the equivalent approach to represent extensible graphs in space, is flexible and simple. However, a database type facility is needed to maintain the representation under modifications. The next two sections describe two other approaches.

8.2 Polyline Based Representation

The approach of the last section splits polylines into their smallest units. The polyline based approach described in this section exploits the ease of computing a new polyline from the original polyline. Thus instead of splitting the original polyline, it computes new polylines based on some minimal set needed to represent the extensible graph in space. For example, Figure 8.2 consists of four polylines L1–L4 which can be used to represent all graphs in space that can be derived from

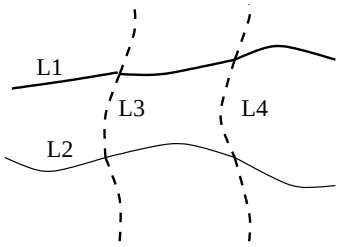


Figure 8.2: Polyline Based Extensible Graph in Space

them.

The following definitions are used for developing the representation:

`Lso:` is a specific class of polyline (Linear Spatial Object). It is an ordered series of non-intersecting and touching line segments. *Lso* is considered to have a direction that is arbitrarily assigned to it to facilitate the definition of points along it. Each *Lso* is identified by its unique identifier.

`Lso-Point:` is a point on an `Lso`, defined as an offset from the start of it.

`End-Point:` is an `Lso-Point` that is also one of the end points of that `Lso`.

`Int-Point:` is an `Lso-Point` that is the intersection point of two or more `Lso`s.

The `Topology` of extensible graphs in space is the one-to-many relationship between the `Node`s, the `End-Point`s, and the `Int-Point`s on different `Lso`s (Figure 8.3a).

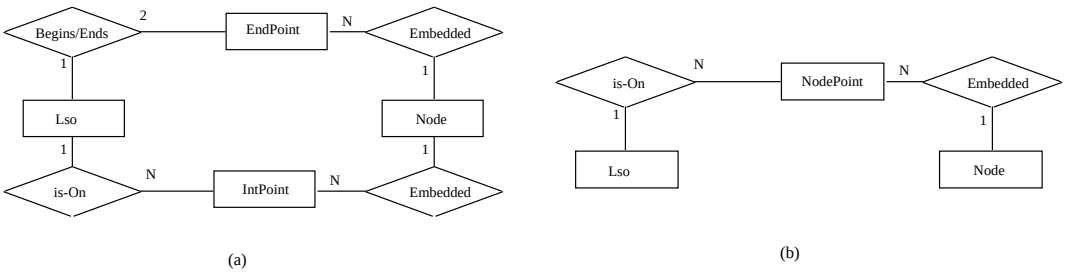


Figure 8.3: Polyline Based Representation

The `Int-Point`s and the `End-Point`s can be maintained in a single collection as `Node-Point`s. The basic representation thus consists of:

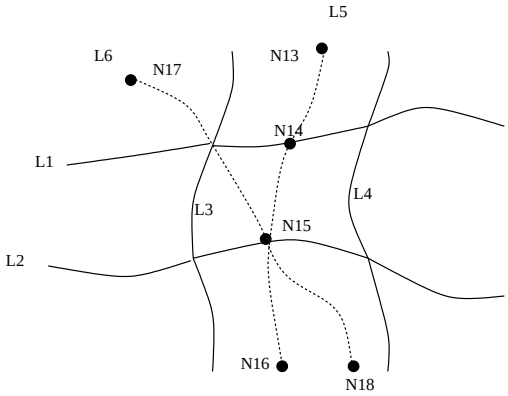


Figure 8.4: Extension in Polyline Based Representation

- A collection of `Lso`s
- A single collection of `Node-Point`s consisting of `End-Point`s and `Int-Point`s
- A one-to-many relationship between the `Node`s and the `Node-Point`s (Figure 8.3b).

8.2.1 Extension and New Graphs

Adding a new polyline involves updating the above three data structures. Geometric computations that will detect the new `Node-Point`s must check for the coincidence of these with existing `Node-Point`s. For example, the addition of two new lines L5 and L6 (Figure 8.4) results in the creation of seven new `Node-Point`s but only six new *Node*s (N13- N18).

From this basic representation any graph formed by a subset of the `Lso`s can be derived through the following procedure:

- Collect all entries in the `Node-Point`s that are defined on any of the `Lso`s that are to generate the graph relation.
- Sort the above collection. The primary key is the `Lso` identifier and the secondary key is the `Node-Point`'s offset value.
- Two consecutive entries in this collection, that are also defined on the same `Lso`, form an edge in the graph relation.

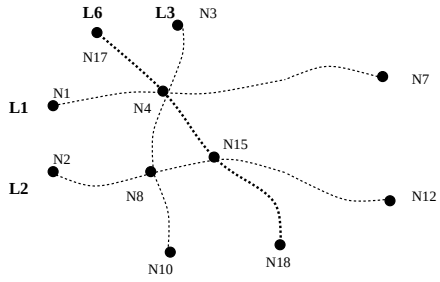


Figure 8.5: Subgraph in Polyline Based Representation

The graph in Figure 8.5 is formed by the `Lsos` (L1, L2, L3 and L6). The addition of new `Lsos` to the model does not affect existing graphs derived in this manner.

Each edge in such a graph is uniquely identified by the two node identifiers and the `Lso` identifier. For example the edge and its geometry from N1 to N4 is identified by the edge triplet $< N1, N4, L1 >$. Different graphs generated from different subsets of the `Lsos` can be related using this edge triplet.

8.3 Hybrid Representation

Extensible graphs in space can be used to model transportation and population graph. In these applications there exists a large underlying graph which needs to be extended and subgraphed. A more efficient method to represent such extensible graphs in space is considered here. This combines the standard representation of a graph as an edge-list to represent the base graph and the polyline based representation of the last section to extend it.

8.3.1 Base Graph

A base graph is a graph whose edges are `Lsos` defined between two nodes of the base graph. This base graph can be viewed as a collection of edges. Figure 8.6 shows the `Lsos`, B1–B12, forming a base graph. Each edge making up the base graph is defined as the edge triplet $< nodeID, nodeID, LsoID >$.

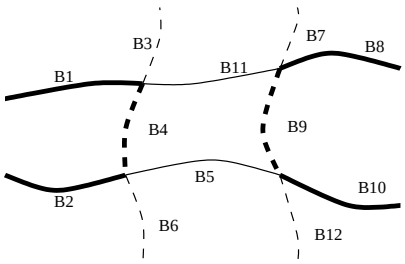


Figure 8.6: Hybrid Representation

8.3.2 Extension and New Graphs

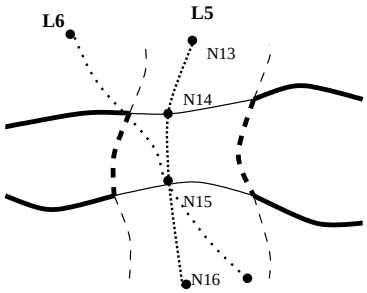


Figure 8.7: Extension in Hybrid Representation

The base graph, extended with additional `Lsos`, is represented using a scheme similar to the polyline based representation of the last section. Thus the same set of data structures are used in addition to the base graph. These are:

- A collection of additional `Lsos` that extend the base graph. This is the `Extension-Lso` collection. In Figure 8.7 the base graph is extended with two `Lsos` L5 and L6.

- A collection of `Node-Points`. An entry in this collection can be one of the following: the `End-Points` of the `Extension-Lsos` or the `IntPoints` that are the intersection of each `Lso` in the `Extension-Lso` collection with all the other `Lsos` in the model.

- A one-to-many relationship between these `Node-Points` and the `Nodes`. This is `Extension-Topology`.

The complete extensible graph is made up of the base graph, the `Extension-Lsos` and the `Extension-Topology`.

A new graph formed by a subset of the `Lsos` of this subspace can be represented in one of the following forms:

- Subgraph of the base graph.
- A graph formed only by the `Extension Lsos` which can be generated as in the polyline based representation.
- A graph formed by a mixture of `Lsos` from the base graph, and the `Extension-Lsos` are represented by a combination of the above two. Thus it consists of a subgraph of the base graph and an additional edge collection.

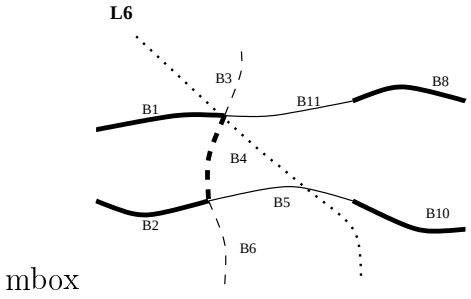


Figure 8.8: Subgraph in the Hybrid Scheme

A new graph generated by `Lsos` B1–B6, B8, B10, and B11 and the `Extension-Lso` L6, as in Figure 8.8, consists of the following: a subgraph of the base graph that includes only the selected base edges, less those edges intersecting with the selected `Extension-Lso` L6 and the additional edge collection consisting of the edge triplets.

8.4 Comparison of the Three Schemes

Three approaches for representing extensible graphs are discussed in this chapter. These are the topological model of a graph in space, the polyline based representation and a hybrid representation.

The topological model, when used for representing 2-D, space will consist of 0-simplices, 1-simplices and 2-simplices. This, however, is adapted to represent only 0-simplices and 1-simplices, in other words graphs in space. The following points can be noted:

- The model requires a database facility to manage the updates resulting from the addition of new polylines. As new polylines are added which intersect with existing polylines, these polylines will be split. The model must maintain the relationships between these split polylines and the original polylines.
- Each graph derived from this model must be created in some graph representation structure. The geometry of the graph edges are polylines derived from the model, which may be a concatenation of several polylines in the model.

  Since each derived graph is created in its own representation, nodeflex definitions can adopt the standard approach of using an edge-map of the created graph.
- Updates to the model by the addition of polylines will not affect these graphs.

The polyline based approach maintains a list of polylines and a table representing the relationship between these polylines and the `Node-Point`s of these polylines. The following points are noted about this approach:

- As new polylines are added, only the table of the `Node-Point` relationship needs to be updated. This is simpler than maintaining a database.
- Each graph derived from the model must be created in some data structure as in the topological representation.

  Nodeflex definitions can adopt the standard approach of using an edge-map of the created graph.
- Updates to the model will not affect these graphs.
- The main disadvantage is that, unlike the topological model where the polylines are already split into atomic units, the geometry of the graph edges must be recomputed from the underlying polylines of the model.

The hybrid method combines the two approaches, resulting in a scheme that best suits the representation of graph space in urban systems. The following points are valid for this approach:

- As new polylines are added only the table of the `Node-Point` relationship needs to be updated. However, this is very much smaller than the table needed in the polyline based representation.
- While new graphs deriving from the model can be created in some graph data structures, it is also possible to represent them as a subgraph of, and an extension to, the base graph.

  The edge-map technique for defining subgraphs used in the previous chapters can also be used to represent the subgraphs of base graphs in the hybrid representation. Only those slots in the edge-map corresponding to the eliminated edges in the base graph will be used. Therefore copies of this same edge-map can be used to define a nodeflex of these new graphs. To complete a nodeflex definition, a parallel edge-map for the newly added edges must also be generated.
- Updates to the model will not affect these graphs.

It can be seen that the hybrid model approach has several advantages when representing a graph that needs to be extended with additional polylines. The implementation of this proposal is described in Section 9.1.2.

8.5 Summary

An important requirement of spatial data representation is the capture of the topological relationships among the spatial attributes. This was described in Section 2.1.2, where 0-cell, 1-cell and 2-cell attributes were considered. In this chapter the topological representation of the reduced dimension of the graph space was examined.

Three approaches were considered and it was suggested that the hybrid data model

is best suited for real life situations, where the base graph is large and relatively constant. However some addition of new 1-cells must be considered.

The hybrid representation could employ edge-maps to define new graphs. A copy of the same edge-map can then be used to define nodeflex over the new graphs.

The proposal was tested in the implementation described in the next chapter. Applications of this hybrid model could not be tested as part of this work.

Chapter 9

Implementation

Graph nodeflex was tested in a purpose-built system using Smalltalk VisualWorks version 2.0. This was chosen for its rich framework for handling graphics, and for the object oriented environment with the potential for software pattern mining. Any GIS, be it a commercially available one or a purpose built one as above, consists of four main components:

1. A data model for capturing geometry and topology.
2. A graphical user interface and display with zoom, pan, browsing, etc.
3. A spatial indexing scheme for managing access to the data.
4. A query and analysis component with operations for handling problems in the application domain.

The system that was built to test the proposals included the following components:

1. A simple database management using text files to store and access graphs and associated data. The planar graph extension of Chapter 8 was also easily tested with this.
2. A graphical user interface for display and interaction with graphs. This was used to edit edge-maps associated with graph nodeflex and was also used for testing purposes.

3. A new type of indexing scheme was implemented and tested in the display and interaction module above.

4. The graph nodeflex was implemented and tested on path finding algorithms over transport graphs

The implementation of these components can be related to the software patterns described in Chapter 2.

9.1 The Data Model

The data model consists of a graph represented as an edge collection. An index file based on the node identifier is used for fast access in determining the in-edges and out-edges from a given node. When the graph is undirected each edge is stored only once and is doubly indexed in the index file by the two nodes of the edge. All of these are maintained and operated on as text based files.

The hybrid model of Chapter 8 is also represented as three additional text files: one representing the polylines of the base graph edges, another file representing the added polylines that extend the base graph, and a third file holding the pairs $< nodeID, NodePoint >$ of the `Extension-topology`.

An instance of the `GraphAccess` class is used to control all access to the text files, and manages all external file access and name handling. This is an example of the *Proxy* pattern (Section 2.3.2) where a single object is used to guard all access to another system.

9.1.1 Edge-map

Nodeflex is based on an edge-map, which is some mapping of values to the edges of graphs. In the transportation and the population distribution domain the underlying graph is formed by the streets and highways. When the assumption is made that the edge set over which an edge-map is to be defined is static, some simplification results

in the manner in which mappings can be maintained on disk. Some preferred order can be imposed on the edge collection, which automatically assigns a local identifier to each object in the collection corresponding to the position of that object in the imposed order. If it is also assumed that the values recorded in a particular edge-map occupy a fixed length of storage, accessing an edge-map value for a given object based on its positional value becomes a direct access to a location. No searching or indexing will be required to access the mapping values.

An edge-map is implemented based on this observation as a separate file parallel to the edge collection. The user interface described later is used to maintain the edge-maps.

9.1.2 Aggregated Graph and Extended Graph

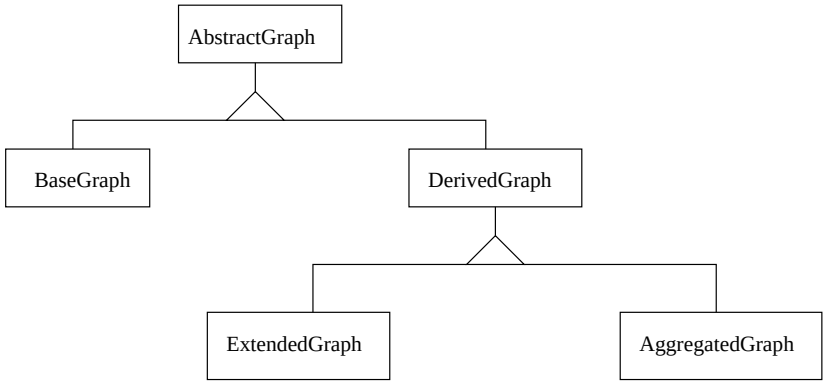


Figure 9.1: Different types of Graphs

An aggregated graph is defined using an edge-map. The same edge-map can be used to subgraph the underlying graph before applying a nodeflex. Thus a particular value in the edge-map will indicate an exclusion value, while all other values in the edge-map will belong to the ordered value set defining a nodeflex.

Since there are three types of graphs defined in the data model, an abstract class is used to represent the abstraction of generic graph properties (`AbstractGraph`). `BaseGraph` inherits from `AbstractGraph` and represents an edge set in a text file and uses `name` to access this. Any other graph (subgraph or aggregate graph) uses an edge-map to define itself. Class `DerivedGraph` inherits from `AbstractGraph`

and holds a reference to its `BaseGraph`, its `name`, and the edge-map defining it. `AggregateGraph` inherits from `DerivedGraph` and holds the following: `pseudo-edges`, `aggregate-tree`, and optionally another edge-map `aggregate-map` (Chapter 7).

Conceptually an IS-A relationship exists between `AbstractGraph` and `BaseGraph`; similarly between `AbstractGraph` and `DerivedGraph`, and between `DerivedGraph` and `AggregateGraph`. Thus inheritance used to implement them reflects the true relationship between these classes. It is feasible to implement `DerivedGraph` as an inherited class from `BaseGraph`, but the IS-A relationship does not exist between these two classes. `DerivedGraph` is based on a `BaseGraph`. The relationship therefore reflects the *Decorator* pattern, where the `BaseGraph` is used to create a more 'decorated' class [GHJV95].

Inheritance from `DerivedGraph` is used to define the `AggregateGraph` class. The `ExtendedGraph` based on the hybrid model of Chapter 8 is also implemented through inheritance from the `DerivedGraph` (Figure 9.1). `DerivedGraph` can be used for defining subgraphs of the `BaseGraph` using an edge-map. This can then be further 'decorated' with aggregation or extension. `ExtendedGraph` could have been implemented as a 'decorator' of `AggregateGraph` thus permitting aggregation of extended graphs.

9.2 Graphical User Interface Display

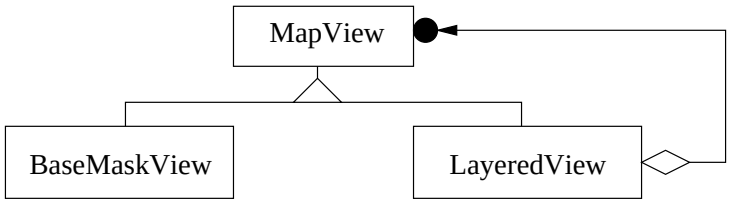


Figure 9.2: Different types of Views

Display and interaction with objects of classes mentioned above was implemented using Smalltalk VisualWork user interface framework, which is based on the pattern *Model/View/Controller*. The objects discussed in the last section are the main models that need to be displayed and interacted with. Of particular relevance is

the display and interaction with edge-maps so that subgraphs and aggregate graphs could be readily defined through interaction.

The Smalltalk user interface framework provides many reusable `View` and `Controller` classes. `Views` are dependent on the application model. Each model can have several views. A view holds a reference to its model and knows to display a specific view of it. All `View` classes have an associated `Controller` class that encapsulates how the view is to interact with the user. Typically `View` classes can be associated with different `Controller` classes to effect different types of interaction. The VisualWorks user interface framework consists of a `SimpleView` class from which `PushButtonView`, `RadioButtonView`, etc are inherited. Scrollable views such as `TextView` and `ListView` can be inherited from `AutoScrollingView`.

MapViewController

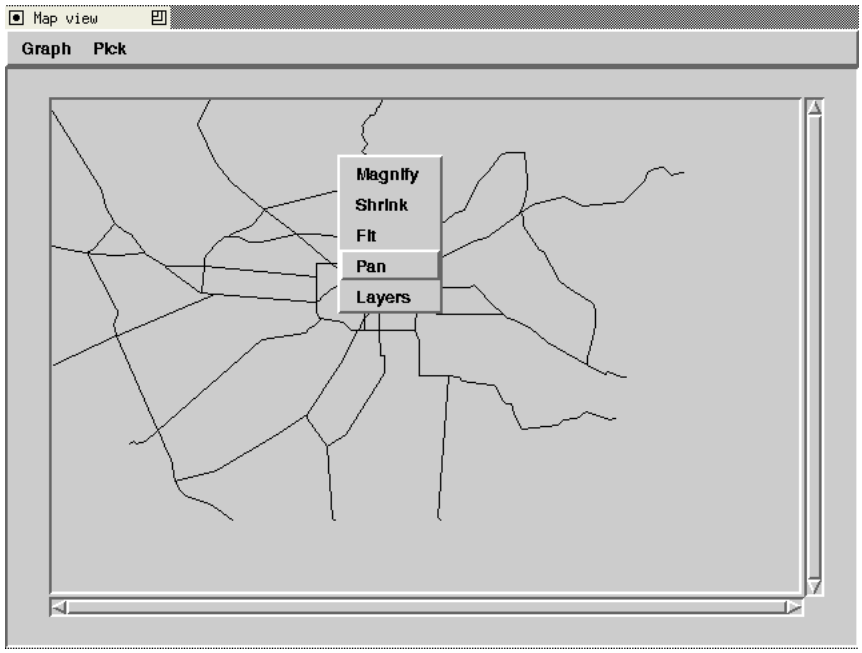


Figure 9.3: Operations Menu

In Smalltalk each view class has a controller that receives the user's input. The controller can be installed into the `View` class at any time. The `MapViewController` class is implemented to work with all the implemented view classes that are used to browse 2-D maps. It adds the four map browsing functions: magnify, shrink, pan

and zoom to its function button (Figure 9.3). In addition, the controller permits the selection of an item that is displayed, which is then highlighted. The highlighting process needs to perform the following steps: pick an object when the mouse button is pressed, highlight that object while the button is depressed, and remove the highlighting of the object when the button is released. It is the controller object that detects the mouse actions, while the actual selection of the picked object, its highlighting, and removing the highlighting must be done by the view.

MapView

Figure 9.2 shows the inheritance hierarchy of the scrollable views that were implemented. They are all intended for viewing 2-D maps. The `MapView` class, inherited from the `AutoscrollingView`, has the ability to display its model on the graphics medium provided. It can magnify, shrink, pan and zoom the display relative to the current view and a selected position. The instance variables of `MapView` class for achieving these functions are `mapScale`, and `mapOrigin`. All functions are triggered by using the three mouse buttons, resulting in a very easy browsing interface.

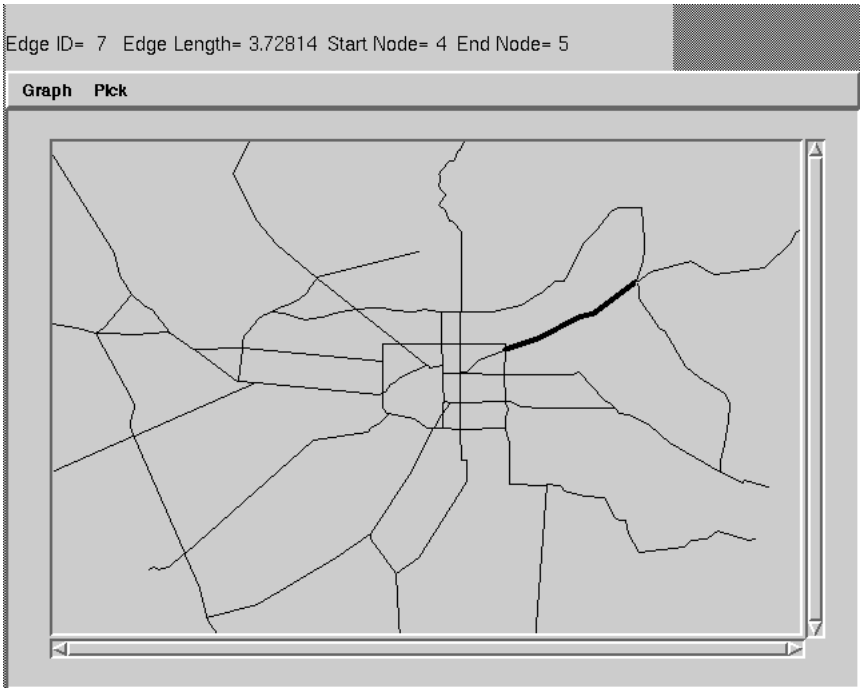


Figure 9.4: Highlighting and Browsing

To facilitate the highlighting of chosen objects the following instance variables are added: `hiliteObject, isHilited, hiliteRandom`, and `browseWindow`. The controller sends a 'highlight' message to the view class which stores the current mouse position. The `View`'s method must then set the `hiliteObject` and `isHilited` status appropriately.

The `View` class queries its model to determine the object to be highlighted, and if an object is found in that position, `hiliteObject` is set to this object and is highlighted. This also sets the `isHilited` flag. The selection of valid objects is derived by querying the model for objects that overlap a small rectangle around the mouse position.

There may be more than one object that is a valid candidate for browsing. By changing the size of the rectangle that is used to query the model, relative to the current display scale, a `View` is able to reduce the number of browsable objects which can be seen from a given mouse position. `hiliteRandom` is then used to randomly select one object from the list of valid candidates. This permits browsing all objects that may overlap. Since the number of items in this list is usually small, by repeated selection all objects may be browsed. This worked well.

The `browseWindow` is an additional text window displayed next to the main window holding the `MapView`. The `browseWindow` is used to display any textual data describing the selected object. This window is displayed and removed together with the other highlight operations used to highlight the object in the `MapView`. Figure 9.4 shows the highlighted object in a `MapView`. The highlighted object is displayed as a thicker line and the `browseWindow` above the main window shows the textual description of the highlighted object.

This highlighting operation is applicable to any 2-D view of a model and its associated controller class.

EdgeMapView

`EdgeMapView` is used to display and edit edge-maps. The highlight procedure described above is adapted to achieve this. An instance variable, `newValue`, of this

class is first set by the user to a desired value. As an edge is selected by the mouse, the edge-map value is set to the `newValue` if it is different from it. If it is the same as `newValue` then it is reset to its original value. Use of colours to display the different edge-map values, as in Figure 7.2, together with this operation, provides an easy interface for editing an edge-map to create the desired graph nodeflex.

LayeredView

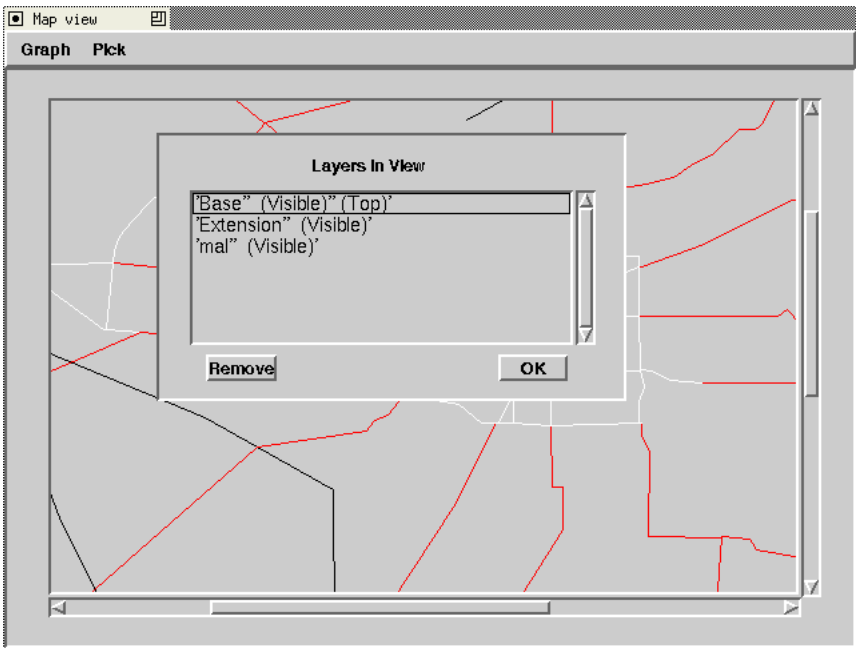


Figure 9.5: Dialogue to Manage Layers

The `LayeredView` class contains a collection of `MapViews` that are not of the type `LayeredView`. This view does not represent its own model, rather it transfers the user's input to one of the views, the top view, in its collection . Its inheritance from the `MapView` class permits uniform behaviour with respect to its mouse button operation.

In order to equip the `MapViews` to be members of a `LayeredView`, the following additional instance variables are added to the `MapView` classes: `visible`, and `top`. These two variables are used only when the `MapView` instance is added to a `LayeredView`. Figure 9.5 shows a dialogue that can be popped using the operations menu (Figure 9.3). Using this dialogue, layers can be made visible or hidden; layers can be

permanently removed from this view; and any one of the layers in the view can become the top layer that receives the user's input.

View-Highlight and Layered-View Patterns in Smalltalk

The discussion above on implementing the `MapView`s can be applied to any 2-D view of a model and its associated controllers. The `LayeredView` can also be used for any 2-D display. This can be formulated as two new software design patterns in Smalltalk (Section 2.3).

9.2.1 View-Highlight Pattern

Intent

Build an interface between subclasses of Smalltalk's `View` and `Controller` classes to provide browsing of objects displayed in the view.

Motivation

2-D views may be implemented for many independent models. Browsing is a common requirement among all such views. Direct access manipulations are the norm in graphical user interfaces. A displayed view may consists of several objects of interest to the user. Operations can be defined with respect to one of these objects. The user's ability to pick this object by selecting it using a pointer from the display is a common approach.

Applicability

Any 2-D view implemented as a Smalltalk view class and its associated Controller class can be modified to provide highlighting, browsing and selection.

Structure

Participants

- `View`: Must respond to highlight and remove highlight messages. Must also take action to magnify, shrink, pan and zoom messages. Must build a transient

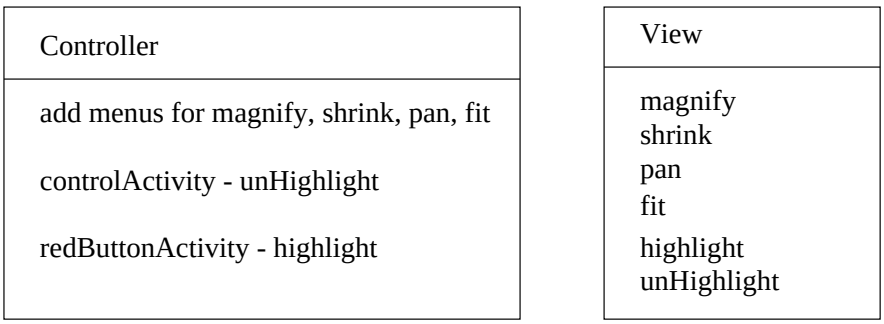


Figure 9.6: View-Highlight Pattern

browse window and display text describing the selected object

- `Controller`: Send highlight and remove highlight messages to its view according to users actions. These can be implemented inside two `Controller` methods: `ControlActivity` and `redButtonActivity`. Must also implement magnify, shrink, pan and zoom in its operations menu

- `LayeredView`: In addition to the actions in the operations menu of Figure 9.6, the controller of this view must also include another action for layer maintenance. This must be able to: remove layers, set the top layer, and the visibility of each layer in the collection

- `Views Container`: Views in Smalltalk are built using the *Composite* pattern. A given view that is displaying a 2-D map may not be the top window in the windowing system of the host. In order to place the transient browse window at an appropriate place the view's top container must be queried about its current position.

Consequences

Implementing the highlighting as described interferes with the ability to pick a point (coordinates) in the current display. To resolve this, either the controller's mode must alternate between the two modes or two separate controllers must be used to implement the two modes on the same view. This can be easily done using the views `setController:` method.

9.2.2 Layered-View Pattern

Intent

Provide a layered 2-D view consisting of several views in a graphical user interface.

Motivation

Layered views enable the user to relate information spatially and easily remove some layers when not needed.

Applicability

Any Smalltalk view class that can display a 2-D view can be used for this purpose. In GIS and design applications information needs to be presented to the user as 2-D maps placed one on top of the other.

Structure

The layered view is implemented as a set of `MapViews` (Figure 9.2).

Participants

- `MapView`: The set of views that take part in a `Layered-View` must all be of this type.

- `MapViewController`: The controller must implement a menu entry to activate maintenance of the views in the layer.

- `LayeredView`: This must implement the dialogue to maintain the view set. These include: setting the top view, setting the visibility of each view and removing a view. It must also transfer most messages to the top view. Some of the messages directed at the view must, however, must be handled by this class without transferring it to the top view. These are: calculating the bounds of the model, which must evaluate the bounding rectangle of all the visible views, and setting the map origins, which must be set for all the views.

9.3 Spatial Indexing

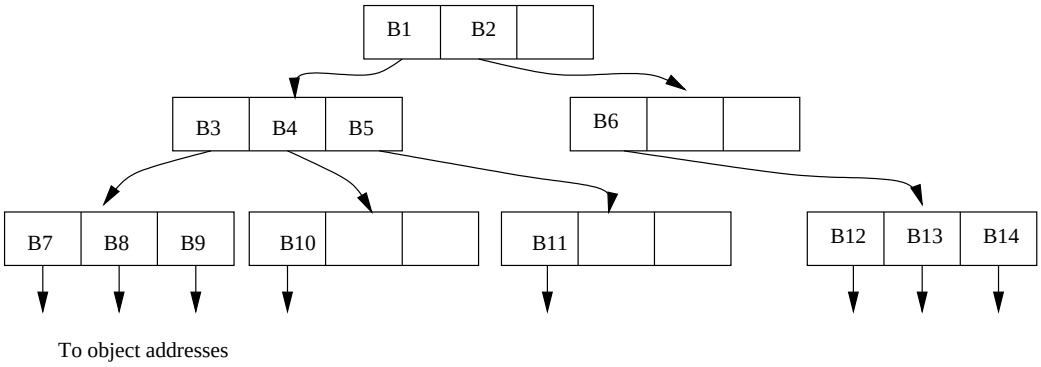


Figure 9.7: A Btree

The display and interaction of graphs needs a spatial indexing scheme to facilitate interactive browsing. Samet [Sam89] suggests a PM quad-tree for indexing a polygonal area bounded by a planar graph. This assumes that the edges of the planar graph are straight lines; thus it will treat each vertex of a polyline as a node of the planar graph. PM quad-trees also rely on the assumption that no two lines intersect except at their end points. In other words it assumes a constrained cell topology of Section 2.1.2. When used to index a planar graph display, PM quad-tree does not provide any assistance in calculating the position of the pointer with respect to the selected line.

The implementation used in this work uses a new scheme based on Rtree [Gut84], which provides these requirements. It does not treat each vertex of a polyline as a node. It does not assume that the polylines do not intersect. It also provides assistance in calculating the position of the pointer with respect to the polyline.

Btree

Rtree is an extension of the Btree indexing scheme [Sed88], which progressively subdivides the full range of indices into smaller ranges, thus forming a tree structure. The address of the source location corresponding to an object is stored at the leaf nodes of the tree (Figure 9.7). Btree indexing is well suited for maintaining the indexing under updates. An implementation of the Btree in SmallTalk consists of a

single class of the form:

```
Class BtreeNode
instance variables: order, isLeaf, keyList, nodeList.
```

Each node in the Btree is an instance of the above class. The `nodeList`s of the nodes higher up the tree point to the `BtreeNode` instances lower down. The `nodeList` at the leaf nodes (`isLeaf`) points to an object in the target collection. The `keyList` holds a list of key values. Two consecutive values in the list indicate the range of key values associated with the corresponding node. The `order` is the maximum size of the `nodeList` and `keyList`. When this size is exceeded, the `BtreeNode` is split into two `BtreeNode`s. When the root node is split, it results in the creation of a new root `BtreeNode`.

Rtree

The Rtree scheme uses the bounding rectangle as the key for indexing. The entire bounding rectangle of the model is progressively subdivided into smaller rectangles, thus forming a tree. However, unlike a 1-D indexing of Btree, where the index ranges do not overlap, in the 2-D indexing of Rtree, the ranges may overlap to some degree. Essentially, the Rtree based scheme used in the implementation consists of viewing each polyline as a set of fixed length segments. Each segment is identified by its position along the polyline as an offset from the start. The length of these segments may be fixed for the entire model or fixed for each polyline. Fixing the length of the segment reduces the amount of data that needs to be maintained in the Rtree. Figure 9.8 shows the polylines and the bounding rectangles of the fixed length segments of the polylines.

Rtree for Polylines

The class `LSoEquiDivider` implements the division of a polyline into equal segments using a step value. The class `LsoStrip` refers to an instance of `LsoEquiDivider` and an index value, thus determining a segment of the polyline. These `LsoStrip`s are

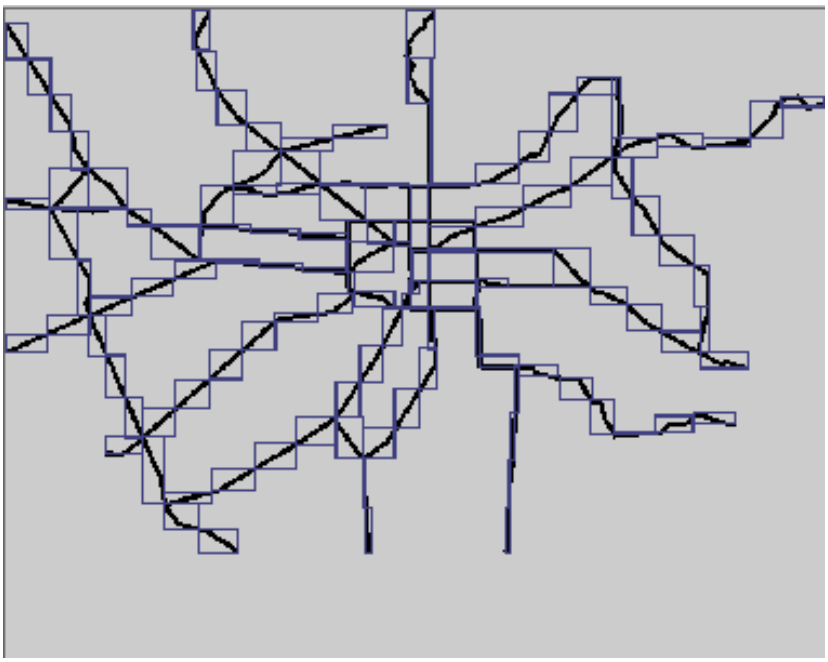


Figure 9.8: Leaf Nodes of the Rtree

used as elements of the Rtree. The strip length may be varied to obtain optimum performance. It was found that even when the strip length is set to half the size of the longest polyline the performance was acceptable. The leaf nodes of the Rtree consist of the bounding rectangles of the segments.

Selecting Polyline Offset

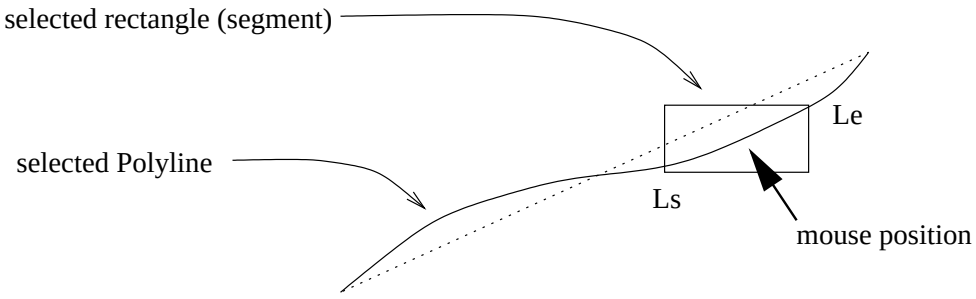


Figure 9.9: Selecting an Offset

When a pointer selects a leaf node the corresponding polyline and segment are identified. The position of the pointer with respect to the polyline may be approximately calculated using the following formula (Figure 9.9).

A given value of $step_c$ will subdivide a polyline into $\frac{polylinelength}{step_c}$ segments. The remainder will form the last segment in the divider.

The start and end offsets of a selected segment from the start of the polyline are derived based on the selected segment's position in the `LsoEquiDivider`.

Let this position be pos_i.

Let the step value of the divider be $step_c$.

The start offset of the selected segment with respect to the polyline is given by $L_s = (pos_i - 1) * step_c$ (Figure 9.9).

The end offset is obtained by adding $step_c$ to the start offset. However, when the segment is the last segment in the divider, the end offset is the same as the length of the polyline.

Let the start and end offsets of the selected segment with respect to the selected polyline be L_s and L_e respectively.

Let L_i be the offset of the pointer position.

The mouse position with respect to the selected polyline is given by the approximate expression

$$L_i = L_s + \frac{L_e - L_s}{2}$$

If a more accurate estimate is required then the coordinates of the start and end of the segment could be calculated first. Let this be $< x_s, y_s >$ and $< x_e, y_e >$ respectively.

Let the pointer coordinates be $< x_i, y_i >$

The pointer position is then given by

$$\frac{L_s - L_i}{L_s - L_e} = \frac{1}{2} * (\frac{x_s - x_i}{x_s - x_e} + \frac{y_s - y_i}{y_s - y_e})$$

Classes of the Rtree

The classes that implement this indexing scheme are:

- `Rtree`: Adds elements whose keys are rectangle bounds. Answers:

  `near: aPoint` and `overlap: aRectangle`.

- `LsoEquiDivider`: generates `LsoStrips` for building the Rtree. It is defined using a step value.

- `LsoStrip`: holds an `LSoEquiDivider` and an index into the divider to represent a particular strip of a polyline. It has an interface to behave like a key-value pair for insertion in an Rtree, where the key is the bounds of the strip and the value is the strip object itself. Since this, in turn, is a polyline another class `LsoWrapper` is defined which provides the polyline behaviour to `LsoStrip`. Thus `LsoWrapper` reflects the *Wrapper* software pattern [GHJV95].

9.4 Operations

The operations of the system have already been described in Chapters 7 and 8. The extraction algorithm of Section 7.1 builds the aggregate node tree of nodeflex. To test the extensible graph in space of Chapter 8, polyline intersection was implemented. The limited accuracy was handled by including the following extension to the SmallTalk implementation of floating point numbers.

A class variable `Precision` was added to the floating point number class and initialised to a desired value. Equality is then tested as follows:

```
n1 approxEQ: n2 within: precision

        ^((n1  - n2)  abs ) < (Precision)
```

9.5 Summary

In this chapter the implementation to demonstrate the proposals in this work was described. The main components of any GIS are: a data model for capturing geometry and topology; a graphical user interface for display with zoom, pan, browsing, etc.; a spatial indexing scheme for managing access to the data; and a query and analysis component with operations for handling problems in the application domain. Our implementation consists of the subsets of these components which were needed to demonstrate the proposal.

Useful contributions deriving from the implementation are: two software patterns for 2-D map display, and a spatial indexing scheme based on Rtree for 1-D objects.

Chapter 10

Conclusion

This work investigated the representation of a specialised geographic space namely, graph space found in the urban environment. The study of this specific space, as opposed to the more general 2-D space, has enabled the development of specialised techniques for its representation. The transportation graph and population graph are instances of the graph space to which the proposals in this work are applied.

The representation of the geographic space, be it 2-D space, 3-D space, or a specialised space such as graph space, needs to deal with some recurring requirements, such as capturing topology, multiple representation and spatial indexing for fast access. These have been addressed in this work. This work uses *Patterns* to discuss, highlight and document these and other recurring problems and their solutions in the graph space representation domain.

10.1 Multiple Representation

The main contribution of this work is in the area of multiple representation of graphs. Multiple representation within GIS, also referred to as generalisation, covers wide ranging applications and solutions. Graph based generalisation can be effective in several areas.

Mainguenaud [Mai95] suggests a multiple representation scheme that is built into

the model for networks in GIS. Stell and Worboys [SW99] consider graph based generalisation that include selection and aggregation of the graph elements for a simplification operator of graphs. This work develops a multiple representation of graphs based only on aggregation that can generated "on the fly" at any time.

Graph aggregation has also been used in the Artificial Intelligence field. Here the neighbourhood graph is used as the unifying data structure for building software to implement certain classes of automated reasoning [YZ96]. Thus it takes a continuous field and a procedure to extract a neighbourhood graph from it. The neighbourhood graph thus built can be further classified into another neighbourhood graph and so on.

Graph based generalisation could be dynamically built using any rule based procedure and the properties of the elements of the neighbourhood graph. This is the approach adopted in [YZ96] and [SW99]. A special case of graph based generalisation is graph aggregation. This work has suggested that whatever method is used to create the graph aggregation, it is equivalent to assigning true/false values to the edges of the neighbourhood graph. [YZ96] takes this aggregated graph and reclassifies it as a new graph for further aggregation, and so on. This work has suggested that this process of building multiple levels of aggregation is equivalent to assigning an ordered set of values to edges. This is then exploited in instances where the values associated with edges implicitly represent an aggregation of the graph. Thus, [YZ96] concentrates on building a generic, rule based approach for building graph aggregation to extract additional knowledge, while the one proposed here simplifies the graph aggregation problem by using ordered edge values.

The ordered edge values are represented as an edge-map. When the edges have a 2-D representation as in the case of graphs in space, the edge-map can be edited using a graphical user interface. The edge-map display consists of the edges being displayed using different colours for each edge-map value. This provides effective visualisation of the graph aggregation for editing purposes.

The proposed scheme permits the definition of a tree that is imposed over a graph through multiple layers of aggregation. This tree is then exploited to eliminate irrelevant edges during shortest path finding, as demonstrated in Chapter 7. The

upper bound of the computational effort needed to execute the best known shortest path finding algorithms is $O(n^2)$, where n is the number of edges of the graph. By reducing the number of edges examined using the aggregate tree over a graph, a substantial reduction in computational effort can be achieved.

Edge value based aggregation also permits the dynamic creation of user defined views of the aggregation, a feature not available in [YZ96]. Application of this in spatial interaction modelling is suggested. This feature is similar to the *develop* and *undevelop* operations suggested in [Mai95] to define a user specified view of a multi-layered graph that is built in at model construction time. However, our proposal, by enabling the definition of graph aggregation through an easy-to-use interface and then proceeding to develop user defined views based on this, is simpler than [Mai95]. This simplicity is achieved by foregoing the ability to add new graph elements at the higher levels of aggregation. This can be justified because this has no direct application in representing graphs in space. The assertion can therefore be made that the proposals and the resulting simplicity have applications where the approach of [Mai95] is not effective.

Two applications of the proposal will be worth investigating further. Firstly, its application in spatial interaction modelling could be more fully tested. [FDC95] states three types of errors that contribute to the Modifiable Aerial Unit Problem when spatial interaction modelling is based on zones. The graph based aggregation suggested here could eliminate some of these errors. Secondly, the application of edge value based multi level aggregation in knowledge extraction could also be studied.

10.2 Other Contributions

Three other contributions of this work are: a hybrid model for representing extensible graph space, a spatial indexing scheme for polylines, and some Smalltalk user interface software patterns for map browsing.

10.2.1 Hybrid Model of Graph Space

Capture of the topology of spatial objects has been investigated extensively by researchers for representing 2-D space. Consideration of this requirement for the specialised graph space leads to some simplifications. The thesis demonstrates this by developing a simplified model for capturing the topology of an extensible graph space.

A cell topology based on 0-cell, 1-cell and 2-cell is used for representing 2-D space. By placing increasingly tight constraints on the relationships among the three types of cells, different cell topologies can be derived. The most restrictive of these constraints is equivalent to the simplicial complex, where all 1-cells are straight lines between two 0-cells and all 2-cells are triangles formed by three 1-cells in the simplicial complex. For the representation of graphs in space, the less restrictive cell topology of overlaying all the polylines is sufficient, and is commonly used in commercial systems.

Maintaining the constraints of the spatial model under the addition of new spatial objects is non trivial and is often ignored in commercial systems where this requirement is not seen to be of great importance. The hybrid model suggested here for representing extensible graph space is relatively simple to implement and has applications in urban systems. This model exploits the ease of computing points along a polyline based on an offset from the start of the polyline. Thus the hybrid model uses the standard overlaid model of graphs in space and deals with the addition of new polylines by using another set of data structures that rely on computing points along the polylines.

10.3 Rtree Indexing of Graph Space

Many of the spatial indexing proposals in [Sam89] for curvilinear data assume a constrained cell topology that assumes that the polylines have been overlaid. They are also designed to index the 0-cell, 1-cell and 2-cells in the model. This work suggests an Rtree based indexing scheme for curvilinear data. Rtree based indexing

schemes use the bounding rectangle of a spatial object as the key on which the indexing is performed. A method of adapting this to graphs in space is suggested in this work. The spatial objects to be indexed are the polylines. If the bounding rectangle of these polylines are used for indexing, then the degree of overlap among the bounding rectangles will be too high to be effective. An indexing scheme based on the bounding rectangles of equal length segments of the polylines is suggested and used in this work. This scheme has advantages compared to those suggested in [Sam89]. It will index any collection of polylines without requiring a constrained cell topology to be maintained. The indexing based on equal length segments enables efficient selection of a point along the selected polyline without the need to compute it by projecting the selected point onto it.

10.4 Map Browsing User Interface Patterns

Multiple representation, capturing topology and spatial indexing are three recurring patterns in the GIS domain. These are still too complex to be documented as a pattern with a simple problem statement and reusable solutions. Two software patterns, however, have been documented in Chapter 9 for implementing 2-D map browsing using the Smalltalk user interface classes. The *View-Highlight* pattern permits the highlighting of a selected item in the map and the display of textual information of the selected item. The *Layered-View* pattern captures the problem of and a solution to viewing multiple superimposed 2-D views.

10.5 Summary

By studying the graph space in urban environments, this work has derived a number of useful contributions. These are:

1. The development of a specific graph aggregation scheme based on the ordered values associated with graph edges. One application of this for finding shortest paths has been demonstrated. Another application in spatial interaction

modelling has been suggested and will be worth studying. Edge value based aggregation could also have uses in other applications that use graph based aggregation and this should be explored further.

2. The hybrid model of extensible graphs in space can be used to model transportation and population graphs, where the ability to explore the effects of adding new edges will be useful.

3. The Rtree based indexing scheme for indexing unconstrained 1-cells is useful for interactive browsing and selecting locations on the 1-cells.

4. The 2-D map browsing patterns, *View-Highlight*, and *Layered-View* can be used to develop 2-D views in the *Model/View/Controller* triad of the Smalltalk user interface framework.

Appendix A

Glossary

Aggregate map: An edge-map of a graph that records the aggregate node identifier to which the edge belongs, the root aggregate node is denoted by zero. *69*

Aggregate node tree: The tree of aggregate node identifiers of a particular graph aggregation.*62*

Boundary node: The nodes of a subgraph that has some edges that are not part of this subgraph.*33*

Connected graph: A graph where each node can be reached from any other node when the edges are treated as undirected.*33*

Disjoint graphs: Two subgraphs that share neither their nodes nor their edges.*46*

Edge-map: A mapping of values associated with the set of edges of a graph.*58*

Linear spatial object: is a specific class of polyline. It is an ordered series of non-intersecting and touching line segments. It is considered to have a direction that is arbitrarily assigned to it to facilitate the definition of points along it.*90*

Nodeflex: A particular interpretation of graph aggregation based on an ordered set of values in an edge-map of the graph.*50*

Order enforcing list: A list data structure that enforces some order on the retrieval of elements from it. Examples are stack, queue and heap.*41*

Pseudo-edges: Assumed undirected graph edges of an aggregated graph view whose end nodes are the aggregated node and one of its boundary nodes.*60*

Single & Multiple get behaviours: Permits items from a set to be retrieved only once or multiple times respectively.*42*

Subgraph: A subset of the edges of an underlying graph.*32*

Appendix B

2-D Map Browsing Classes

AutoScrollingView subclass: MapView

```
instance variables:
  mapScale mapOrigin hiliteObject isHilited
  hiliteRandom browseWindow visible top

MapView comment:

A view class in the MVC triad to handle display of maps. The model is
displayed translated to mapOrigin and scaled to mapScale  by this view.
The yellowButton can be used to zoom in and out of the map.

 <mapScale.> The scale at which the model is displayed.
 <mapOrigin> The translation applied to the model before display.

 The following instance variables are used for hi-lighting a
 selected object

 <hiliteObject isHilited hiliteRandom browseWindow>
 < visible top > These two are used only to maintain the views in
                 LayeredView list
```

```
MapView methods:

initialize
   super initialize.
   mapScale := 30.
   visible := true.
   top := true.
   isHilited := false.
   hiliteRandom := Random new.
   self controller pickReceiver: self pickMethod: #hilite pickUndo:
             #unHilite.
"Menu actions are performed by the view"
    self controller performer: self.
  ^self.! !

model: aModel
   super model: aModel.
   mapOrigin := self modelBounds origin.
    ^self.!

defaultControllerClass
  ^MapViewController! !

mapScale: aFloat
"Change the scale of the displayed map."
  | oldBounds |
  oldBounds := self preferredBounds.
  mapScale :=  aFloat.
  self changedPreferredBounds: oldBounds.
   ^self.!
```

```
preferredBounds
"Return the preferred bounds based on current scale and origin"
     ^(self modelBounds translatedBy: mapOrigin negated)
               scaledBy: mapScale.

fit
"Fit the preferred bounds into the current bound of the view."
   self mapScale:
      (( self bounds height ) /  ( self modelBounds height )   min:
                ( self  bounds width ) / ( self modelBounds width )).
    self positionTo: 0@0.
    self invalidate.!

magnify
"Magnify the view at the point where the menu
button,<yellowButton>  was pressed"
  self magnify: 2 at: self controller sensor mousePoint.!

pan
"Pan to the point where the yellowButton was pressed"
   self  centreOn: self controller sensor mousePoint.
   self invalidate.!

shrink
"Shrink the scale of the view centred on where the yellowButton
was pressed"
  self magnify: 0.5 at: self controller sensor mousePoint.

centreOn: aPoint
"Position the view centred around aPoint"
    self positionTo: (aPoint- (self bounds extent * 0.5)) negated.
```

```
magnify: amount at: aPoint
"Magnify the view by amount centred around aPoint"
     self mapScale: mapScale*amount.
     self centreOn: aPoint * amount.
     self invalidate.
     ^self.!

mapOrigin: aPoint
    mapOrigin := aPoint.!

modelBounds
  | bnds |
  bnds := self model bounds.
  (bnds =nil)
  ifTrue:  [^Rectangle origin: 0@0 extent: 1@1]
  ifFalse: [^bnds].

displayOn: aGraphicsContext
  self model displayOn: aGraphicsContext
           translatedBy: mapOrigin negated scaledBy: mapScale.
  isHilited ifTrue: [ self displayHilite: hiliteObject On: aGraphicsContext].

displayHilite: anObject On: aGraphicsContext
   | lw paint |
   lw := aGraphicsContext lineWidth.
   paint := aGraphicsContext paint.
   aGraphicsContext paint: (SymbolicPaint key: #backgroundColor).
    anObject displayOn: aGraphicsContext
            translatedBy: mapOrigin negated scaledBy: mapScale.
     aGraphicsContext lineWidth: 4.
     aGraphicsContext paint: (SymbolicPaint key:  #foregroundColor).
     anObject displayOn: aGraphicsContext
            translatedBy: mapOrigin negated scaledBy: mapScale.
```

```
     aGraphicsContext paint: paint; lineWidth: lw.!

hilite
  self hilite: self controller sensor cursorPoint container: self.

hilite: aPoint container: containerView
  | hiliteList |
  isHilited ifFalse: [
    hiliteList := self model overlap: (Rectangle origin:
                  (self viewPoint: aPoint) extent: self hiliteExtent).
    (hiliteList size) > 0
     ifTrue: [
       isHilited := true.
       hiliteObject := hiliteList at:
                    ((hiliteRandom next * hiliteList size) truncated + 1).
        self hiliteOnContainer: containerView]].

hiliteOnContainer: containerView
    browseWindow :=
          TransientWindow aPopUpIn:  (containerView browserBox).
    browseWindow map.
    (ComposedText withText:
             (Text fromString:  (self hiliteText: hiliteObject )))
                         displayOn: (browseWindow graphicsContext).
    self invalidateHilite: containerView.

invalidateHilite: containerView
   | rect |
   isHilited
   ifTrue: [
      rect := hiliteObject bounds.
      containerView invalidateRectangle: (self mapRect: rect)].
```

```
unHilite
        self unHilite: self.

unHilite: containerView
   isHilited
   ifTrue: [
     browseWindow recycleTransient.
     self invalidateHilite: containerView.
     hiliteObject := nil.
     isHilited := false ].


browserBox
"Calculate the screen position for a box to be displayed when <select>
button is pressed in the map view"
  ^(Rectangle origin: self topComponent displayBox origin extent:
          300@50) translatedBy: 0@-50.

hiliteExtent
        ^1.0 / mapScale.
```

ControllerWithMenu subclass: MapViewController

```
instance variables:
   pickMethod pickReceiver pickUndo mouseClicked

MapViewController comment:

The controller in the MVC triad.  Browsing is done using the <select>
button. It adds map zoom, pan and fit actions to the <operation>
button.

 <menuPoint> CursorPoint where yellowButton was pressed.
```

```
  The following three instance variables handle the highlighting
  behaviour of the view-controller. By installing different objects
  into them this behaviour can be altered.
   < pickMethod pickReceiver pickUndo>

MapViewController methods:

initializeMenu
   | mb |
"Build the menu."
   mb := MenuBuilder new.
   mb add: 'Magnify' -> #magnify.
   mb add: 'Shrink' -> #shrink.
   mb add: 'Fit'   -> #fit.
   mb add: 'Pan' ->#pan.
   mb add: 'Layers' -> #layers.
"Install the menu."
   self menuHolder: mb menu asValue.! !

pickReceiver: anObject pickMethod: aMethod pickUndo: anUndo
  pickMethod := aMethod.
  pickReceiver := anObject.
  pickUndo := anUndo.

controlActivity
"Check for mouse button and keyboard activity."
"If the <select> mouse button is being pressed"
  self sensor redButtonPressed
  ifTrue: [^self redButtonActivity].
"If the <operate> button is being pressed, display the menu."
  self sensor yellowButtonPressed
  ifTrue: [ ^self yellowButtonActivity].
```

```
  mouseClicked := false.
  pickReceiver perform: pickUndo.

redButtonActivity
  mouseClicked
  ifFalse: [
    pickReceiver perform: pickMethod. mouseClicked := true].
```

MapView subclass: LayeredView

```
instance variables: mainView colln

LayeredView Comments:

LayeredView permits the display of several views in the same area. The
controller actions are directed to the top view. The top view and
visibility of each view can be modified through the operations
menu. Layers can be added and deleted from the view.

LayeredView methods:

addView: aView
  (colln isEmpty)
  ifTrue: [
    aView top: true. aView visible: true. mainView := aView]
  ifFalse: [ aView top: false ].
  colln add: aView.
  aView mapScale: (self mapScale).
  self adjustBounds.
```

```
mapScale: aFloat
  super mapScale: aFloat.
  colln do: [ :view | view mapScale: aFloat].!

removeAll
  mainView := nil.
  [colln isEmpty] whileFalse: [colln removeFirst].!

selectionLayers
  | ls |
  ls := List new.
  colln do:  [:view | ls add: view].
  ^ls.! !

displayOn: aGraphicsContext
  (colln isEmpty)
  ifFalse: [
    colln do: [ :view |
      (view visible)
       ifTrue: [ | gc |
         gc := aGraphicsContext. view displayOn: gc ]]].

hilite
  (colln isEmpty)
   ifFalse: [
      mainView hilite: self controller sensor cursorPoint
                                              container: self].

unHilite
  (colln isEmpty) ifFalse: [mainView unHilite: self].
```

```
initialize
  super initialize.
  colln := List new.
  mainView := self.
  self controller pickReceiver:
      self pickMethod: #hilite pickUndo: #unHilite.
  ^self.

layers
  | ls dlg |
  (self isKindOf:  LayeredView)
  ifTrue: [
    (colln isEmpty)
     ifFalse: [
       ls :=  SelectionInList with: self selectionLayers.
       dlg := (LayerDialog  new) layers: ls.
       ls selectionIndexHolder onChangeSend:
             #changedSelection to: dlg.
       dlg openInterface: #listSpec for: self.
       dlg changed ifFalse: [ ^self].
       colln := ls list.
       colln do: [:view | (view top)
       ifTrue: [ mainView :=  view. ]].
       self invalidate]].

adjustBounds
  self mapOrigin: self modelBounds origin.
  self invalidate.

mapOrigin: aPoint
  mapOrigin := aPoint.
  colln do: [ :view | view mapOrigin: aPoint].
```

```
modelBounds
| bnds |
  (colln isEmpty)
  ifFalse: [
    bnds := mainView model bounds.
    colln do: [:view | bnds := bnds merge: (view model bounds)].
    ^bnds]
  ifTrue: [^Rectangle origin: 0@0  extent: 1@1].
```

SimpleDialog subclass: LayerDialog

```
instance variables: changed layers
```

```
LayerDialog comments:
```

```
This dialog is used to maintain the views in a LayeredView.
```

```
LayerDialog methods:
```

```
changedSelection
  ((self layers selection) == nil) ifTrue: [^self].
  (self layers selection top)
   ifFalse: [
     self layers selection visible:
         (self layers selection visible not)].
   changed := true.
```

```
done
  self layers list first invalidate.
  self close.

removeSelection
  | view |
  view := self layers selection.
  (view = nil) ifTrue: [^self].
  (view top) ifTrue: [^self].
  self layers list remove: view.
  self layers list: (self layers list).
  changed := true.

topSelection
  self layers list do: [:view| view top: false].
  self layers selection top: true.
  self layers selection visible: true.
  self layers list: (self layers list).
  changed := true.
```

Bibliography

[AA96] V Ahl and T F H Allen. *Hierarchy Theory, A Vision, Vocabulary and Epistemology.* Columbia University Press, 1996.

[AG97] N A Adam and A Gangopadhyay. *Database Issues in Geographic Information Systems.* Kluwer Academic Publishers, 1997.

[AIS77] C Alexander, S Ishikawa, and M Silverstein. *A Pattern Language.* Oxford University Press, 1977.

[Alb99] J Albrecht. Defining geographic knowledge. 1999. http://www.geog.ucsb.edu/~jochen/public_html/varenius/ja-pos.html.

[BEMW90] R A Becker, G Eick, E O Miller, and A R Wilks. Network visualization. In *Proc. 4th Spatial Data Handling, Zurich*, 1990.

[BK91] B P Bruegger and W Kuhn. Multiple topological representations. Technical Report 91-17, National Centre for Geographic Information and Analysis, 1991.

[Bos90] J Bosak. *Decomposition of Graphs.* Kluwer Academic Publishers, 1990.

[Bro87] F P Brooks. No silver bullet. Essence and accidents of software engineering. *Computer*, pages 10–19, April 1987.

[BX94] M Batty and Y Xie. Modelling inside GIS: Part 1, model structures, exploratory data analysis and aggregation. *Int. J. Geographical Information Systems*, 8(3):291–307, 1994.

[Car79] B Carre. *Graphs and Networks.* Oxford University Press, 1979.

[Car98] A Car. Hierarchical spatial reasoning: a geocomputation method. In *Proceedings of the Third International Conference on GeoComputation*, 1998. http://www.strabo.geo.port.ac.uk/geocomp/geo98/18/gc_18.htm.

[CG86] H Couclelis and S Gale. Space and spaces. *Geografiska Annaler B*, 86, 1986.

[Cou92] H Couclelis. Beyond raster vector debate in GIS. In A Frank, I Campari, and U Formentini, editors, , *LNCS 639*. Springer-Verlag, 1992.

[CS96] R L Church and P Sorensen. Integrating normative location models into GIS: problems and prospects with p–median model. In P Longley and M Batty, editors, *Spatial Analysis: Modelling in a GIS Environment*. GeoInformation International, 1996.

[DH97] V Delis and T Hadzilacos. On the assessment of generalisation consistency. In *Advances in Spatial Databases, LNCS 1262*, 1997.

[DV92] K J Dueker and R Vrana. Dynamic segmentation revisited: a mile-point linear data model. *URISA Journal*, 4(2):94–105, 1992.

[EFJ89] M J Egenhofer, A W Frank, and J P Jackson. A topological data model for spatial databases. In A Buchmann, O Gunther, T R Smith, and Y F Yang, editors, *Advances in Spatial Databases, LNCS 409*. Springer-Verlag, 1989.

[EKS97] M Ester, H P Kriegel, and J Sander. Spatial data mining: A database approach. In *Advances in Spatial Databases, LNCS 262*, 1997.

[Erw93] M Erwig. Graph algorithm = iteration + data structures? In *Graph Theoretic Approaches in Computer Science, LNCS 9999*. Springer-Verlag, 1993.

[Erw94] M Erwig. *Graphs in Spatial Databases*. PhD thesis, Fern Universitaet Hagen, Germany, 1994.

[ESR90] ESRI Inc. *Understanding GIS - The ACR-INFO Method*, 1990.

[ESR92] ESRI Inc. *Arc/Info Dynamic Segmentation Module, User Manual*, 1992.

[Eur95] European Committee for Standardisation. *Geographic Data Files, version 3.0*, 1995.

[FDC95] A S Fotheringham, P J Densham, and A Curtis. The zone definition problem in location–allocation modelling. *Geographical Analysis*, 27:60–77, 1995.

[Fre91] G N Frederickson. Planar graph decomposition and all pairs shortest path. *Journal of the Association of Computing Machinery*, 38:102–204, 1991.

[Gab96] R P Gabriel. *Patterns of Software: Tales from the Software Community.* Oxford University Press, 1996.

[GBdN97] S Gordillo, F Balaguer, and F das Neves. Generating the architecture of GIS applications with design patterns. In *Proc. of 5th ACM Workshop on Geographic Information Systems, Las Vegas*, 1997.

[GHJV95] E Gamma, R Helm, R Johnson, and J Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley, 1995.

[Goo92a] M Goodchild. Geographic information science. *Int. Journal Geographical Information Systems*, 6(1), 1992.

[Goo92b] M Goodchild. Geographical data modeling. *Computers and Geo-Sciences*, 18(4), 1992.

[Goo98] E G Goodaire. *Discrete Mathematics with Graph Theory.* Prentice Hall, 1998.

[GS93] R H Güting and M Schneider. Realms: A foundation for spatial data types in database systems. In *3rd Int. Symposium on Large Spatial Databases, Singapore*, 1993.

[Gut84] A Guttman. R-Trees: A dynamic index structure for spatial searching. *Proceedings of ACM SIGMOD Int. Conference on Management of Data*, pages 47–57, 1984.

[Güt89] R H Güting. GRAL: An extensible relational database system for geometric applications. In *15th Int. Conf. on Very Large Databases, Brighton*, 1989.

[Güt94a] R H Güting. GraphDB: Modelling and querying graphs in databases. In *20th Int. Conf. on Very Large Databases, Santiago*, 1994.

[Güt94b] R H Güting. An introduction to spatial databases systems. *Very Large Database Journal*, 3(4), 1994.

[HC91] L Haas and W Cody. Exploiting extensible DBMS in integrated Geographic Information Systems. In *Advances in Spatial Databases, LNCS 525*, 1991.

[Hux91] W E Huxhold. *Introduction to Urban Geographic Information Systems.* Oxford University Press, 1991.

[Joh77] D B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the Association of Computing Machinery*, 24:1–13, 1977.

[Joh88] R E Johnson. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[KPS97] G Koster, B Pagel, and H Six. GIS application development within GeoOOA. *Int. J. Geographical Information Systems*, 11(4):307–334, 1997.

[Kru92] C W Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[Las94] Laser-Scan Cambrige UK. *The Gothic Versioned Object-Oriented Database: an Introduction*, 1994.

[LT92] R Laurini and D Thompson, editors. *Fundamentals of Spatial Information Systems.* Academic Press, 1992.

[Mai95] M Mainguenaud. Modelling the network component of Geographical Information Systems. *Int. J. Geographical Information Systems*, 9(6):575–593, 1995.

[Mar90] R W Marx. The TIGER system: Automating the geographic structure of the United States census. In D J Peuquet and D F Marble, editors, *Introductory Readings in Geographic Information Systems*. Taylor & Francis, 1990.

[Meh84] K Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[MMS93] P Milne, S Milton, and J L Smith. Geographical object-oriented databases–a case study. *Int. J. Geographical Information Systems*, 7(1):39–55, 1993.

[Mol98] M Molenaar. *Introduction to the Theory of Spatial Object Modelling*. Taylor & Francis, 1998.

[MR94] J E McCormack and S A Roberts. Overview of the trip planning system prototype. Technical report, School of Computer Studies, University of Leeds, 1994.

[MS90] M Mannino and L D Shapiro. Extensions to query languages for graph traversal problems. *IEEE Transactions on knowledge and data engineering*, 2(3):353–363, 1990.

[MW95] J Muller and R Weibel. *GIS and Generalization: Methodology and Practice*. Taylor and Francis, 1995.

[Mye90] B Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[Nag95] M Naguleswaran. Achieving software reuse through domain frameworks. In *New Zealand Computer Science Research Students' Conference*, 1995.

[Nag97] M Naguleswaran. Issues in representing spatially embedded graphs. In *2nd Int. GeoComputation Conference, Otago, NZ*, 1997.

[Nag98] M Naguleswaran. Structures over graph for handling population data. In *Proceedings of 10th Colloquium of the Spatial Information Research Centre, Otago, NZ*, 1998.

[Nil82] N J Nilsson. *Principles of Artificial Intelligence*. Springer-verlag, 1982.

[NK96] M Naguleswaran and W Kreutzer. Towards a domain framework for modelling linear spatial objects in urban systems. In *Proceedings of Software Engineering Education & Practice, Otago, NZ*, 1996.

[Oos93] P J Oosterom. *Reactive data structures for Geographic Information Systems*. Oxford University Press, 1993.

[PD91] R Prieto-Diaz. Domain analysis for reusability. In R Prieto-Diaz and G Arango, editors, *Domain Analysis and Software System Modeling*. IEEE Computer Society Press, Los Alamitos, Calif., 1991.

[PD93] R Prieto-Diaz. Status report: Software reusability. *IEEE Software*, pages 61–66, May 1993.

[Peu88] F Peuquet. Representation of geographic space: Toward a conceptual synthesis. *Annals of the Association of American Geographers*, 78, 1988.

[Pre95] W Pree. *Design patterns for object-oriented software development*. Addison-Wesley, 1995.

[PS85] A F Preparata and M I Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[RHDM86] A Rosenthal, S Heiler, U Dayal, and F Maola. Traversal recursion: A practical approach to supporting recursive applications. In *Proceedings of the SIGMOD Conference*, pages 166–176, 1986.

[Sam89] H Samet. *Design and Analysis of Spatial Data Structures*. Adison-Wesley, 1989.

[Sed88] R Sedgewick. *Algorithms*. Adison-Wesley, 1988.

[SFG97] S Shekar, A Fetterer, and B Goyal. Materialization trade-offs in hierarchical shortest path algorithms. In *Advances in Spatial Databases, LNCS 1262*, 1997.

[Sim73] H Simon. Organization of complex systems. In H Pattee, editor, *Hierarchy Theory*. New York: Braziller, 1973.

[Sma92] SmallWorld Systems Inc. *SmallWorld GIS Customisation Reference*, 1992.

[SRH90] M Stonebraker, L Rowe, and M Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[SW99] J G Stell and M F Worboys. Generalizing graphs using amalgamation and selection. In *Advances in Spatial Databases, LNCS 1651*, 1999.

[Tim97] S Timpf. Cartographic objects in a multi-scale data structure. In M Cragila and H Couclelis, editors, *Geographic Information Research: Bridging the Atlantic*. Taylor & Francis, 1997.

[Tim98] S Timpf. *Hierarchical Structures in Map Series*. PhD thesis, Technical University of Vienna, 1998.

[vEdJ90] J R van Eck and T de Jong. Adapting data structures and algorithms for faster transport network computations. In *Proc. 4th Spatial Data Handling, Zurich*, 1990.

[WGP97] D Wright, M Goodchild, and J Proctor. Demystifying the persistent ambiguity of GIS as "tool" versus "science". *Annals of the Association of American Geographers*, 87(2), 1997.

[Wor92] M F Worboys. A generic model for planar geographical objects. *Int. J. Geographical Information Systems*, 6(5):353–372, 1992.

[Wor95] M F Worboys. *GIS: A Computing Perspective*. Taylor & Francis, 1995.

[YZ96] K Yip and F Zhao. Spatial aggregation: Theory and applications. *Journal of Artificial Intelligence Research*, 1(26), 1996.

[ZF97] L Zhoa and T Foster. Plots: pattern language of transport systems. In *EuroPlop97*, 1997.