

Development of a neighbourhood graph for trafficability analysis

Kennet Gustafsson, Joel Hägerstrand



FOI is an assignment-based authority under the Ministry of Defence. The core activities are research, method and technology development, as well as studies for the use of defence and security. The organization employs around 1350 people of whom around 950 are researchers. This makes FOI the largest research institute in Sweden. FOI provides its customers with leading expertise in a large number of fields such as security-policy studies and analyses in defence and security, assessment of different types of threats, systems for control and management of crises, protection against and management of hazardous substances, IT-security and the potential of new sensors.



FOI
Defence Research Agency
Command and Control Systems
P.O. Box 1165
SE-581 11 Linköping

Tel:
Fax:

www.foi.se

Development of a neighbourhood graph for trafficability analysis

Issuing organization FOI – Swedish Defence Research Agency Command and Control Systems P.O. Box 1165 SE-581 11 Linköping	Report number, ISRN FOI-R--1698--SE	Report type Methodology report
	Research area code 4. C4ISTAR	
	Month year June 2005	Project no. E7089
	Sub area code 42 Above water Surveillance, Target acquisition and Reconnaissance	
	Sub area code 2	
Author/s (editor/s) Kennet Gustafsson Joel Hägerstrand	Project manager Erland Jungert	
	Approved by Johan Mårtensson	
	Sponsoring agency Swedish Armed Forces	
	Scientifically and technically responsible Erland Jungert and Fredrik Lantz	
Report title Development of a neighbourhood graph for trafficability analysis		
Abstract (not more than 200 words) <p>Successful trafficability analysis requires usage of many data sources, e.g. elevation and land use data, in high resolution. However, this requires a large storage capacity and may lead to unduly long execution times when trying to find suitable paths. To allow for efficient representation and search in an area of interest (AOI), a searchable graph is created where the best paths between two arbitrary locations in the AOI can be found. Terrain features, identified from laser-radar data, and land use data are fused to create a compound map of the AOI.</p> <p>In order to create the graph, the map is partitioned into areas of homogeneous trafficability characteristics. These areas are represented by the nodes in the graph and the edges in the graph represent paths between neighbouring areas.</p> <p>Different methods to partition the map is described and analysed. Dijkstra's algorithm are combined with the A*-algorithm to find the best paths between two locations in the AOI.</p>		
Keywords trafficability, driveability, neighbourhood graph, terrain model, GIS		
Further bibliographic information	Language English	
ISSN 1650-1942	Pages 63 p.	
	Price acc. to pricelist	

Utgivare FOI - Totalförsvarets Forskningsinstitut - Ledningssystem Box 1165 581 11 Linköping	Rapportnummer, ISRN FOI-R--1698--SE	Klassificering Metodrapport
	Forskningsområde 4. Ledning, informationsteknik och sensorer	
	Månad, år Juni 2005	Projektnummer E7089
	Delområde 42 Spaningssensorer	
	Delområde 2	
Författare/redaktör Kennet Gustafsson Joel Hägerstrand	Projektledare Erland Jungert	
	Godkänd av Johan Mårtensson	
	Uppdragsgivare/kundbeteckning Försvarsmakten	
	Tekniskt och/eller vetenskapligt ansvarig Erland Jungert och Fredrik Lantz	
Rapportens titel (i översättning) Utveckling av en grannskapsgraf för framkomlighetsanalys		
Sammanfattning (högst 200 ord) <p>Framgångsrik framkomlighetsanalys kräver användning av många datakällor, t.ex. elevation och markanvändningsdata, i hög upplösning. Detta kräver, i sin tur, stor lagringskapacitet och kan leda till olämpligt långa exekveringstider för att finna bra färdvägar. För att tillåta effektiv representation och sökning i ett aktuellt område av intresse skapas därför en graf där de bästa vägarna mellan två godtyckliga positioner i området kan identifieras. Terrängobjekt, identifierade från laser-radar data, och markanvändningsdata fusioneras för att skapa en sammansatt karta av det aktuella området.</p> <p>För att skapa grafen måste kartan partitioneras i mindre områden som är homogena ur framkomlighetssynvinkel. Dessa mindre områden representeras av noder och vägar mellan områden representeras av bågar i grafen.</p> <p>Olika metoder för att partitionera kartan beskrivs och analyseras. Dijkstra's algoritmen kombineras med A*-algoritmen för att hitta de bästa vägarna mellan två positioner i det aktuella området.</p>		
Nyckelord framkomlighet, grannskapsgraf, terrängmodell, GIS		
Övriga bibliografiska uppgifter	Språk Engelska	
ISSN 1650-1942	Antal sidor: 63 s.	
Distribution enligt missiv	Pris: Enligt prislista	

Acknowledgements

We want to thank Erland Jungert and Fredrik Lantz for presenting the problem to us and for our open-minded discussions throughout this thesis. The work has been both challenging and interesting. Also, Thomas Johansson and Åsa Swahn have been helpful with proofreading and corrections during the production of this thesis, thank you.

We also want to thank Jiri Trnka for advice and technical support.

This report has also been published at Linköping University, Department of Computer Science, LITH-IDA-EX--05/047--SE.

Contents

1	Introduction	11
1.1	Problem description.....	11
1.2	Delimitations	11
1.3	Method.....	11
2	Theoretical basis.....	13
2.1	Development environment	13
2.1.1	API - MapObjects	13
2.2	Data structures	13
2.2.1	Feature.....	13
2.2.2	FeatureClass.....	13
2.2.3	Point	14
2.2.4	Polyline	14
2.2.5	Polygon.....	14
2.2.6	Shapefile format.....	15
2.3	Graph theory.....	15
3	Related work.....	17
4	Process overview	19
4.1	Data pre-processing	20
4.2	Map data merging.....	20
4.3	Map data partitioning	20
4.4	Graph creation.....	21
4.5	Graph search.....	21
5	Data pre-processing	23
5.1	Data requirements.....	23
5.2	Data structure.....	24
5.3	Polygon generation.....	24
5.3.1	Centre line adjustment	25
5.3.2	Centre line validation.....	25
5.4	Overlap removal.....	25
5.4.1	Process description.....	26
5.4.2	Sliver polygons	27
5.5	Creating attributes	28
6	Map data merging	29
6.1	Land use map.....	29
6.2	Overlay creation.....	30
6.2.1	Process description.....	30

7	Map data partitioning	33
7.1	The partitioning mechanism.....	34
7.1.1	The cut lines mechanism.....	34
7.2	The partitioning strategy.....	36
7.3	The split point strategy.....	36
7.3.1	The binary variant.....	36
7.3.2	The N-ary variant.....	37
7.4	Results.....	39
8	The neighbourhood graph	41
8.1	Graph generation.....	41
8.1.1	Node generation.....	41
8.1.2	Edge generation.....	42
8.1.3	Cost function.....	43
8.1.4	Results.....	43
8.2	Graph search.....	43
8.2.1	Results.....	46
9	Discussion	47
9.1	Conclusions.....	47
9.2	Future research.....	47
9.2.1	Polygon generation.....	47
9.2.2	Partitioning and graph-forming.....	47
9.2.3	Robustness.....	48
9.3	Closing words.....	47
	References	49
	Thesis specification (Swedish)	53
	MapDataManager	54
	Architecture.....	54
	Portability.....	54
	Installation.....	55
	User interface.....	56
	Menus.....	56
	Normal usage.....	59
	Grapher	60
	User interface.....	60
	Partitioning.....	61
	Graph creation.....	61
	Graph search.....	62

Introduction

Trafficability analysis, also called driveability analysis, in off-road terrain is an important decision support for all activities concerning movement in terrain. This kind of analysis, seen from a military perspective, is needed both to get an opinion of others' ability of movement and for planning your own off-road routes.

At the *Swedish Defence Research Agency* (FOI) in Linköping, a method for analysis of driveability [2] in different parts of terrain has previously been developed. The method involves the creation of a driveability map, a coloured map that tells where certain vehicles can or cannot drive.

Within FOI projects *Information system for target recognition* (ISM) [1] and its successor *Information system for ground surveillance* (IS-MS), a decision support system is being developed to treat, among other things, trafficability analyses.

1.1 Problem description

The task is to develop a software prototype that can determine the best paths between two delimited areas in arbitrary terrain. The software should use a network-based model to represent the different areas and how they are connected. This can be done with a graph, where nodes represent areas, and edges represent adjacency relations. This kind of graph will be referred to as a *neighbourhood graph*.

Terrain features, such as ditches and hills, have been identified using a high-resolution terrain elevation model with an existing identification tool [3]. Other geographical data, such as *land use* data, is taken from existing digital maps [14]. These data sources are used when creating the neighbourhood graph.

An initial task is to create a procedure that can extract data from the terrain feature identification tool [3], and convert them into a manageable format.

The resulting analysis should be visualized on a map, where the best paths between two given delimited areas are highlighted.

1.1.1 Delimitations

When searching for the best paths, it is desirable to find the most distinctly different paths. Distinctly different paths are paths that, for instance, take another route around a hill or lake. In a graph, two paths might differ from each other in only one node. This difference does not guarantee that the paths are distinctly different. This problem is not addressed in this work; different paths may differ in only one node.

In this work, movement within different areas is associated with a cost that is dependent on a vehicle. The associated costs will be calculated for a single type of example-vehicle only.

1.3 Method

The procedure in this Master's thesis has been first to study related work in the area of off-road trafficability. Secondly, to study different strategies for partitioning spatial data. Finally, the implementation of a tool to partition spatial data and create a graph from the partitioned spatial data is achieved. In parallel to creating this tool, another tool was created that can extract data from the driveability analysis [2], manipulate and merge map data with different contents.

Both [2] and [3] are developed with MapObjects for Java, see chapter 0, so consequently, MapObjects was used here as well.

2 Theoretical basis

This chapter gives the reader the theoretical basis needed to understand topics discussed in coming chapters.

1.2 Development environment

2.1.1 API - MapObjects

MapObjects - Java Edition is a powerful collection of client- and serverside components used to build custom, cross-platform mapping and *geographic information system* (GIS) applications [10]. MapObjects is developed by ESRI, see [10].

It has a large collection of *Java beans*, which can be used for building user interfaces. These beans offer basic map handling functionality such as map viewing, layer management and toolbars.

MapObjects offers support for standardized file formats, e.g. shapefile.

It supports vector data analysis and manipulation, such as spatial queries and set operations, e.g. union and intersection of polygons.

2.2 Data structures

The definitions and descriptions of data structures and primitives are given in a GIS/ESRI/MapObjects related context and may differ from the general view.

2.2.1 Feature

Feature is a collective term used to refer to points, lines and polygons [5].

Conceptually, a feature corresponds to an individual row in a shapefile or *FeatureClass* table containing spatial geometries, e.g. polygons, and related attribute data, ordered in fields, see Figure 1 and chapter 0. Three types of fields are mandatory; an ID field, a SHAPE field and a data field with arbitrary name and of arbitrary type, e.g. String.

2.2.2 FeatureClass

A *FeatureClass* is a set of features, where each feature conforms to the same field definition. The type of geometrical shape should also match, e.g. each feature must refer to a polygon. For more details see [5].

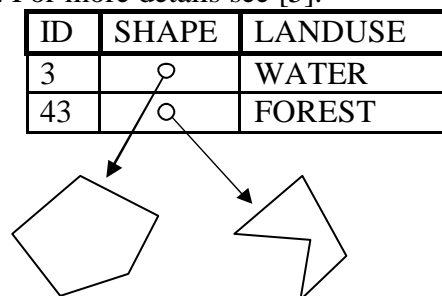


Figure 1: A FeatureClass illustrated as a table. With two Feature entries (rows) and one data attribute, LANDUSE. A field collection defines the header row.

2.2.3 Point

A point consists of a pair of double-precision coordinates in the order X, Y [6]. Points are, for instance, used to represent stones.

2.2.4 Polyline

A polyline is an ordered set of vertices and consists of at least one part [6]. A part is a sequence of at least two vertices (connected with line segments), see Figure 2. Parts may or may not be connected to one another. Parts may or may not intersect each other. Polylines are, for instance, used to represent roads.

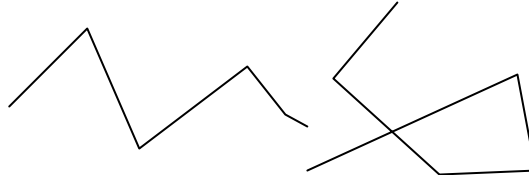


Figure 2: A single two-part polyline.

2.2.5 Polygon

Polygons are used to represent surface areas of various shapes, such as lakes. A polygon consists of at least one ring [6]. A ring is a connected sequence of at least four vertices, which form a closed, non-self-intersecting loop. Note that the first and last vertex of a ring have the same geometrical position, see Figure 3. This means that a ring has three or more connected line segments. A polygon may consist of multiple exterior and interior rings. Interior rings are also known as *holes*. Interior rings must be located on polygon surfaces.

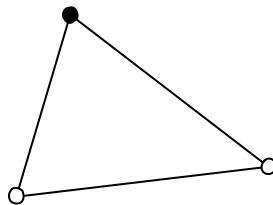


Figure 3: Ring with four vertices and three line segments. The filled point shows the coinciding first and last vertex of the ring.

An exterior ring defines a polygon surface; an interior ring defines a hole in such a surface, see Figure 4. Polygons consisting of one exterior ring are referred to as *simple*; polygons with more than one ring are referred to as *complex*.

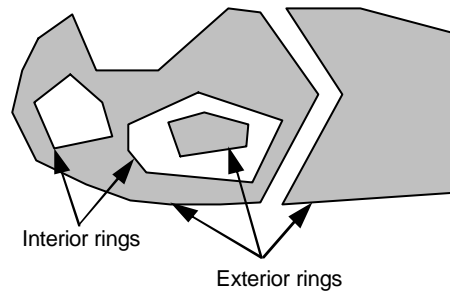
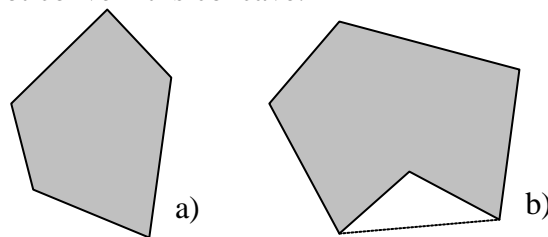


Figure 4: A single complex polygon with multiple interior and exterior rings.

A convex polygon is a polygon in which no line segment connecting two vertices is outside the polygon, see Figure 5. A polygon with holes is never convex. If a simple polygon is not convex it is concave.



**Figure 5: a) A convex polygon.
b) A simple concave polygon. The dotted line segment, connecting two vertices, is outside the polygon surface, which contradicts its convexity.**

2.2.6 Shapefile format

There are a number of different file formats for storing map data. One is the *shapefile* format, see [6]. MapObjects has extensive support for this format. Shapefiles are used to store geometry and attribute information. The shapefile format consists of three mandatory files with the following suffixes: .shp, .dbf and .shx. The .shp file contains records of vector data (geometries). One of these records can, for instance, represent a polygon or a polyline. The .dbf file contains attribute tables for the geometries in the .shp file. The .shx file contains an index table of the records in the .shp file.

2.3 Graph theory

A graph consists of a non-empty set of nodes and a set of edges. Graphs can be used to represent objects and relations (connections) between these objects, where nodes represent objects and edges relations. For instance, cities can be seen as nodes and connecting roads as edges. Different attributes, such as costs, are usually assigned to both nodes and edges. The cost of an edge can, for instance, represent the distance between two cities that the edge connects. In this work graphs are used to represent a network of areas, where areas are represented by nodes, and border relations by edges. The border relation holds if two areas share a common border. This is further described in chapter 8. For more information about graph theory, see [16].

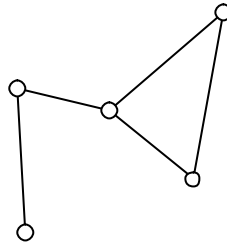


Figure 6: Graphical representation of a graph; circles are nodes, lines are edges.

Graphs can be directed or undirected. A directed graph has directed edges, which means that edges are valid in one direction only. An undirected graph has undirected edges, which are valid in both directions.

A *multigraph* is a special kind of graph, where more than one edge is allowed between two nodes. In case of a directed graph, there may be more than one edge with the same direction.

A *path* is a sequence of connected edges between two nodes in a graph. The cost of a path is the sum of all individual costs of nodes and edges used in the path.

A *least cost path*, between two given nodes, is a path whose cost is lower than the cost of all other paths between these nodes.

A *planar graph* is a graph which can be drawn in the plane in such a way that no edges cross each other. In such a graph, if it is undirected and loop-free, $e \leq 3v - 6$, where e is the number of edges and v is the number of vertices.

Dijkstra's algorithm, see [16], finds the least cost path between one node and all other nodes in a graph. The different paths can be stored efficiently with *back edges* for each node, i.e. each node stores a reference to the preceding edge in the least cost path.

A^* (pronounced A-star), see [17], can be used for finding the K least cost paths between two nodes in a graph. A^* uses a *heuristic function* to guide its search, and its efficiency is highly dependent on this function. The heuristic function should give an optimistic approximation of the cost to reach the destination from a given node, and 0 if the given node is the destination node. If so, the function is called *admissible*, and results in an optimal path instead of an approximately optimal path. When dealing with networks with geometrical interpretation, a heuristic function based on the geometrical distance between a given node and the destination is often used. This is straightforward if the edge cost is closely related to the length of the edges, if it is not; such a heuristic function becomes less useful.

3 Related work

Off-road trafficability has been treated in many different applications. The following ones have been our main source of inspiration. Similarities and differences between these and this work are discussed here. Besides trafficability, some of them also treat partitioning and graph search.

Donlon and Forbus [8] have shown how to use a GIS for reasoning about trafficability. They define trafficability as “a measure of the capability for vehicular movement through some region”. Since a vehicle is involved, trafficability is a relationship between the vehicle and the area through which it moves. They define many factors that trafficability relies on, such as slope, soil factors and vegetation. Donlon and Forbus partition space according to these factors and create an overlay^a to identify areas with homogenous characteristics. Donlon and Forbus work is similar to this work in the sense that both treat off-road trafficability. However, in this work a network describing the terrain is created, a network that can be used to calculate different paths. Their result is a generated overlay, which can be seen as a map with trafficable areas for a given vehicle.

Glinton et. al. [9], [13] considers the *Intelligence Preparation of the Battlefield* (IPB). IPB is a pre-process to military operations that involves gathering, analysis and organisation of intelligence. A number of overlays, each of which describes untrafficable terrain are combined to show all obstacles. This combined obstacle overlay (COO) tells at a glance the ease of movement for a given vehicle. Military vehicles often move in groups (formations). Finding suitable paths through terrain for such units is similar to a max-flow problem. Glinton et. al. partition space using *Voronoi diagrams*^b, which gives a network-like appearance of the free space. This network represents corridors that are possible to traffic and can be compared to an electric circuit with a resistor for each corridor. A narrow corridor can be compared to a resistor with high resistance.

The work of Glinton et. al. is relevant to this problem since it partitions space and creates a network. Their network is used to find suitable corridors for multiple vehicles that often travel in a formation. In this work, a network for finding the best path for a single vehicle is created. Glinton et. al. can probably find the best path for one unit (a vehicle) with their network. They base their network on obstacles and free space in contrast to the approach of this work which is based on terrain features such as ditches. This will lead to a network that can be used for calculating any vehicles' trafficability. Trafficability depends on many different factors such as soil type and vegetation. Calculation in this work will have the possibility of considering such facts instead of treating all obstacle-free areas uniformly.

Holmes and Jungert [7] have developed two different methods for route planning within digitized maps. Both methods employ A* search in a graph, a graph which represent how tiles of free space are connected in the original map information. Their work is restricted to 2-D binary maps. Binary maps have a representation of

^a An overlay is in this military sense a transparent sheet where relevant characteristics are registered from the underlying map [8].

^b A Voronoi diagram of a collection of geometric objects is a partition of space into cells, each of which consist of the points closer to one particular object than to any other [11], [12].

only two types of space, which either are free or occupied (obstacles). Because of this restriction, there are only two costs associated with movement within these regions. Movement within free space is of uniform cost. Movement within obstacles is not possible, why such regions impose an infinite cost. As an example, consider route planning at sea with islands representing occupied space and sea representing free space.

In [7] a simplified nautical map containing only water and land is used. To create a searchable graph, a partitioning of free space (water) into smaller parts called *tiles* is performed. The partitioning is done in such a way that distinct paths are created around islands. Movement through a number of tiles will represent a rough route. A *tile graph* is created with nodes representing each tile. Two different methods were also implemented, where both find the *shortest* path from one point to another.

In contrast to this work, [7] and all the other works presented above treat the information in a binary manner with free space and obstacles. Most interesting is how Holmes and Jungert partition data using horizontal lines. Their partitioning strategy is similar to the one used in this work.

4 Process overview

This chapter gives an overview of this work, and in that way also describes the solution to the problem. The chapter briefly describes the purpose of each step which is then further described in subsequent chapters.

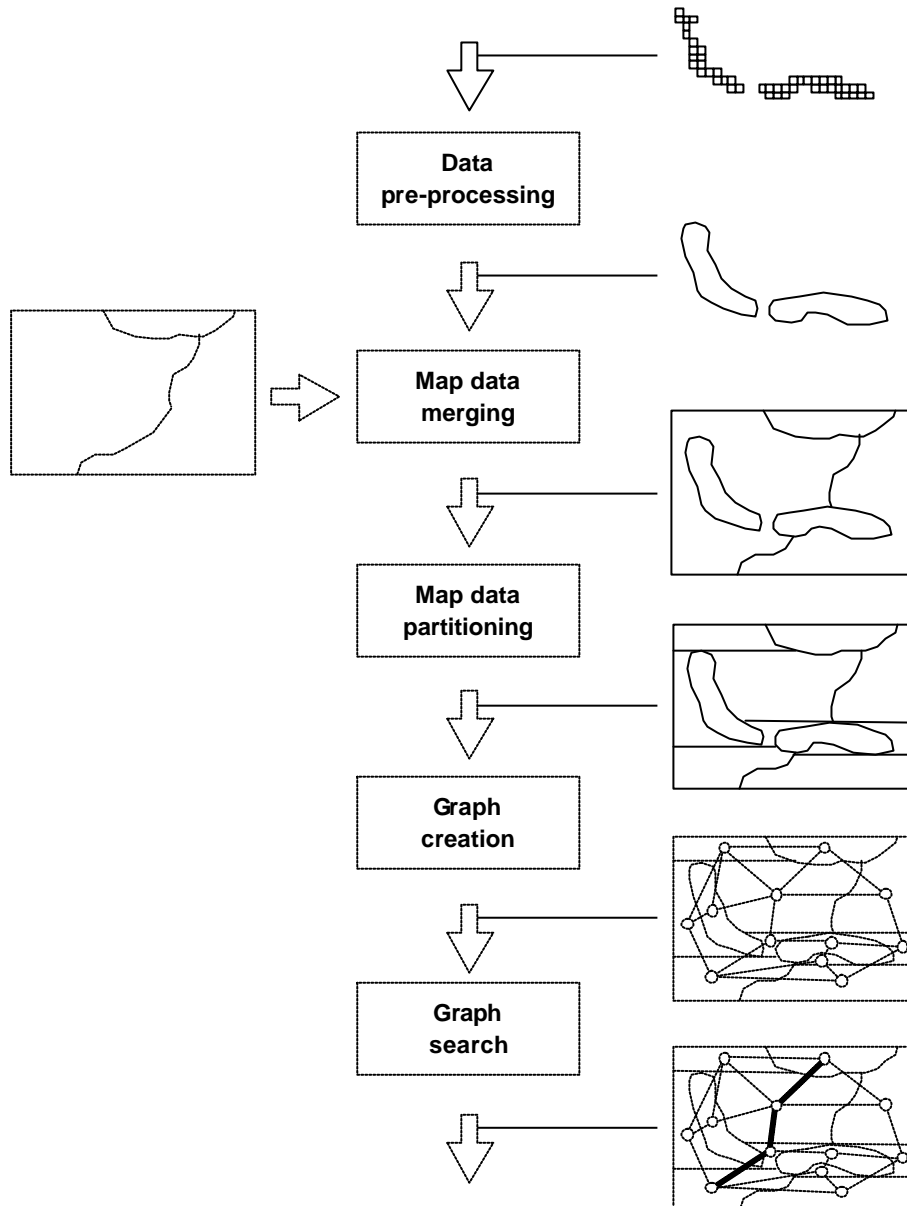


Figure 7: The work can be seen as a number of steps (centre column). Each step results in different output that serves as input to the next step.

The problem is to, from identified objects, create a network represented as a graph. Nodes in the graph represent identified objects and edges represent possible pathways between objects. A search algorithm should be used to find the best routes between two nodes in the graph.

The solution is a system containing a number of steps where each step depends on the preceding steps. The different steps can be seen in Figure 7.

4.1 Data pre-processing

Available from [2] are two kinds of identified objects, ditches and slopes. These objects are represented by a set of 2×2 m squares that have been grouped together. The objects are extracted from a Java program developed in [2], to obtain better performance in calculation and visualization, see chapter 0. These objects are given a new representation in this step.

The expected outcome of this data pre-processing step is map data that represent the objects by means of polygons.

Every polygon will represent an area with uniform properties. Depending on those properties, the cost of driving through different areas with a vehicle can be calculated by a cost function.

4.2 Map data merging

Data from the pre-processing step needs to be complemented with additional information, partly to replace possible non-identified areas (holes), and partly to add more attributes to the identified objects. The additional information, which is merged with the data from the previous step, comes from a land use map, see chapter 0.

The output of this step will, in contrast to the input, have full coverage of the surface (no gaps).

4.3 Map data partitioning

The concept of having polygons representing areas with uniform properties, can result in map data with detached areas such as a set of islands, see Figure 8a. Such data must be partitioned as described in chapter 7 before proceeding to the next step of this process, building a graph.

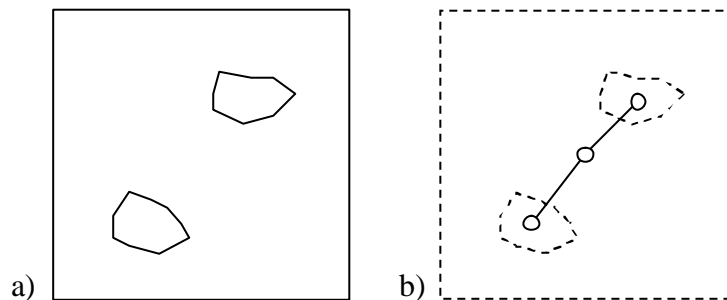


Figure 8: a) Map data containing three elements; two islands and one surrounding area.
b) A graph representing the underlying areas.

The graph consists of nodes that correspond to areas, and edges that represent paths between these areas. Such a graph, representing the areas in Figure 8a, would only have three nodes, see Figure 8b. Each island is represented as a node. The surrounding area is represented as a node as well.

If a user of the system would like to calculate the best path from the lower left corner of the map to the upper right corner, see Figure 8b. The graph has only one node to represent the surrounding area which implies that the start and end points of this search are the same. Another requirement of the system is the ability to describe different pathways around objects. This is not possible for map data like

in Figure 8a. Another strategy is to create a graph that better represents the possible movements in the underlying map. See Figure 9 for an example of a graph created from partitioned map data.

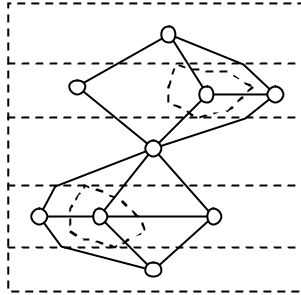


Figure 9: A graph created from partitioned map data.

The output of this step is partitioned map data that can be used to create a graph corresponding to that data in a manner that will be described in chapter 7.

4.4 Graph creation

The purpose of creating the graph is to have a searchable data structure. For details see chapter 8. The output of this step is a graph that represents the areas of the different objects with nodes and possible pathways between areas with edges.

4.5 Graph search

The purpose of the graph search is to find a number of possible routes by means of a search algorithm. Two algorithms that together find the *k best routes* in the graph are used, see chapter 8.

A source and a destination node must be supplied to perform a search. The output is a route highlighted on the underlying map data.

5 Data pre-processing

A work by Sjövall [3] resulted in a Java program called *Category Viewer (CV)* [3] that can recognize geographical features such as ditches, roads and ponds from *laser radar*^c data. The laser radar data is transformed into 2×2m tiles where each tile is matched to a specific category [3]. These categorized tiles are then used together with a number of filters to recognize different objects. A recognized object from CV, like a ditch, is represented by a number of grouped 2×2m tiles.

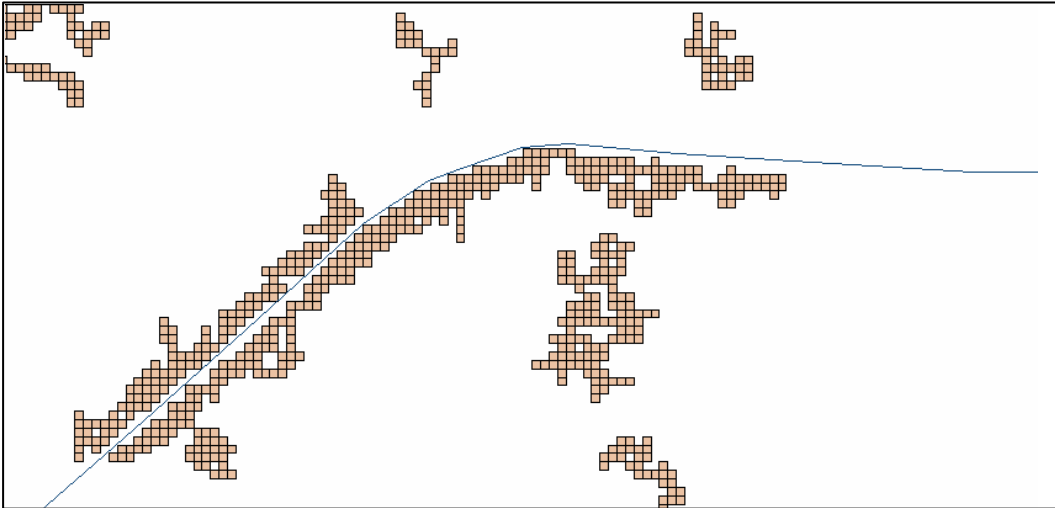


Figure 10: Category Viewer can recognize different geographical features. In this picture, two ditches have been recognized (shown together with the centre line of a road). The other surrounding objects are cavities that have been recognized in the same process.

Some of these recognized objects (ditches and slopes) were further analyzed together with different vehicle properties by Edlund [2] to decide where various vehicles can drive. The result was called *Driveability Viewer (DV)*, a Java program that uses colouring to visualize the driveable areas in a *driveability map*.

5.1 Data requirements

In this work it was desirable to continue working on data from the two previous works [2], [3]. Another representation of data than the output of CV and DV (grouped 2×2m squares) is needed for the following reasons:

- A single ditch from CV and DV can be made by hundreds of grouped squares (polygons). When working with most of the polygon-related operations in MapObjects, like subtraction or union, the demand for computer resources (memory and processing time) will increase as the number of polygons grows [4].
- Visualizing objects represented by hundreds of squares is also computationally demanding and is far too detailed for most applications.

^c *Laser radar* is a high-resolution surveying technique, often used in land and ocean surveying.

- Besides the computational disadvantages, the objects need a more generalized shape that is easier to distinguish when visualized. Also, a shape that better represents the true objects is required.

A new form of representation is required, in order to be able to do faster geometric operations and improved visualization.

5.2 Data structure

GIS-related software usually has geometries (lines, points and polygons) stored in a database. A typical database record of a polygon consists of all coordinate pairs defining its vertices and some unique id of the polygon. By connecting a table of geometries with information from another table, database queries can be used for fast geographical analyses.

MO offers a data structure called *FeatureClass* (see chapter 0) for storage of geometrical objects and their attributes in a table (similar to database tables). This data structure allows for many convenient built-in methods to be used. GIS-related queries can also be executed on a *FeatureClass* because its structure is similar to a database. MO provides functionality for saving a data structure to file, in the *shapefile* format. Saving in this format makes it possible to open and use the data structure in other GIS-related software. This data structure has been used throughout most parts of this work, because of these advantages.

For debugging purposes, open source software called *JUMP Workbench* [15] has been used. *JUMP* can be used for viewing, creating and manipulating spatial data. This is very convenient since it supports the *shapefile* format as well.

5.3 Polygon generation

MapObjects provides functionality for generating buffers around lines, points and polygons [5]. There is a function that, given a radius called *buffer distance*, returns a polygon that represents the outer shell of the geometry.

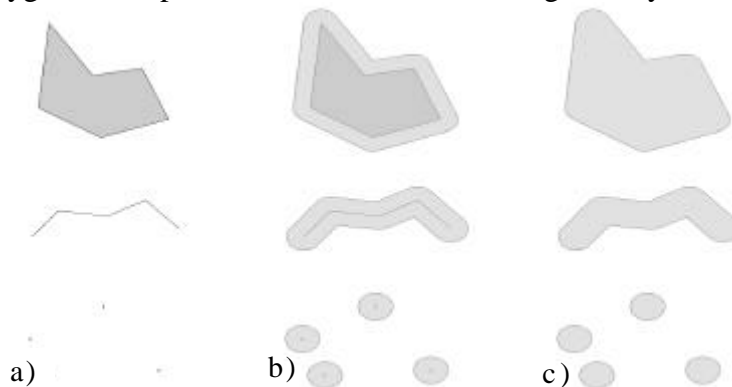


Figure 11 a) Example of geometries that can be used with MapObjects buffer function: polygons, polylines and points.
 b) An outer shell is created around the geometries with the buffer function.
 c) The resulting buffers.

DV provides functionality for calculation of centre lines from the recognized objects. Buffering those centre lines results in a new representation that is less detailed than the square-shaped objects, see Figure 12.

After initial prototyping, the representation from MO's buffer functionality was chosen instead of the square-shape since it met the requirements for visualization and is less computationally demanding.

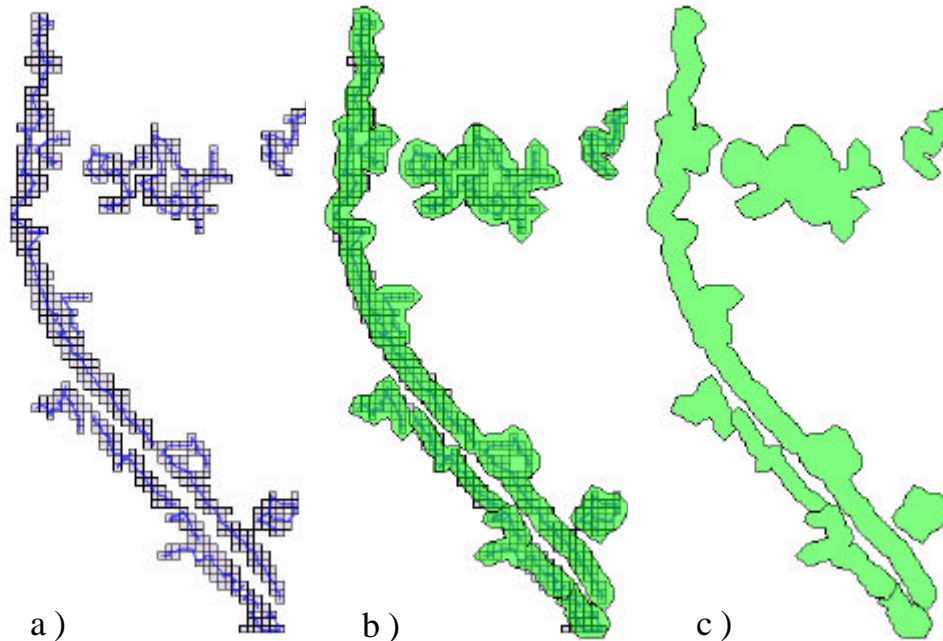


Figure 12: a) Two square-shaped ditches together with some cavities generated by DV together with the calculated centre lines.
 b) MapObjects buffering function applied to the centre lines.
 c) The resulting buffer is a new representation that is less computationally demanding and is easier to distinguish from other objects when visualized.

5.3.1 Centre line adjustment

DV's functionality for calculation of centre lines also enables adjustment of their level of detail. The centre lines have an effect on the generated buffers. If a centre line consists of many vertices (highly detailed) the corresponding buffer will also be very detailed. If a centre line is too generalised the corresponding buffer will not be a good abstraction of the underlying square-shaped representation. The appearance and detail level of the centre lines have been adjusted to a level suitable for our needs by adjusting available parameters in DV.

5.3.2 Centre line validation

The buffer function used, from MO, is not robust. When trying to buffer a polyline that has self-intersecting segments, the result is undefined. In the worst case, such a polyline may lead to a situation where the buffer function never terminates. Therefore, centre lines are validated at an early stage to avoid problems with the buffer function.

5.4 Overlap removal

When buffering a centre line, a value of the buffer radius (buffer distance) must be chosen. An estimate of the width is calculated and collected from DV. The generated buffers may overlap each other since the given width only is an

estimate. This overlap has to be removed since the map partitioning process, as well as many other polygon related operations, needs a well-defined data for robustness and to work properly. The result from all these operations are undefined if map data has overlaps.

This work treats two types of identified objects, *ditches* and, the more loosely defined, *slopes*. Firstly, ditches may overlap each other as shown in Figure 13. Secondly, slopes may also overlap each other. Furthermore, ditches and slopes may overlap each other, and in some cases much more than shown in Figure 13.

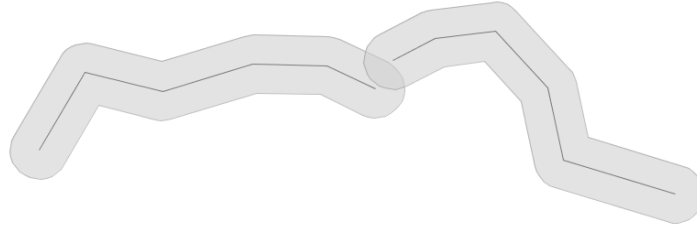


Figure 13: Generated buffers can overlap.

5.4.1 Process description

The following process for overlap removal has been inspired by the *polygon overlay process* described in [4]. Here is a brief description:

1. Firstly, make a coarse test to see if two polygons overlap by comparing their enclosing rectangles, see Figure 14b. This test is quick compared to the test in step 2.
2. An exact check for overlap with a *point-in-polygon* (PIP) process is done, if the enclosing rectangles overlapped, see Figure 14c. The PIP process [4] verifies whether a given point is inside a polygon or not. In this case, when checking for overlap, every vertex of one polygon must be compared to the other polygon. The PIP process is done by first creating a half-line^d from the considered vertex. It can be determined whether the point is inside or outside the polygon by counting the number of intersections of this half-line. A point inside a polygon has an odd number of intersections.
3. When removing overlap from two polygons, one of them has precedence and will not be cut. This is accomplished by checking the polygons' attribute tables. For example, the ditch with the highest *maxslope* will have precedence over the other ditch.
4. When an overlap has been identified, the removal starts by identifying where the two polygon boundaries intersect with each other. See Figure 14d.
5. Where an intersection is found, new vertices are inserted to both polygons so that new line segments are created. The polygon with precedence (in Figure 14, the right one has precedence) is now completed and the other

^d A half-line, in this context, is a line projected in only one direction from the starting point.

one will lose all points identified as overlapping in step 3. The result can be seen in Figure 14e.

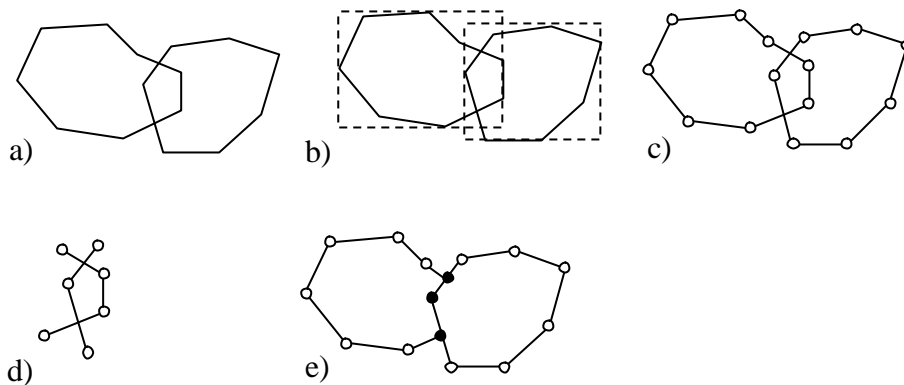


Figure 14: a) Two overlapping polygons.
 b) Checking for overlap with enclosing rectangles.
 c) Checking for overlap with a point-in-polygon process.
 d) Checking line intersections to determine new vertices.
 e) The resulting polygons have three coinciding vertices (in black) .

This process is complex to implement. To save time, most of the steps have been implemented using *set operations* provided by MapObjects.

5.4.2 Sliver polygons

When dealing with polygon-related operations, *sliver polygons* [4] can arise, see Figure 15. Sliver polygons are polygons that, instead of having coinciding segments, have thin slices of empty space or overlap between them. Using built-in operations from MapObjects, overlap removal has shown to be unstable and can in special cases generate sliver polygons. Sliver polygons are a common issue for geometric operations. These kind of erroneous polygons must be taken care of to ensure robustness of other polygon operations. A proposed solution is to identify the sliver polygons and repair them [4].

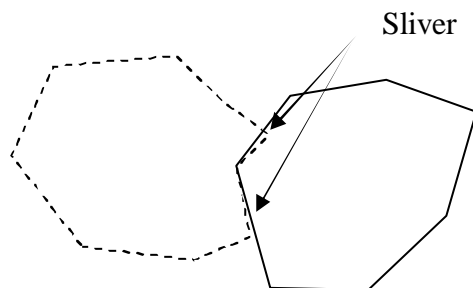


Figure 15: Sliver polygons are polygons that, instead of having coinciding segments, have thin slices of empty space or overlap between them.

There are many reasons why sliver polygons can arise. When removing overlaps, there is a calculation to decide where line segments cross and where to put new vertices, see Figure 14d-e. The result is dependant on the accuracy of the data

used (number of decimals). It is hard to guarantee an error-free result, when using some of MO's built-in operations, since their implementations are hidden.

5.5 Creating attributes

The identified objects from DV each have a number of attributes, which are stored in a data structure internal to DV. The attributes are extracted at the same time as generating polygons from the square-shaped objects in DV. The attributes and the generated polygon are stored together in a *FeatureClass*.

The following attributes are extracted from DV:

- **TYPE** Stands for "type of object" and holds the objects' type name e.g. DITCH or SLOPE.
- **ID** A FeatureClass holds a set of geometries, which must have a unique id. The attribute table of a FeatureClass already has a mandatory field that holds such an id. This field is a copy of that field and is used for debugging purposes only.
- **WIDTH** A value (in meters) that is calculated from the 2×2m square representation of an object by DV. The value represents an average width of the groups of squares. This value is used by the buffer function described earlier to generate buffers with corresponding size.
- **MAXSLOPE** A value (in degrees) representing the highest slope within an object. This is used when removing overlaps to determine which object has precedence.
- **MAINDIR** A value that is not used, representing the main direction (north-south etc.) of an object. For more information see [2].

These are all the attributes available from DV. Both *ditch* and *slope* objects use these attributes.

6 Map data merging

In this work, all available information from DV is put together, creating an overlay map as seen in Figure 16.

The data from DV contains only two kinds of recognized objects, ditches and slopes. When considering a map with only such objects, there will be several unknown areas (gaps) around these.

In order to associate those areas with appropriate costs in the network, see chapter 8, some additional information must be added to these unknown areas.

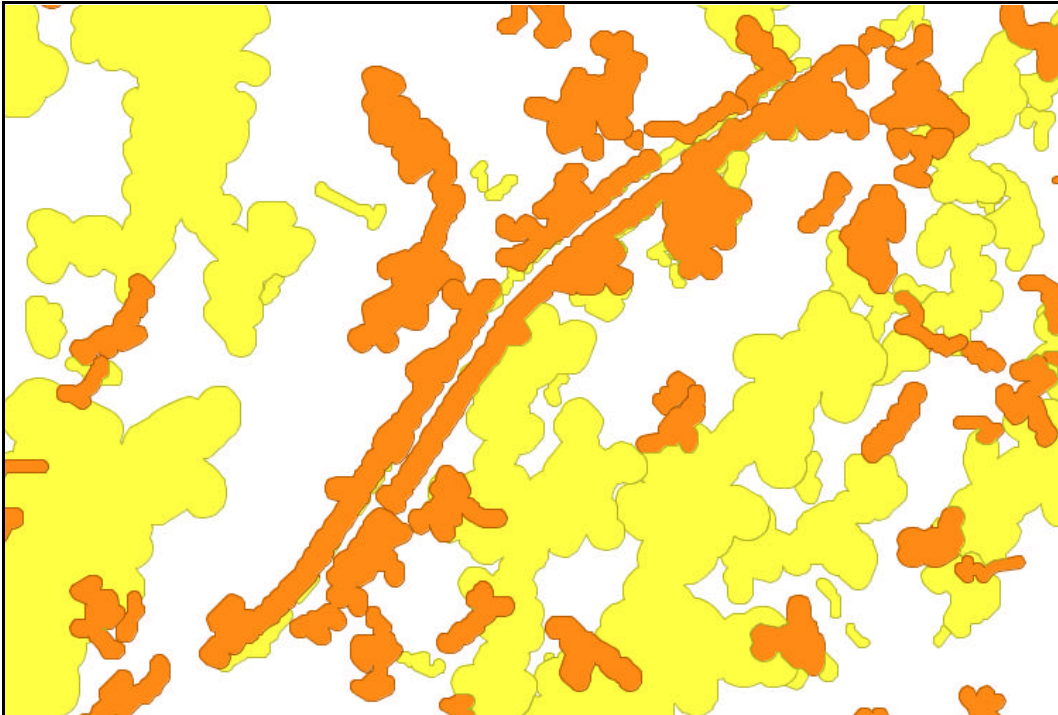
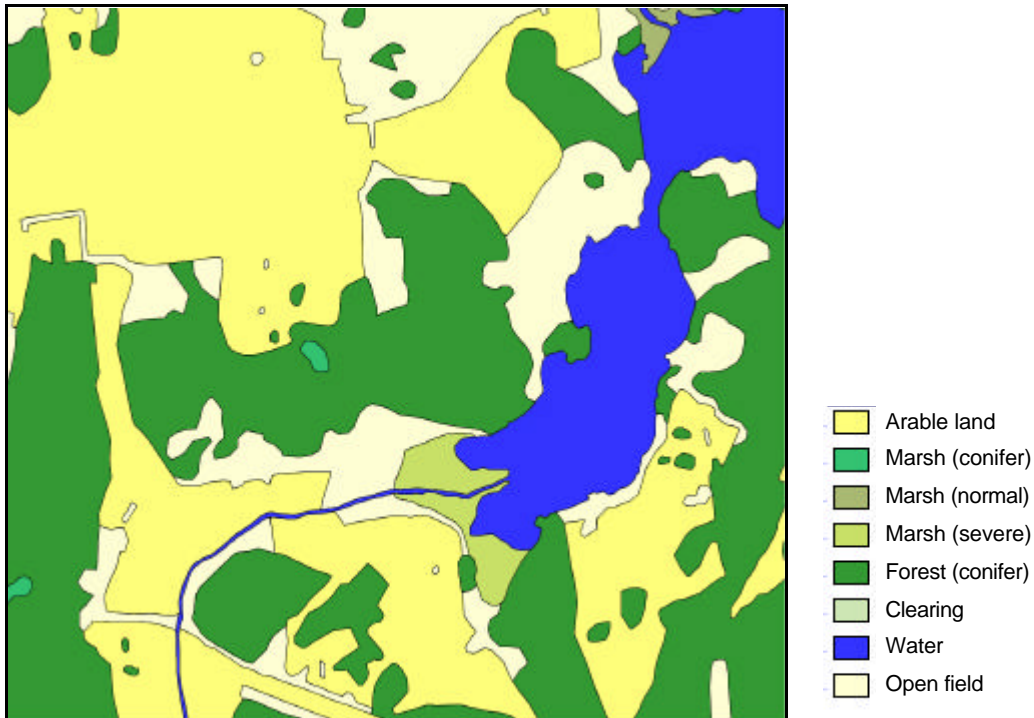


Figure 16: A map of identified ditches (dark) and slopes (light) from DV do not totally cover the surface. More information must be added to create a map with full coverage of the surface.

6.1 Land use map

Several related works [2], [8] have identified soil factors and type of vegetation as important factors when analysing trafficability. A natural way of giving the non-recognized areas relevant information is therefore to replace them with a map of land use (LUM). A LUM describes which parts consist of forest, arable land, residential construction etc.



Map copyright Lantmäteriverket 2001, ärende nr L2002/308

Figure 17: A land use map showing arable land, forest, water etc.

If the information from a LUM is merged with data generated by DV, the LUM will fill in all unrecognized areas and provide additional information. This means that a map will be created that can tell which ditches are located in forest and which are located in arable land. The LUM provides important information about the identified objects, so the map of ditches is merged with a LUM. The LUM used comes from *Geografiska SverigeData – Fastighetskartan (GSD)* [14] and was delivered in *Shapefile format* [7]. GSD contains a large collection of layers, and the LUM is a polygon layer with full surface coverage.

6.2 Overlay creation

The process of merging polygon data to create new polygons is called *the polygon overlay process* [4], which means that different map layers are merged to create a new map layer with new polygons. For vector data, this is described as “perhaps the most challenging computational requirement” in [4]. In fact, this is the most time consuming process in this work.

6.2.1 Process description

When merging different map layers, it is likely that overlaps occur. In this case, merging a LUM, that itself has full surface coverage, with another layer from DV, will create overlaps. The goal is to combine them such that polygons without overlaps are created. Here follows a brief description of the method:

1. Starting with two map layers: a LUM and a layer containing identified objects like ditches and slopes (DV-layer). Find all overlapping geometries from the LUM, for each object in the DV-layer, see Figure 18a.

2. If the DV-polygon has only one overlapping LUM-polygon, the DV-polygon is left unchanged. An attribute describing the overlapping LUM-polygon is copied to the DV-polygons attribute table. In this way, a ditch can be described as overlapped by arable land. If the DV-polygon has various overlapping LUM-polygons, the DV-polygon is partitioned into correspondingly large pieces. These pieces get corresponding attributes copied into their attribute tables. The result is a layer of ditches and slopes divided into, for example, ditch-in-forest or ditch-in-arable-land, see Figure 18b.
3. The result from step two is then used to clip a hole in the original LUM. This procedure is needed to make data fit together without errors and is the single most time consuming part of this process. Some optimization has already been done by first making a union of all ditches and slopes and using that to clip once in the LUM. The result is a LUM with holes, see Figure 18c. This operation requires about 2 hours to execute (Pentium 1.7GHz, 512MB) when using all available data from DV (ditches and slopes, 800×800m).
4. Now the results from step 2 and 3 can be merged. They should fit together like pieces in a jigsaw puzzle, see Figure 18d.

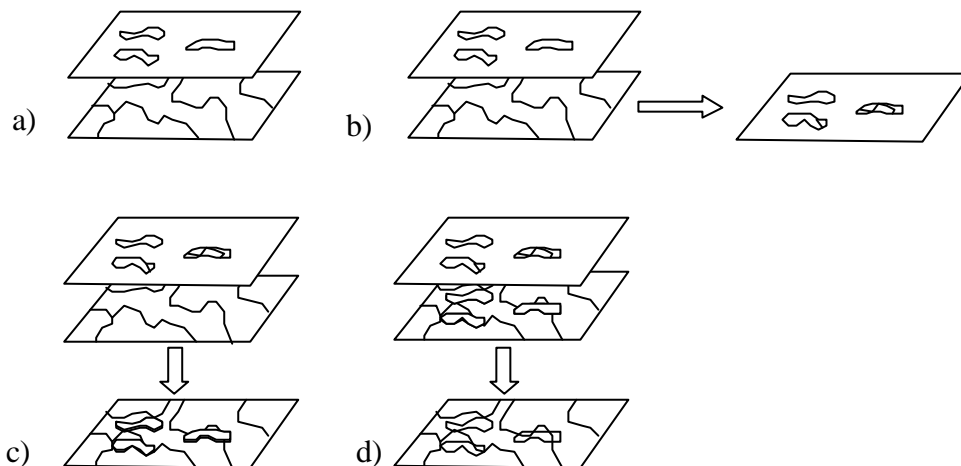


Figure 18: a) Two layers are compared to find overlapping polygons. One layer is a LUM and another layer consists of ditches and slopes.
 b) Ditches and slopes get additional information copied from the LUM. Ditches can now be identified as ditch-in-forest for example.
 c) All ditches and slopes are used to cut holes in the LUM.
 d) The results from b and c are merged like pieces in a jigsaw puzzle.

6.3 Results

The results from chapter 5 and 0 are saved to files for later use in the partitioning process described in chapter 7. An empirical analysis of the performance shows that the execution time grows rapidly with the number of polygons, see Figure 19. The performance test takes all operations from chapter 5 and 0 into account. Step 3 described in chapter 0 involves a union operation that requires approximately 99% of the execution time. The tests are performed on a computer with a 1.7GHz

Pentium CPU and 512 MB of main memory. The four different points in Figure 19 correspond to the number of polygons contained in data from 200×200m, 400×400m, 600×600m and 800×800m. Partitioning an 800×800m area consisting of about 1300 polygons takes about 120 minutes to execute.

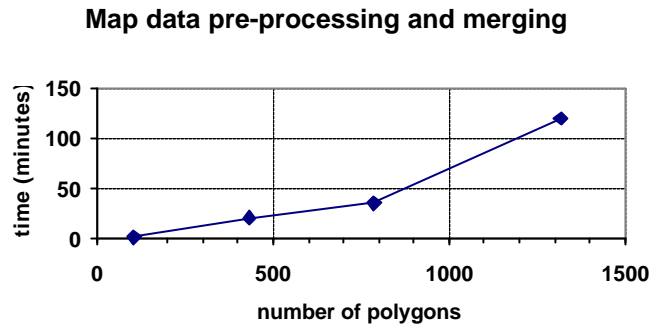


Figure 19: Performance test of the map data pre-processing and merging

7 Map data partitioning

Partitioning is the process of dividing something into parts, in this case polygon surfaces. The map data often contains polygons of complex shapes and with large vertex counts. Polygons are also often complex in the sense that they contain holes (have interior rings). These properties are undesirable for reasons that will be explained. In this work, partitioning is divided into two parts: a partitioning mechanism and a partitioning strategy. The mechanism refers to the actual process of cutting areas into pieces, whereas the strategy refers to where, at which locations, the areas are cut.

In the neighbourhood graph, each node represents a polygon and each edge a border relation. Each node has a position which is used to approximate the location of a polygon. Furthermore, each edge is used to approximate the movement across the border from one polygon to another. To make these approximations sensibly, the polygons must meet certain requirements:

- The shape of the polygons should not be too complex, i.e. the polygons should be close to convex. A convex polygon guarantees that there is a straight line between all points of the polygon, without the line intersecting the edges of the polygon, thus, guaranteeing that all borders can be reached with a straight line from the node, without the line intersecting other adjacent polygons.
- The size of the polygons should be restricted. The bigger the polygons get, the less accurate the graph nodes approximate the polygons.
- The length of connected borders between adjacent polygons should be restricted. The longer the border gets, the less accurate the edges approximate the border crossing between adjacent polygons.
- The polygons must not contain holes. This will help ensure that pathways on both sides of polygons can be represented. Furthermore, this makes it possible to implement simpler partitioning algorithms which are unable to create complex polygons.
- Unless a multigraph is used, see chapter 0, there must only be one separate border between two adjacent polygons. This will ensure that pathways on both sides of polygons can be represented, see Figure 20. In case a multigraph is used, the edges must have an attribute to keep track of which separate border is crossed. This is also useful in the non-multigraph case, but not necessary. This work does not use multigraphs.

Note that these requirements have been comprised to be suitable for the method of graph creation described in 0.

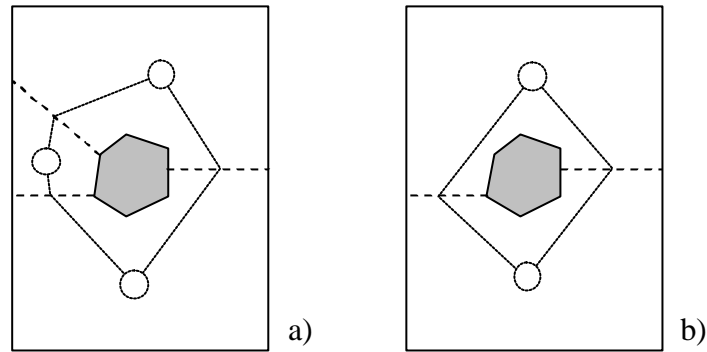


Figure 20: Possible paths are created on both sides of the grey polygon using:
 a) three adjacent polygons around the grey polygon, which limits the number of separate borders,
 b) two adjacent polygons, if a multigraph is used.

Polygons in un-partitioned map data normally meet these requirements poorly, see Figure 21. The task of the partitioning process is to divide polygons into smaller ones that better meet these requirements.



Figure 21: Example of a large and complex polygon. A single node would not be a good approximation of this polygon. This polygon may have hundreds of surrounding neighbours and several islands. If this polygon were to be represented by a single node, the edges would deviate significantly from the actual borders that the edges should approximate.

7.1 The partitioning mechanism

A tool is needed to divide areas, from map data, e.g. polygon data, into several parts. This tool is referred to as a partitioning mechanism. This work is concerned with dividing polygons, thus an algorithm for dividing polygons is needed.

7.1.1 The cut lines mechanism

This partitioning mechanism uses a set of cut lines to divide a polygon, see Figure 22. Cut lines are simply lines that define locations to cut. First the cut lines are positioned using a partitioning strategy, and then the dividing algorithm uses these lines to cut the polygon into pieces.

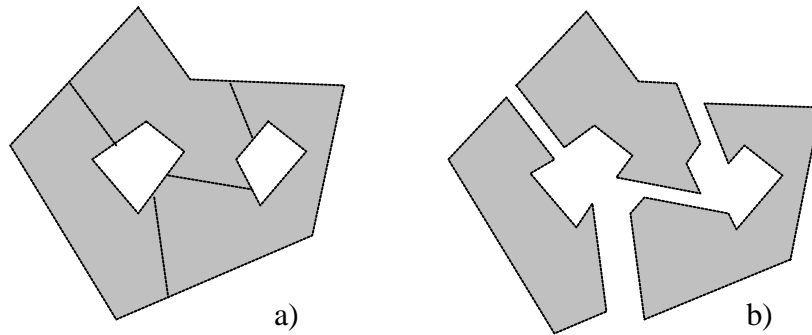


Figure 22: a) A polygon with two holes, and four cut lines denoted by the dotted lines. b) An exploded view of the polygons resulting from the cutting algorithm using the set of cut lines in *a*.

Cut lines can be placed as in Figure 22. If the cutting algorithm is to work, cut lines must be placed so that:

- no resulting polygons have holes, not as in Figure 23a; this is due to an algorithm limitation,
- the end points of cut lines touch a ring segment or ring vertex, not as in Figure 23b,
- no cut lines intersect each other or rings, not as in Figure 23c; note that touching end points is not considered intersection here,
- the cut lines must be positioned on the polygon surface, not as in Figure 23d,
- no very thin polygons or polygons with very small interior angles are created, not as in Figure 23f; this is due to a robustness limitation of the algorithm.

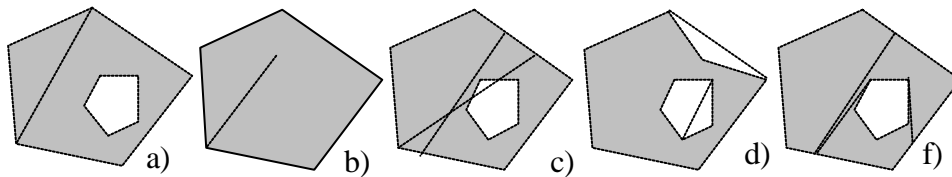


Figure 23: Some invalid cut line placements.

To avoid sliver polygons and to keep borders intact, new vertices are introduced in neighbouring polygons at the locations where cut lines have touched, see Figure 24.

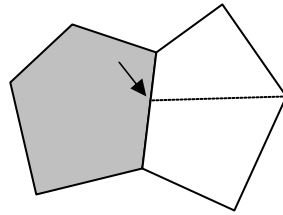


Figure 24: A new vertex is added to the grey polygon where the cut line of the white polygon touches.

7.2 The partitioning strategy

A strategy for positioning cut lines is clearly needed. All encountered partitioning strategies have been of the binary kind, i.e. an area class is either free space or obstacle. The problem in this work is not binary, but N-ary; forests, ditches, fields and roads etc. all have different trafficability. Thus an N-ary partitioning strategy had to be developed. An existing binary strategy served as a base, and was extended to cope with the N-ary case.

7.3 The split point strategy

This partitioning strategy divides areas into horizontal strips at certain locations identified as *split points*. Split points is a concept developed in a related work [7], for binary data, which had to be extended to work with N-ary data. The areas generated from the partitioning process are referred to as *tiles*.

7.3.1 The binary variant

If only two types of area classes are considered, i.e. areas which are trafficable (free space) and those that are not (obstacles), you get a binary trafficability problem. The binary variant can for instance, as in [7], be used for marine trafficability, where water is trafficable and land is not.

In [7], splitting of free space occurs at the points that, with respect to the y axis, are local maximum or local minimum points of the obstacles, see Figure 25.

These points determine the location of the split points.

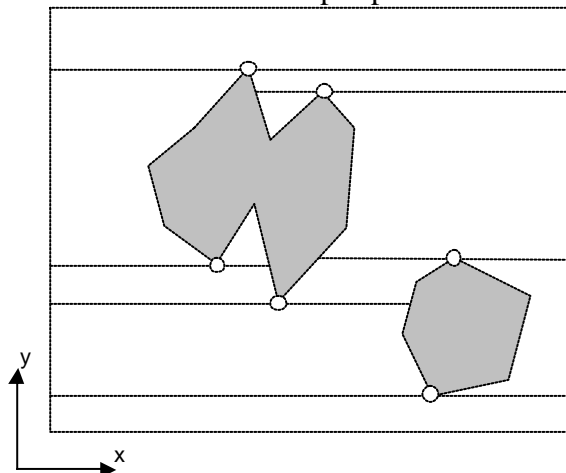


Figure 25: The grey areas are obstacles; the surrounding white space is free space; the small circles are split points. The areas enclosed by either dashed lines and/or obstacle edges and/or map edges are tiles.

This variant is binary, and only partitions free space into strips, whilst the obstacles are left intact. Further, this variant is not general since free space cannot be inside obstacles. These details make this variant unsuitable for our needs. However, it still serves as a good base, since it gives a background for our way of reasoning and a terminology.

7.3.2 The N-ary variant

This variant divides areas (polygons) into strips, just like the binary version, except that there is now no notion of free space and obstacle. Areas, which may be seen as obstacles in a binary case, are also divided.

A way of reinterpreting the definition to apply to N-ary data is required, because partitioning in the binary version originally is defined for binary data. The white, free space in Figure 25 can be seen as a polygon with two holes. These holes could be filled with obstacle polygons, shaped as each hole respectively. These obstacle polygons are not necessary in the binary case. It is sufficient to state that the free space polygon is trafficable, and that everything else, e.g. the holes, is not. This is not true for the N-ary case; everything may be traversed, so everything must be partitioned, even the polygons in the obstacle holes, see Figure 26.

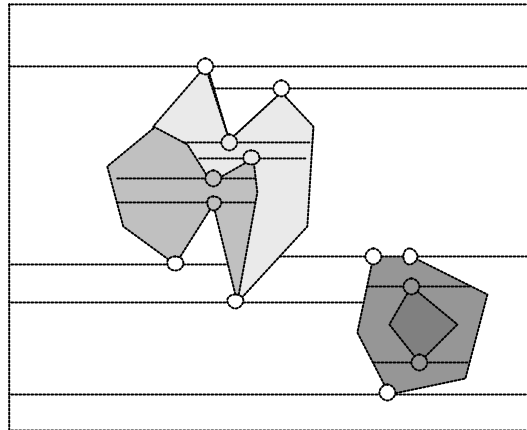


Figure 26: An example of a partition of a non binary case. The split points have been coloured according to the polygon to which they belong. Note that the white box also is a polygon with two holes in it.

The definition of a split point for the N-ary case is different from the binary case. For the N-ary case, split points are, with respect to the y-axis, at maximum and minimum vertices of the polygon rings, where the left or the right side of the vertex is on the polygon surface, see Figure 27. Note that split points and cut lines are handled on a per polygon basis, i.e. the positions of split points and cut lines of one polygon are not affected by split points and cut lines of other polygons. Each split point generates one or two cut lines. One to the left and/or one to the right of the split point, see Figure 27. The cut line is extended between a split point and the horizontally closest edge of all rings in the polygon. Sometimes a split point only generates one cut line. This happens when there are horizontal line segments in the polygon, as seen in Figure 27.

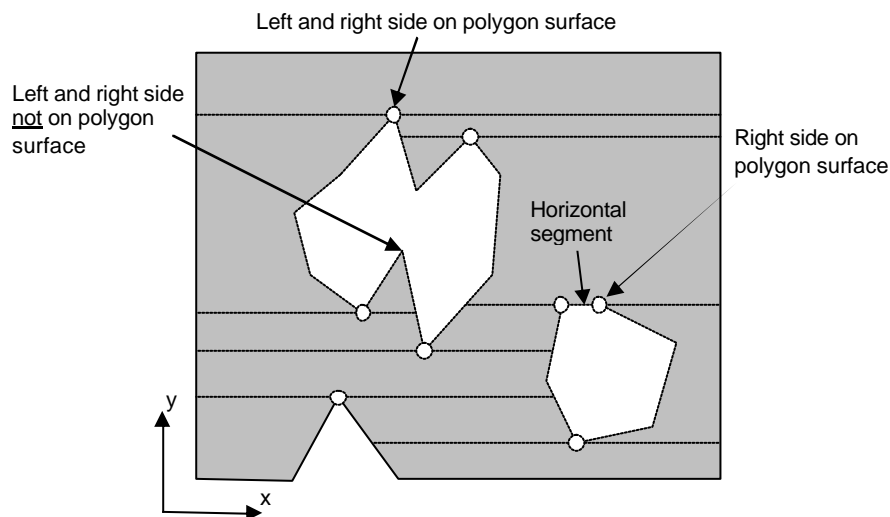


Figure 27: A single grey polygon with two white holes. The split points and the cut lines are illustrated with circles and dotted lines respectively. Note that only local extreme points with its left or right side on the polygon surface are considered split points.

7.4 Results

A theoretical analysis of the time complexity of the partitioning algorithm is difficult to make. There are many factors that influence the result, such as number of polygons, number of vertices in polygons, number of neighbours of each polygon, etc. In addition, some sub-routines are developed by a third party, thus the time complexity of these routines can sometimes only be guessed. However, an empirical analysis can be used to measure the performance of the algorithm on selected data, see Figure 35. The tests are performed using a computer with a 440 MHz UltraSPARC-IIi CPU and 384 MB of main memory.

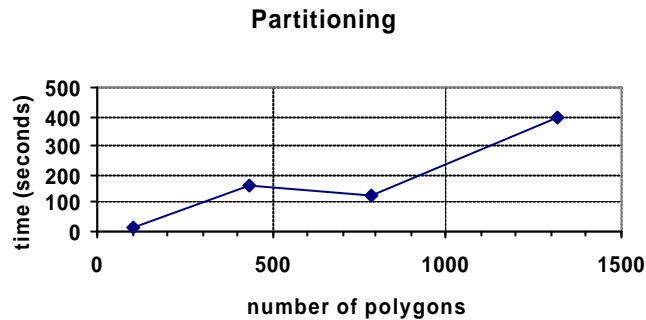


Figure 28: Performance test of the partitioning algorithm.

Partitioning of an 800×800m area, consisting of about 1300 polygons, results in about 9200 polygons and takes approximately seven minutes to calculate, see Figure 29.

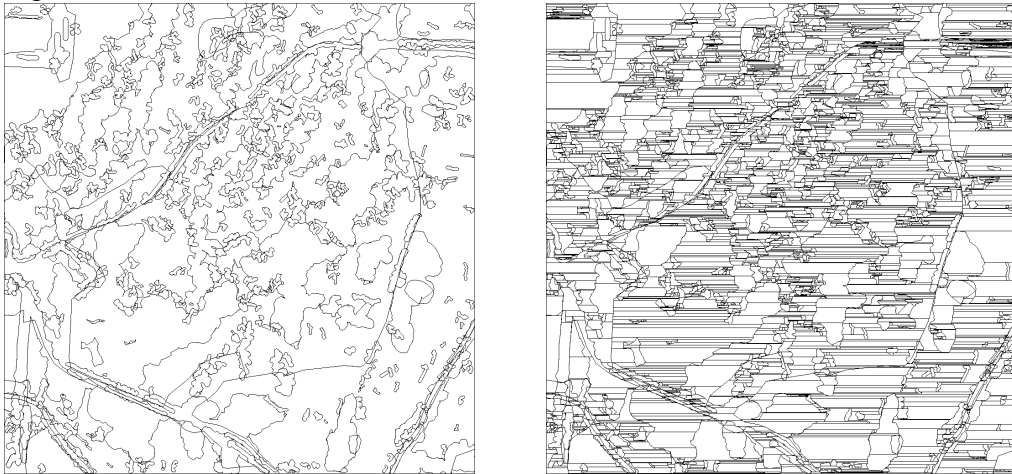


Figure 29: To the left: An 800×800m area of un-partitioned polygon data. To the right: The polygon data partitioned using the split point strategy.

8 The neighbourhood graph

The purpose of the neighbourhood graph is to describe areas in a map, and how these areas are connected. In this work, the areas are represented by polygons. In the graph, each node represents a polygon and each edge a border relation, see Figure 30. The position of each node is used to approximate the location of a polygon. Furthermore, each edge is used to approximate the movement across the border from one polygon to another. An authentic example of a neighbourhood graph created for a partitioned map is shown in Figure 36 at the end of this chapter.

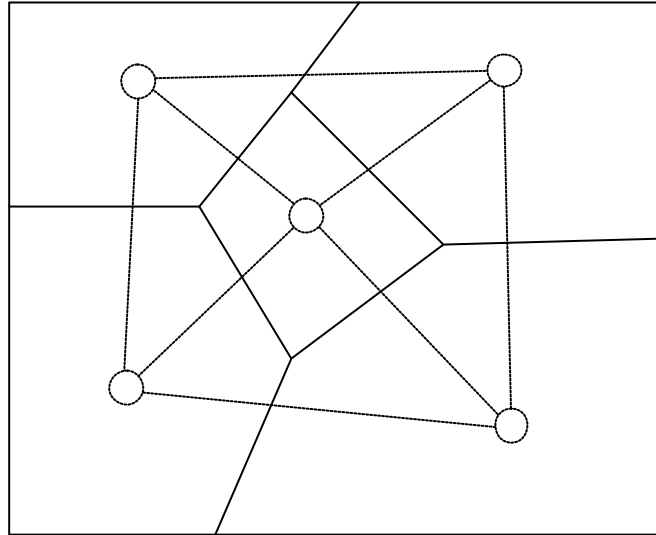


Figure 30: A number of polygons and a neighbourhood graph representing the polygons and their relations. The small circles are nodes representing the polygons; the dashed lines are edges representing border relations of the polygons.

8.1 Graph generation

The graph is generated in a number of steps. In the first step, the nodes are created and positioned. In the second, edges are added for each neighbouring polygon pair. Note that the created graph is planar, thus there is a linear relation between the number of edges and nodes.

8.1.1 Node generation

When generating the graph, a node is generated for each polygon. This node represents the location of the polygon and should be somewhere on the surface of the polygon. The centre of gravity cannot be used since it is only guaranteed to be on the polygon surface for convex polygons. Instead, the position of the node is calculated by finding the intersection between the polygon surface and a horizontal line that goes through the centre of the polygon's bounding box. This intersection results in a number of horizontal line segments. The centre of any of these resulting line segments can be used as the node position, see Figure 31. This method is only guaranteed to work with polygons with one exterior ring.

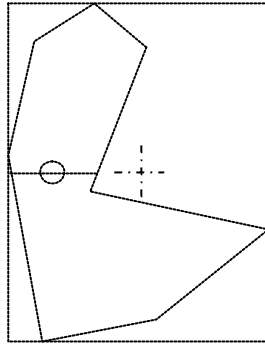


Figure 31: Node position of a polygon. The rectangle is the bounding box of the polygon and the cross its centre. The dashed line is the intersection of the polygon and the horizontal centre line of the bounding box. The circle is the final position of the node, which is at the centre of the intersection line.

8.1.2 Edge generation

When all nodes have been created, edges are added. Edges are added between all nodes whose associated polygons are neighbours. Each edge is supplemented with a *break point*, which will direct the edge over the relevant section of the border. The purpose of this is to try to reduce the number of edges crossing over other polygons than the ones whose nodes the edge connects, see Figure 32. The position of a break point of an edge for two adjacent polygons is determined by the following steps:

1. The common border of the polygons is extracted as a polyline.
2. A vertical or a horizontal line is created through the centre of the polyline's bounding box.
3. The intersection of the line and the polyline gives the position of the break point.

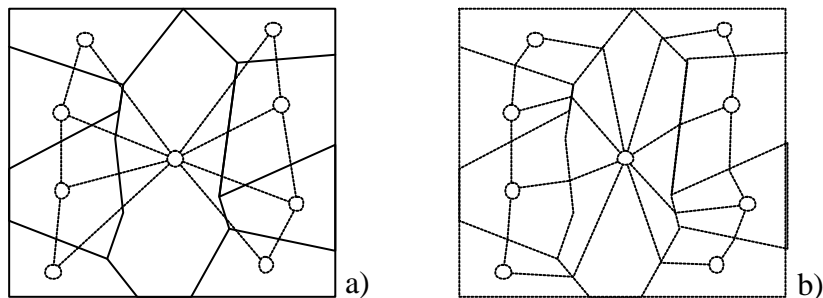


Figure 32: a) Edges without break points. b) Edges with break points.

8.1.3 Cost function

A set of attributes is associated with each polygon. These attributes have a name and a value, see Figure 33. The values are used to calculate a cost for moving in the area to which the polygon corresponds. The attributes are associated with a certain node, but the costs are associated with the edges. Let the cost of moving a meter, in an area represented by a node n , be $D(n)$.

NAME	VALUE
SHAPE	[geometry, e.g. polygon]
LANDUSE	FOREST
TYPE	DITCH
WIDTH	2
MAXSLOPE	20

Figure 33: Typical attributes for a feature object.

The actual cost of an edge depends on how much of the edge is in certain areas and the cost of moving in those areas. The break point is used to estimate how much of an edge that is in a certain area, see Figure 34. The cost, $C(e_{i,j})$, of an edge $e_{i,j}$, is obtained from:

$$C(e_{i,j}) = l_a(e_{i,j}) \cdot D(n_i) + l_b(e_{i,j}) \cdot D(n_j),$$

where l_a and l_b are the lengths from node n_i to break point $b_{i,j}$ and from break point $b_{i,j}$ to node n_j respectively. The nodes n_i and n_j are source node and destination node, respectively.

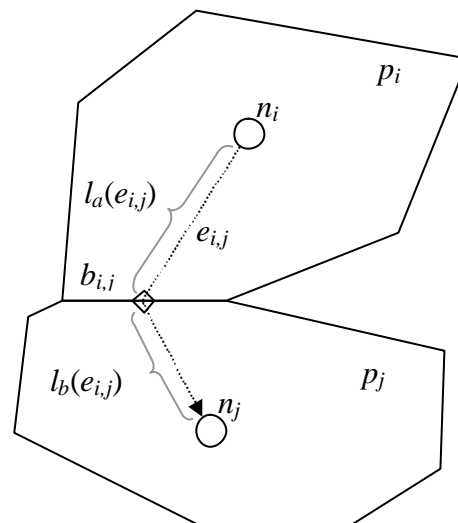


Figure 34: Components used when calculating the edge cost. The break point $b_{i,j}$ is denoted by a diamond; the nodes n_i and n_j , are denoted by circles; the edge $e_{i,j}$ is denoted by a dashed line. l_a and l_b are the lengths of the edge portions on each side of the break point.

8.1.4 Results

A theoretical analysis of the time complexity of the graph creation algorithm is difficult to make. There are many factors that influence the result, such as number of polygons, number of vertices in polygons, number of neighbours for each polygon, etc. In addition, some sub-routines are developed by a third party, thus the time complexity of these routines can sometimes only be guessed. However,

an empirical analysis can be used to measure the performance of the algorithm on selected data, see Figure 35. The tests are performed using a computer with a 440 MHz UltraSPARC-III CPU and 384 MB of main memory.

Graph creation

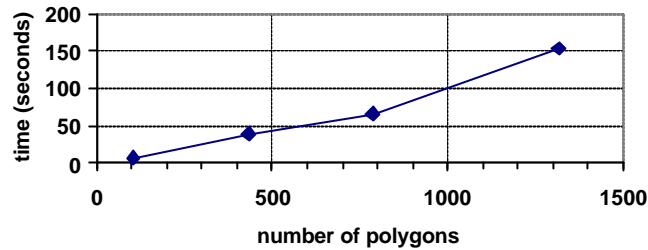


Figure 35: Performance test of the graph creation algorithm.

A partitioned 800×800 m area, with about 9200 polygons, results in a graph with about 9200 nodes and about 46000 edges, and takes approximately three minutes to generate, see Figure 36.

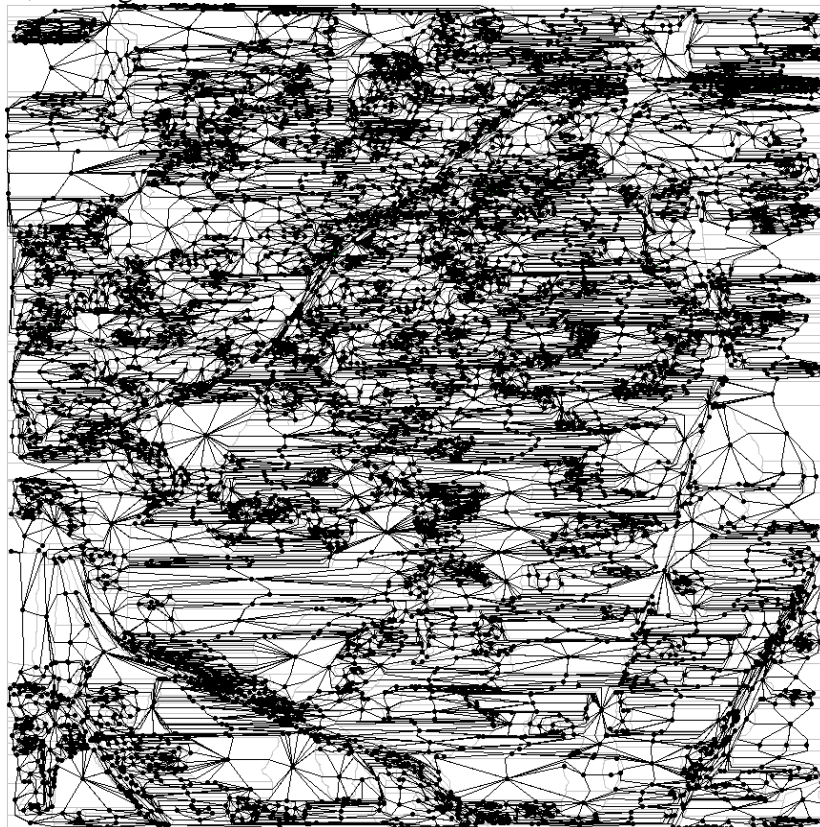


Figure 36: Graph created from a split point partitioned 800×800 m area. Tiles are outlined by grey lines; edges and nodes are denoted by black points and lines, respectively.

8.2 Graph search

A neighbourhood graph can be used to determine paths between delimited areas. It is of interest to find the best (least cost) path (or paths) between two given areas; a source area and a destination area. This can be done by using graph search algorithms. Dijkstra's algorithm can be used to find the least cost path from one node to all other nodes, or with modification, the other way around, i.e. least cost paths from all nodes to a single node. A specialized version of the A* can be used to find multiple least cost paths between a pair of nodes in successive searches, see [17]. The paths are found in increasing order of cost.

The algorithm stores an internal state between searches. If the number of required paths K is known beforehand, the algorithm can be optimized for this, by only generating enough state data to find K paths, i.e. only expand K paths. If K is large or the number of required paths is unspecified, many sub-paths have to be expanded and stored during the execution of A*. Thus it is imperative that the heuristic function gives a very accurate approximation, hence the algorithm expands fewer sub-paths, for A* to work in practice. In this work it is desirable to find several good paths where K is not known in advance, or may be very large. Thus it is important to find a good heuristic function for A*. For graphs containing nodes with geometrical positions, the actual node distances can sometimes be used for the heuristic function. This may be feasible if the cost of edges is closely related to this distance, e.g. for a graph representing a network of roads. This is not the case for this work; here the type of terrain usually is the dominant factor and not the distance itself. The geometrical distance can consequently not be used for the heuristic function in this case. Fortunately, Dijkstra's algorithm can be used to create exact values for the heuristic function. Considering an undirected graph, Dijkstra's algorithm can be run in advance to generate the path cost to reach all nodes from a single node, D . This cost is the same for reaching D from all other nodes. A* can use these costs for the heuristic function where D is the destination node. However, if a directed graph is considered, a path from A to B may not have the same cost as a path from B to A . There may not even be such a path. In other words, the commutativity of path costs for undirected graphs cannot be used. Instead, the cost problem is resolved by having either Dijkstra's algorithm or the A* algorithm "reversed". Reversing Dijkstra's algorithm is done by altering the algorithm so that it calculates the costs to reach a single node from all other nodes. These costs are used for the heuristic function of A*. Reversing A* is done by altering the algorithm so that it uses the heuristic function to approximate the costs to reach nodes from the source node, see [17]. These costs are what the normal Dijkstra's algorithm produces.

8.2.1 Results

The initial step of the search algorithm, to generate heuristic values using Dijkstra's algorithm, only takes a few seconds for a graph with about 9000 nodes. The subsequent steps of generating paths with successive executions of the A* algorithm takes about a second per path. The tests are performed using a computer with a 440 MHz UltraSPARC-IIi CPU and 384 MB of main memory. The results from a search can be seen in Figure 37. The paths show different routes which try to avoid the ditches.

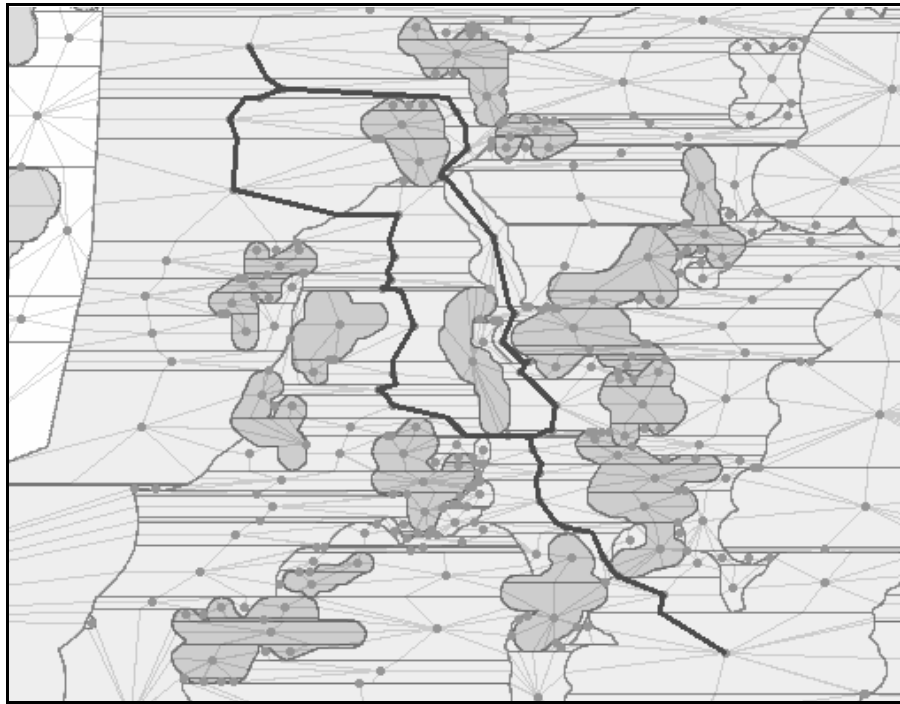


Figure 37: An example of a graph search. The black lines show multiple paths from a source node to a destination node. The darker areas are ditches.

9 Discussion

This chapter weighs the pros and cons of our achieved results. We also discuss how the results can be used and give suggestions for future research.

9.1 Conclusions

In this work, it has been shown that it is possible to model arbitrary terrain using a graph in which paths can be searched for. The advantage of basing the model on arbitrary terrain instead of on an overlay of obstacles like in [9] and [13] is that costs in the graph can be changed on a per terrain feature basis to encompass different types of vehicles.

The pre-processing of map data is rather time-consuming, but the idea is to build a repository of data so that pre-processing has to be done only once. The graph search, which is our application of the graph, can be considered fast. A search based on map data from an 800×800m area, with about 9000 nodes, typically takes a few seconds to execute.

9.2 Future research

Because of a limited schedule, we did not have the opportunity to try all of our ideas. We have tried to document some of the ideas that we think could be of interest for future research.

9.2.1 Polygon generation

We generate polygons by buffering centre lines. The centre lines have been generated from identified objects that are represented by 2×2m squares. This is not a robust solution since the generated buffers can consist of invalid polygons. Invalid polygons can cause problems in other polygon-related operations performed later in our process.

Instead of buffering centre lines there is another possibly less problematic solution. The identified objects already have a polygon representation, grouped squares, from which is possible to start. By buffering the grouped squares with a very short radius instead of buffering their calculated centre lines, a large (still square-like) outline-polygon will be generated. These outline-polygons will not require as much computational power or memory but will have an unsuitable shape for our partitioning algorithm. By applying a generalization algorithm that smoothes the outline to an appropriate detail-level, polygon data that better meet the requirements could be obtained.

9.2.2 Partitioning and graph-forming

In this work only a single partitioning strategy was implemented. It would be interesting to test other ones as well. We think that the desirable requirements could be better fulfilled using other, more complex strategies.

The partitioning strategy used in this work is simple and effective, but it has its problems. The tiles are not necessarily compact, i.e. they can have the form of thin horizontal strips. A preceding or succeeding vertical partitioning to limit the length of the horizontal strips may resolve this. Also, some polygon splits, which are bound to cause trouble, are difficult to avoid, e.g. very thin polygons, and

polygons with very small interior angles. However, this can be difficult to avoid in other partitioning methods as well.

Another partitioning strategy can be, for instance, to place cut lines between the two closest points of two different polygon rings, see Figure 38. This could offer a method to estimate how wide a passage is. This estimate could be used for evaluating the trafficability given a vehicle's width. Additional cut lines could be used to make the polygons close to convex. This strategy is inspired from *delatunay triangulation*, see [12].

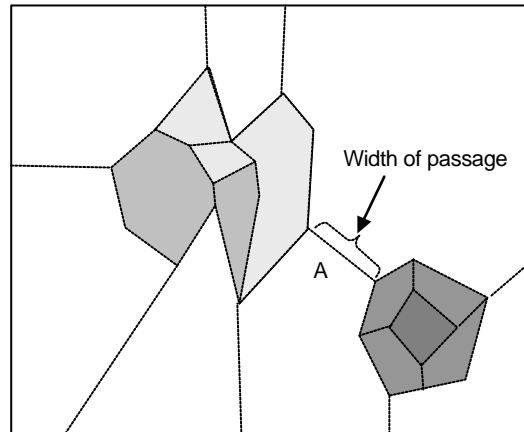


Figure 38: An example of how an alternative partitioning strategy may look. The cut line A signifies the shortest distance between the two interior rings of the white polygon. If the vehicle should pass between the darker polygons, over cut line A, then this cut line's length could be used as measurement for the width of the passage.

When forming the graph, i.e. positioning the nodes and edges, no trafficability aspects are considered. This is because the graph form should be general and vehicle independent. However, there might be some types of objects that could be a hinder for all intended vehicles. Given this, the graph could use this additional information to form the graph. The same information applies when partitioning data; areas could be partitioned differently depending on the actual type of terrain. The ability to save graphs is of interest. To do this, a format must be defined. Since the graph is highly dependent on the attributes of Features which can be stored in shapefiles, the graph could be saved alongside all other files, which the shapefile format consists of. This would also make the actual polygons available that the nodes represent as well as other functionality that the shapefile format offers.

9.2.3 Robustness

There are many geometry algorithms used in the system, some are developed in this work, while some are third party library functions. A number of the library functions have problems with robustness. Not to imply that developers of third party libraries claim to have created robust algorithms; sometimes the developers have chosen speed before robustness on purpose. The lack of robustness of some of the algorithms used in the system is directly inherited from the third party functions. In these cases, the only viable solution seems to be to find more robust third party alternatives, or actually implement these functions.

Most of the non-robustness issues arise because of malformed or dubious data.

The unsupervised polygon and vector operations of the system are difficult if not

impossible to get robust, e.g. sliver polygons arise and intersections are missed. A tool for correcting such trouble-polygons must be used between the different steps of the system. An alternative solution may be to transform the vector data into some discrete representation during the actual polygon and vector operations. Using raster data altogether, or in conjunction with vector data, could also be considered.

Data from third party companies often come with certain guarantees, e.g. polygons are valid, the distances between rings in polygons are always larger than a certain value and the interior angles of polygons are always larger than a certain value. These data are probably partly digitized by hand, analyzed and modified by some advanced tool. We generate a lot of data automatically in a cascade fashion, e.g. centre lines, buffers, merged data and partitioned data. It is difficult to make any of the earlier mentioned guarantees on this data between the various steps. If there is an error in an early step it usually propagates to the later ones, and at each step the error increases.

9.4 Closing words

The results of this work can serve as a basis for further research. The search result depends on what costs are taken into account when executing the search. The cost can for instance represent the risk of moving through different types of terrain. The area of trafficability is interesting from different points of view. Mostly, we have encountered military applications, but there are important civil ones such as rescue operations where arbitrary terrain must be forced. This work will, hopefully, contribute to solving such problems in future works.

References

- [1] E. Jungert, C. Grönwall (eds.), *From Sensors to Decision – Towards improved situation awareness in a network centric defence*. Technical Report FOI-R--1041--SE, Command and Control Systems, FOI, 2003.
- [2] S. Edlund, *Driveability analysis – using a digital terrain model and map data*, Technical Report FOI-R--1241--SE, Command and Control Systems, FOI, 2004.
- [3] M. Sjövall, *Object and Feature Recognition in a Digital Terrain Model*, Technical Report FOI-R--0499--SE, Command and Control Systems, FOI, 2002.
- [4] R. Laurini, D. Thompson, *Fundamentals of Spatial Information Systems*, Academic Press, London, 1996.
- [5] ESRI, *MapObjects – Java. Developer’s Guide*, 2003.
- [6] ESRI, *ESRI Shapefile Technical Description*, An ESRI White Paper, <http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf>, July 1998.
- [7] P. Holmes, E. Jungert, *Symbolic and Geometric Connectivity Graph Methods for Route Planning in Digitized Maps*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 14, May 1992, No. 5, p. 549-565.
- [8] J. J. Donlon, K. D. Forbus, *Using a Geographical Information System for Qualitative Spatial Reasoning about Trafficability*, From *Proceedings of QR99*, volume 4364, Loch Awe, Scotland, June 1999.
- [9] R. Ginton, C. Grindle, J. Giampapa, M. Lewis, S. Owens, K. Sycara, *Terrain-Based Information Fusion and Inference*, From *Proceedings of the 7th International Conference on Information Fusion*, Stockholm, Sweden, June 2004.
- [10] ESRI, <http://www.esri.com>.
- [11] D. Eppstein, *Voronoi diagrams*, <http://www.ics.uci.edu/~eppstein/gina/voronoi.html>, Information and Computer Sciences Online, visited Jan 14, 2005.
- [12] P Chew, *Voronoi Diagram / Delaunay Triangulation*. <http://www.cs.cornell.edu/Info/People/chew/Delaunay.html>, Cornell University, March 1997, visited Jan 14, 2005.
- [13] R. Ginton, J. Giampapa, S. Owens, K Sycara, *Integrating Context for Information Fusion: Automating Intelligence Preparation of the Battlefield*, From *Proceedings of the 5th Conference on Human Performance, Situation Awareness, and Automation Technology*, Daytona Beach, FL, March, 2004.
- [14] Lantmäteriet, *Produktbeskrivning: GSD-Fastighetskartan i Shape och MapInfo-format*, <http://www.lm.se/gsd/fastighetskartan/fastshmi.pdf>, August 2004, visited Jan 15, 2005.
- [15] Vivid Solutions Inc, *JUMP Workbench*, <http://www.vividsolutions.com>, January, 2004.
- [16] H. R. Lewis & L Denenberg, *Data Structures & Their Algorithms*, Addison-Wesley, 1991
- [17] V.M.Jiménez, A. Marzal, J. Monné, *A Comparison of Two Exact Algorithms for Finding the N-Best Sentence Hypotheses in Continuous Speech Recognition*, From *Proceedings of the 4th European Conference on Speech*

Communication and Technology, EUROSPEECH-95, pp. 1071-1074,
Madrid, 1995

- [18] Sun Microsystem, *Java 2 Plattform, Standard Edition, v 1.4.2 API Specification*, <http://java.sun.com/j2se/1.4.2/docs/api/>, November 2003, visited Jan 26, 2005.

Thesis specification (Swedish)

Examensarbete – Beslutsstöd för Framkomlighetsanalys 2

Bakgrund:

Analys av framkomlighet i terräng ger ett viktigt beslutsunderlag för alla typer av aktiviteter som kräver förflyttning i terrängen. Denna typ av analys behövs både för att kunna bedöma andras möjligheter till förflyttning (målföljning), och för planering av egna förflyttningar. Vid FOI i Linköping finns sedan tidigare en metod framtagen för att bestämma framkomlighet i olika delar av terrängen, för att skapa en framkomlighetskarta. Sedan tidigare finns också en algoritm för att hitta den bästa vägen mellan olika noder i ett nätverk. För att kunna utnyttja dessa tillsammans måste dessa metoder utvecklas, samt ett nätverk skapas utifrån framkomlighetskartan.

Problemställning:

Uppgiften består i att skriva ett program för att bestämma de bästa vägarna mellan två givna områden. För att göra detta måste man skapa ett nätverk utifrån en framkomlighetskarta. Framkomlighetsegenskaperna är beräknade utifrån en högupplöst höjdmodell, information om terrängtyp från Gröna kartan (skog, väg etc) och information om fordonens egenskaper. Ett mindre jobb för att utnyttja den tidigare utvecklade programvaran måste också göras. Den färdiga analysen ska visualiseras på en karta där de bästa vägarna finns utmärkta.

Kontaktpersoner:

Erland Jungert, jungert@foi.se, 013-37 83 37

Fredrik Lantz, flantz@foi.se, 013-37 82 36

Institutionen för Data- och informationsfusion

Avdelningen för Ledningssystem

FOI, Linköping

Utbildning:

C- eller D-linje.

Tidsperiod:

Start sker tidigast 20:e september eller senare under hösten -04.

MapDataManager

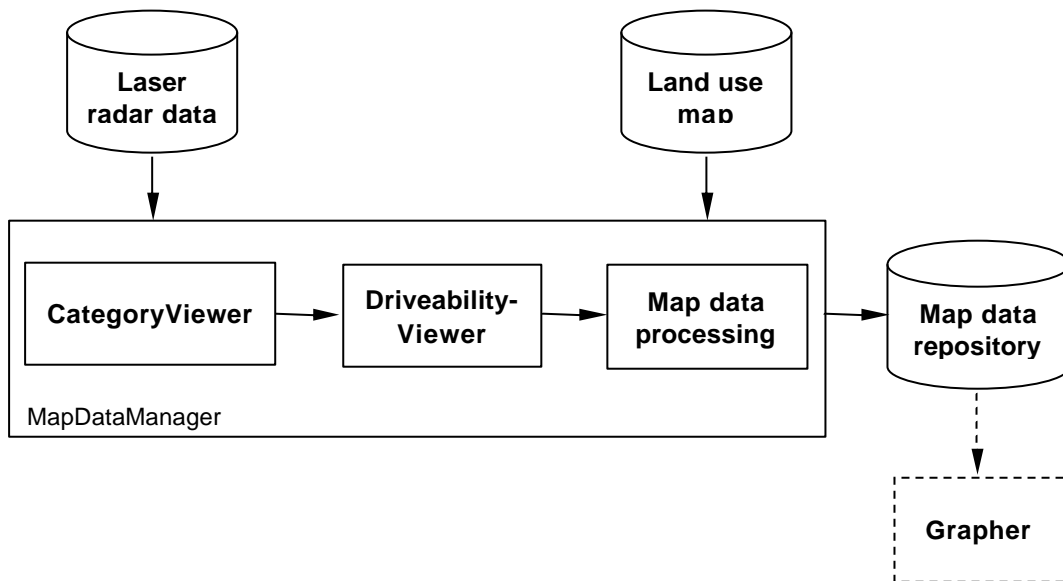
This is a brief description of a tool developed to prepare data for partitioning and graph creation in another tool, *Grapher*, see Appendix C. Offered functionality corresponds to the initial steps described in chapters 5 and 0.

Architecture

This application, *MapDataManager* (MDM), is written in Java (version 1.4.2) [18] and uses MapObjects [5], a toolkit from ESRI [10]. MDM is an extension of DV [2], introduced in chapter 5. DV is an extension of CV [3], also introduced in chapter 5.

CV is an application that, from a special kind of map data [3], can identify geometrical features (objects) and store them in a data structure. DV extracts these objects from CV and calculates some properties like average width of objects and store these properties in a data structure.

MDM is designed to extract the identified objects together with corresponding properties from DV and then prepare this data for further processing in *Grapher*. The result from MDM can be saved as *shapefiles* [6], which will serve as a map data repository for *Grapher*.



MapDataManager is composed of **CategoryViewer** and **DriveabilityViewer**. Arrows show the information-flow through the system.

Portability

Software written in Java is generally portable to different platforms. MDM is developed and tested on a UNIX platform. MDM has also passed some tests on Windows XP, but with some modifications:

MDM is composed of CV and DV, which have certain paths to its input data hard-coded. To run MGM on another platform or on the same platform but from another machine or directory, these paths must be updated and MGM recompiled. The paths for which change is needed are found in a source file of

DriveabilityViewer called *Properties.java*. This is obviously a limitation, but acceptable since this can be considered as a series of prototypes.

Installation

The MDM installation consists of the following directory-tree, which must not be altered for proper execution:

```
MapDataManager/  
  Build/  
    CategoryViewer      compiled classes for CategoryViewer  
    DriveabilityViewer  compiled classes for DriveabilityViewer  
    mapDataManager      compiled classes for mapDataManager  
  data/  
    inData/  
      fastighetskartan  map data from Lantmäteriet  
      laserradar        data from laser radar  
    outData/  
      200x200m          out data repository (size 200x200m)  
      400x400m          out data repository (size 400x400m)  
      600x600m          out data repository (size 600x600m)  
      800x800m          out data repository (size 800x800m)  
  docs/                 generated Javadoc for the application  
  filters               filters used by CategoryViewer  
  lib                   binary libraries from MapObjects  
  src                   sources for the application
```

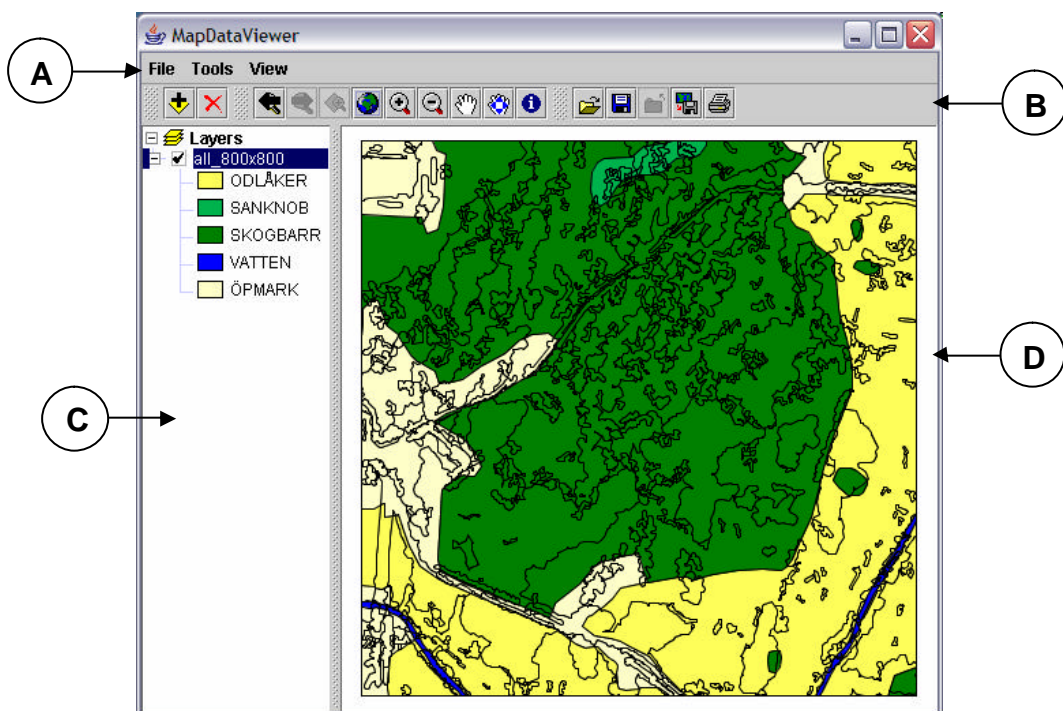
The installation includes a pre-generated out-data-repository with data in four different sizes (200x200m - 400x400m).

Besides the source code, Java class-files, input data-files etc. a Java virtual machine must be installed. Versions compatible with version 1.4.2 can be downloaded free from [18].

User interface

The user interface is divided into the following different areas, see screenshot below:

- A. **Menu bar** with functions for data import, data manipulation and visualization.
- B. **Toolbar**, buttons with functions for zooming-, panning-, searching- and saving screen shots.
- C. **Layer view** with a hierarchical tree view of all map layers that are open and available in the system.
- D. **Map view**, where map layers that have been selected in the *layer view* are shown.



MapDataManager's user interface is divided into four areas.

Menus

Most of the functions are carried out from the menus. There are three menus available: *File*, *Tools* and *View* with the following contents:

File

- | | |
|---------------------------|---|
| Save selected layer | // This menu choice saves the selected layer from the layer view to a shapefile. |
| Import to temp repository | // This menu choice imports ditches and slopes from DrivabilityViewer and stores them in a temporary repository (RAM, not to disc). |

Import and generate merged data	// This menu choice imports ditches and slopes from DriveabilityViewer and merges them with a land use map. The result is three new layers: one with ditches, one with slopes, and one with the merged data.
Temporary repository	// This menu choice becomes available after choosing the <i>Import to temp repository</i> menu command. It holds ditches and slopes in a grouped square representation. It also holds calculated centre lines for both ditches and slopes. From here the desired objects can be extracted for further processing.
Open (GSD) Land use map	// This menu choice opens a land use map (GSD fastighetskartan) from the installation directory.
Open (GSD) Road map	// This menu choice opens a road map (GSD fastighetskartan) from the installation directory.
Tools	
Reduce layer extent	// This menu choice reduces the extent (map size) of one of two selected layers. The purpose is to reduce the extent of layers originating from <i>GSD fastighetskartan</i> to the same extent as imported data from DriveabilityViewer.
Remove line loops	// This menu choice removes any existing line loops from a selected (line) layer. Line loops can cause the <i>Generate buffers</i> function to not terminate or may generate an error message.
Generate buffers	// This menu choice generates outer shell polygons (buffers) on the geometries of a selected (line) layer. The purpose is to create polygon objects from DriveabilityViewer's centre lines.
Remove invalid geometries	// This menu choice removes any existing invalid geometry from a selected layer. The result is shown in a new layer.
Merge	
Ditches and Slopes	// This menu choice merges two selected layers, one with slopes and one with ditches, to a new layer.
DV-data with Land use map	// This menu choice merges two selected layers, one with DV-data (Ditches or

	slopes from DriveabilityViewer) and one with a land use map, to a new layer.
DV-data with Roads	// This menu choice merges two selected layers, one with DV-data (Ditches or slopes from DriveabilityViewer) and one with Roads (created from the <i>Experiments</i> submenu), to a new layer.
Remove overlap	
Ditches or Slopes	// This menu choice removes overlaps from a selected layer. The result is shown in a new layer.
Test data	// This menu choice removes overlaps from a selected layer containing test data created with JUMP Workbench (different attributes than ditches and slopes). The result is shown in a new layer.
Experiments	
Generate outlines	// This menu choice generates outer shell polygons (buffers) on the geometries of a selected (polygon) layer. The purpose is to create polygon objects from DriveabilityViewer's grouped squares, only implemented for experimental purposes.
Buffer roads	// This menu generates outer shell polygons from a selected line layer (of roads). The generated polygons will be eight meters wide. This menu choice is only implemented for experimental purposes.
View	
Colorize Land use map	// This menu choice colorizes the selected layer (containing a land use map).

Normal usage

The normal use of this program can be described in a number of steps:

- Extract data (ditches and slopes) from DV.
- Extract corresponding centre lines from DV.
- Remove possibly intersecting segments (line loops) from centre lines.
- From centre lines create new polygon data (buffers).
- Remove individual overlaps for both ditches and slopes.
- Merge ditches and slopes and remove their overlaps.
- Merge the ditches and slopes with additional map data (land use map), to receive map data with full surface coverage and more attributes.
- Save the resulting map data to file (*shapefile*) for later use in the tool *Grapher*.

All steps described above, except for saving the result to file, can be done with one single selection from the menu, see screenshot below. If this is not wanted or data from other sources should be used, all steps can also be executed manually from the menus.



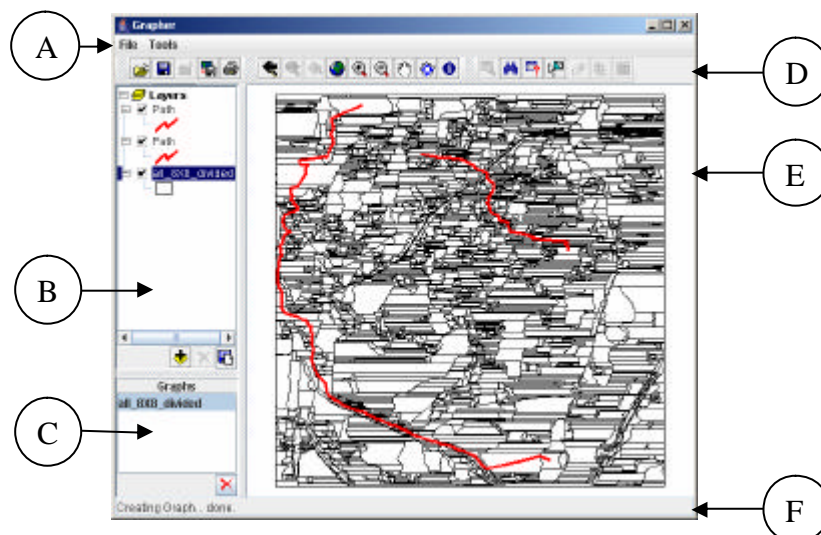
Import and prepare data from DV in one step by selecting *Import and generate merged data* from the *File* menu.

Grapher

This program has the ability to partition polygon data and build/search graphs, using the strategies explained earlier in this report.

User interface



- A. Menu bar – Contains menus for partitioning and graph functionality.
- B. Layer view – A tree and toolbar for managing layers, e.g selecting, opening, closing and saving layers.
- C. Graph view – A list and toolbar for managing graphs, e.g. selecting and closing graphs.
- D. Toolbar – Standard MapObjects toolbars, containing functionality for zooming, panning, searching etc.
- E. Map view - View of the layers accounted for in the layer view.
- F. Status bar – Shows the status of some time consuming processes, e.g. partitioning and graph creation.



Screenshot of Grapher.


Partitioning

This is a step by step instruction on how to partition data using Grapher.

1. Add a layer containing polygon data. This is done by clicking the *add layer* button  in the layer view, and then selecting a data source containing polygon data e.g. `all_800x800.shp` from MDM's data repository.
2. Select the layer to partition in the layer view by clicking on its name.
3. Start the partitioning process by selecting the *Tools* menu on the menu bar, then selecting *Partitioning*, then clicking on *Partition*.
4. Wait a couple of minutes... The progress of the process can be followed from the status bar.
5. A new layer should show up in the layer view and the map view. This layer can be saved for later use by clicking the *save layer* button  in the layer view.

Graph creation

This is a step by step instruction on how to create and visualize a graph using Grapher.

1. Add a layer containing polygon data, preferably partitioned data. This is done by clicking the *add layer* button  in the layer view, and then selecting a data source containing polygon data.
2. Select the layer, from which the graph should be created, in the layer view by clicking on its name. This can be a layer from step 1 or a layer created by following the *partitioning* instructions.
3. Start the graph creation process by selecting the *Tools* menu on the menu bar, then selecting *Graph*, then clicking on *Create Neighbourhood Graph*.
4. Wait a couple of minutes... The progress of the process can be followed from the status bar.
5. The graph is now created and should show up in the graph view.
6. The previous step merely created the graph, now it could be visualized if one so desires. Select the graph by clicking its name in the graph view. Then under *Tools* on the menu bar, select *Graph*, and then click *Visualize Graph*.
7. Wait a couple of minutes... The progress of the process can be followed from the status bar.
8. Two new layers, representing the graph, should show up in the layer view and map view, one layer for nodes and one for edges.

Graph search

This is a step by step instruction on how to perform a search on a graph using Grapher.

1. Create a graph using the steps in the *Graph creation* instructions.
2. Select the graph by clicking on its name in the graph view.
3. Select the layer, in the layer view, from which the graph was created.
4. Open the search dialog by selecting *Tools* on the menu bar, then selecting *Graph* and finally clicking *Search Graph*.
5. Click on two polygons that should be used for the search. This will fill in the source and destination fields of the search dialog with the id of the clicked polygons.
6. Click the *search* button repeatedly to generate successive paths.