universität wien

DISSERTATION

Titel der Dissertation

„Algorithms for surface water networks and their catchments - with special emphasis on the EU Water Framework Directive“

Verfasserin

Mag. Doris Riedl

angestrebter akademischer Grad

Doktorin der Naturwissenschaften Dr. rer. nat.

Wien, im Mai 2014

Studienkennzahl lt. Studienblatt:	A 091 455
Dissertationsgebiet lt. Studienblatt:	Kartographie und Geoinformation
Betreuerin / Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Kainz

For

Andreas,

Magdalena, Katharina, and Valentina

i. Abstract

Network analysis tools belong today among the standard tools in many Geographical Information Systems. Their use has expanded into many areas. In hydrology, hydrometry, in the frame of river basin management and other closely related fields the tools are used for the processing and analysis of river and catchment networks. For evaluations such as for example those that have to be made for reporting and planning purposes in the context of the EU Water Framework Directive (WFD), there are not always sufficiently tuned algorithms created or widely implemented as I could see during my work in this field.

This thesis deals with these algorithms that are specially attuned to the problems of the WFD. After an introductory theoretical part, where basic concepts are explained and a comprehensive explanation of the components of a river network, including the digital storage options of network structures is given, there follows a a section which deals with the questions regarding digital river networks (including their catchment areas), as they are needed for the evaluations for the EU Water framework Directive. Based on the conclusions drawn in this section existing GIS tools and algorithms where sought, and it was tried to create still missing algorithms.

In a further step algorithms for network processing, the calculation of river systems and data aggregation for the used GIS software ArcMap [ARCMAP 2012] using the freely available IDE Eclipse [Eclipse 2012] in the form of Java [JAVA SE 6] Add-ins were implemented exemplaryly. They should facilitate future work for data preparation and analysis in the context of the EU Water framework Directive. Here, particular attention was given to achieve runtimes of linear orders, to perform reasonably well with the large amounts of data, as they exist in large river networks. The depth-first search (DSF) proved to be a particularly suitable approach for the calculations. In a concrete example it can even be shown that a new algorithm, based on this DSF approach, improves significantly the runtime when compared to an existing tool's algorithm.

ii. Zusammenfassung

Netzwerkanalysetools gehören heute zum Standardwerkzeug vieler Geografischer Informationssysteme. Ihr Einsatz erstreckt sich über viele Bereiche. Auch in der Hydrologie, Hydrometrie und im Rahmen der Einzugsgebietebewirtschaftung sowie weiteren verwandten Bereichen bietet sich ihr Einsatz zur Bearbeitung und Analyse von Fluss- und Einzugsgebietsnetzwerken an. Für Auswertungen, wie sie z.B. im Umfeld der EU-Wasserrahmenrichtlinie benötigt werden, sind allerdings noch nicht ausreichend abgestimmte Algorithmen geschaffen oder implementiert worden.

Die vorliegende Dissertation beschäftigt sich mit diesen, auf Fragestellungen der Wasserrahmenrichtlinie abgestimmten Algorithmen. Nach einem einleitenden theoretischen Teil, in dem grundlegende Begriffe erklärt und eine umfassende Erläuterung der Bestandteile eines Fließgewässernetzwerkes inklusive der digitalen Speichermöglichkeiten von Netzwerkstrukturen gegeben wird, folgt ein Abschnitt der sich mit den Fragestellungen an digitale Flussnetzwerke (inkl. deren Einzugsgebiete), wie sie für die Auswertungen für die EU-Wasserrahmenrichtlinie benötigt werden, beschäftigt. Anhand der in diesem Abschnitt erstellten Schlussfolgerungen wurde gezielt nach bereits existierenden GIS-Werkzeugen und Algorithmen gesucht, bzw. versucht fehlende Algorithmen zu ergänzen.

In einem weiteren Schritt wurden exemplarisch Algorithmen zur Netzwerkaufbereitung, zur Berechnung von Flussordnungen und zur Datenaggregierung für die verwendete GIS-Software ArcMap [ARCMAP 2012] mit Hilfe der frei zur Verfügung stehenden Endwicklungsumgebung Eclipse [ECLIPSE 2012] in Form von Java [JAVA SE 6] Add-Ins implementiert, welche zukünftige Arbeiten für die Datenaufbereitung und Analyse im Umfeld der EU-Wasserrahmenrichtlinie erleichtern sollen. Dabei wurde besonders darauf geachtet, möglichst lineare Laufzeiten zu erzielen, um mit den großen Datenmengen, wie sie in großen Flussnetzen anfallen, in sinnvoller Zeit Berechnungen durchführen zu können. Als besonders geeigneter Ansatz für die Berechnungen zeigte sich dabei die Tiefensuche (DSF). Anhand eines konkreten Beispiels kann sogar gezeigt werden, dass ein neuer, auf diesem DSF-Ansatz basierender Algorithmus signifikant verbesserte Laufzeiten gegenüber einem in ein bestehendes Werkzeug integrierten Algorithmus erzielt.

iii. Contents

iv. Figures

v. Tables

vi. Preface

Some 10 years of my life I spent working for the Austrian Federal Environment Agency and was mostly concerned with pollution data of groundwater and surface waters. My main work was that of a "typical" geographer: I sought to connect data with space. My tool of choice was a Geographic Information System. In the early days of my work this was a relatively unwieldy thing. It was slow, when using a big load of data and many of the calculations I needed, I first had to translate into scripting languages. Those scripting part became larger and larger and turned into small programs that made our GIS life at the water department a bit easier. Since then, time has gone by and technology has developed a lot. When I got the chance to come back to university to write a thesis, I decided to base it on my knowledge of water data and how they are connected to space, furthermore how they are usable in the best possible way when applying GIS.

The writing of the thesis would not have been possible if I had had no help:
My husband Andreas ensured that I had the peace to do the work for my thesis. He spent a lot of weekends and afternoons away with our three daughters to give me enough time to write. He also urged me to go further if necessary, provided discussion and was helpful in many other ways. My daughters Magdalena, Katharina, and Valentina knew when it was time to be quiet.
At the department of Geography and Regional Research my colleagues were a good audience when musing over one of the several problems that arose. I want to name some of them: Wolfgang Kainz was really patient to wait for the results and helped where possible. Roland Mittermaier kept faith with me and always tried to cheer me up, and Andreas Chlaupek who shared an office with me for some years readily provided me with mathematical information and (never complaining) listened to a lot of lamentation.
In the Austrian Federal Environment Agency my former colleagues helped by testing my outcomes and giving me valuable information for the use of networks. Especially I want to thank Günter Eisenkölb and Gabriele Vincze.
During programing I received useful feedback from Thomas Jandl, who listened and commented patiently as I mentioned some coding problems.
Thank you all!

0 Introduction

Water is an important component of life. The more the stress on water grows, the more it is necessary to come to its aid. Hydrologists, environmentalists, geomorphologists, people in water basin administration, people in flood protection, and many others work partly in the field of water protection and need simple tools to prepare and analyse data according to their purposes. GIS already offer a bandwidth of tools that can be used to serve those purposes. During my work at the Austrian Federal Environment Agency I used GIS to provide my colleagues with spatial data in the field of groundwater and surface water, to combine different themes and to calculate new or extract existing information. On European level it has also been recognized that GIS is an important tool when it comes to analyse and model data. When the WFD (the European Water Framework Directive) [WFD 2000] - a framework to ensure good water quality - was drafted, GIS was included in Annex I. After it finally was adopted at the end of the year 2000 (12 December 2000), working groups were formed to discuss the various aspects and find common solutions. One of them was the Working Group 3.1 that mainly dealt with GIS questions and met regularly in the European Joint Research Centre in Ispra, Italy. At a more detailed level, there were special GIS working groups installed for at least every big water basin - concerning Austria it was for example the Danube GIS-group that works in the frame of the International Commission for the Protection of the Danube River, or the Rhine GIS-group whose members me(e)t mostly in Koblenz, or the Elbe-group - that had to find their own specific solutions to the questions of the WFD. Those working groups, whose meetings I could attend several times, deal(t) mostly with the preparation of common data concepts, data integration (both on semantic level in form of data models and domain values, as well as on spatial level when trying to fit together the many different georeferencing systems, scales and degrees of generalization) and data preparation. Many steps had to be taken to reach common data sets that could be used for EU-wide evaluation of the data, which form some of the basics and outcomes of the national River Basin Management Plans. Still, some underlying concepts (e.g. on scale and generalization) will have to be followed up in more detail in the coming stages of data preparation to enhance data quality and comparability.

A wide choice of WFD data is closely related to the surface waters. Those surface waters are split into individual water bodies for the purposes of the WFD, but in reality form a vast network structure. The network's water bodies' data can only be handled correctly with simultaneous consideration of the data from linked water bodies. The processing of the surface water networks is therefore an essential part of the work performed for the WFD. The GIS analysts can rely on tools for network analysis, which serve already many purposes. Tools for the creation of networks exist as well as tools for tracing them. For more special questions (e.g. special subject-specific aggregations in a network) new tools were created and still have to be created. This is where this thesis has its starting point. Even when working with a widely used and extensive GIS-system the need for more specialised network analysis tools arose during my work on WFD-data. Tools provided as extra extensions were used (e.g. ArcHydroTools) and own tools were implemented at first with Visual Basic and later with Java to make certain data preparation and analysis steps possible. The tools serve their purpose (in the sense that they produce the right values), but most of them have a real drawback, they are slow. The analysis of some underlying algorithms (or in case of not using an open source product, the testing of the tool with data and watching of its progress) showed that their runtime was mostly long (quadratic order or even of a higher one). This results in disproportionally long processing times.

With respect to these introductory thoughts and findings the main focus of the thesis will be on what data preparation and analysis is required to work with surface water networks and additional water data that can be combined with these networks and how can resulting tools be made quicker than comparable existing tools (created by are at the moment. Thus the main scientific question of the thesis is:

- Which algorithms for the analysis of hydrologic networks are useful and necessary and how can they be implemented in an optimized way?

As algorithms are usually combined with data input and output a side question will be:

- Is there a need for special, that is algorithm based, data structures, which have to be developed along with the algorithms?

Therefore, the following objectives can be derived:

- The finding or creation of algorithms for networks that have a processing time of a less than quadratic order.
- The implementation of these algorithms in a GIS (incl. description and testing).

To limit the large field of network preparation and analysis around the WFD to something manageable I limited the data to vector data. To start with what you will not find in this thesis (before you will read what is in it) - there will be no handling of the question on how to derive the network data from a raster model, nothing about flat areas, depressions, burning in rivers[1] and so forth.

A further restriction was made in mainly considering surface water. Groundwater will only appear marginally. And as it is a very wide and special field there will also not be any discharge or runoff modelling

This thesis will concentrate on vector data in hydrology and hydrometry, as they are readily available. As an introduction to the matter in chapter 1 and 2, the thesis will cover an overview on surface water networks and their catchment areas, including their representation. Although Goodchild states that *the poor definition of resolution for vector data is a strong argument for the use of raster data in rigorous scientific research* [GOODCHILD 2011], raster data is not a first choice with networks. The representation of surface water networks as graphs is with vector data most intuitively, as will be shown in chapter 2. From vector data to networks there is only one step that includes attributions to define the relation between the lines. Vector data networks are easy to trace and they are easy to combine with additional layers. With vector data it is easy to incorporate the load of attributes for the WFD or the measures, to get a link to the kilometre-wise GIS data available for the large rivers.

In chapter 3 it will be shown how surface water network data (incl. catchments) is used. This is achieved by extracting information from scientific articles and books and by talking to and working with people of different sciences and administrative bodies.

[1] To fit the DEM with available vector stream data and to improve the continuity of a derived river network, cells of the DEM are lowered (burned in) where an overlaying vector stream segment intersects them.

[2] Yes, I know there are some regions in our world where there are more lakes than rivers. C.f. [BRITTON

Chapter 4 forms the core of the thesis. This part of the thesis will give an introduction to algorithms based on a literature review with emphasis on the orders of runtime and show the description and Java implementation for algorithms, both found in literature as well as newly created ones, that are all based on the concluding list from chapter 3.

Chapter 5 is the practical part of the thesis where the implementations are tested against real data and some outputs are shown.

In chapter 6 I give a résumé and outlook for future work.
The implementation of the algorithms produces a big amount of code. To make the thesis a bit more readable, only the central methods of the code are printed and commented on in the main text. The complete Java code can be found on the CD-ROM attached to the thesis.

Literature is used in this work. Therefore, literal citations are distinguished from my own text as indented paragraphs, or if only a small piece of text it is printed in italic letters. Citations were copied out of books and journals as is, any obvious typing errors are marked with [!]. In the citations, the commonly used scientific journals cross references to other literature, like [1, 6] were omitted, because there are no analogies in the current text and it damages the readability of the text.

1 The hydrologic network - nature vs. model

1.1 Hydrologic networks

In hydrological research, the term "network" can be used for both the connection of the drainage system in nature as well as in its digital representation. To avoid confusion I use the term ***surface water network*** as the synonym for connected surface waters in the real world and ***digital water network*** and ***network model*** for its counterpart in the digital and graphical environment.

The first step to find the appropriate terms to use for the elements of a surface water network leads to the Water Framework Directive (WFD) [WFD 2000, 327/6] itself. In Article 2 the WFD contains many definitions relevant to hydrology, monitoring, administration etc. Those found relevant to surface water networks can be found in section 1.1.1.

The WFD definitions serve as a starting point for more detailed definitions of the elements of surface waters. Section 1.1.2 uses the WFD definitions and tries to make a holistic picture of a surface water network, starting with a simple composition of rivers and fitting the other elements into the structure one after the other. Thereby trying to assess the rise in complexity, each additional element contributes to the network.

Section 1.2 and its subsections are dedicated to the explanation of how the surface water network and all its elements can be represented as a model (both analogue as well as digital). For this reason a short introduction to graph theory and networks is given and a small digression in scale dependency is made.

1.1.1 WFD definitions

The Water Framework Directive contains several definitions that are of relevance for this thesis and form the basis of the thesis' definitions:

> **'Surface water'** means inland waters, except groundwater, transitional

waters and coastal waters, except in respect of chemical status for which it shall also include territorial waters.

'**Inland water**' means all standing or flowing water on the surface of the land, and all groundwater on the landward side of the baseline from which the breadth of territorial waters is measured.

'**River**' means a body of inland water flowing for the most part on the surface of the land but which may flow underground for part of its course.

'**Lake**' means a body of standing inland surface water.

'**Transitional waters**' are bodies of surface water in the vicinity of river mouths which are partly saline in character as a result of their proximity to coastal waters but which are substantially influenced by freshwater flows.

'**Coastal water**' means surface water on the landward side of a line, every point of which is at a distance of one nautical mile on the seaward side from the nearest point of the baseline from which the breadth of territorial waters is measured, extending where appropriate up to the outer limit of transitional waters.

'**Body of surface water**' means a discrete and significant element of surface water such as a lake, a reservoir, a stream, river or canal, part of a stream, river or canal, a transitional water or a stretch of coastal water.

1.1.2 Elements of a surface water network

A **surface water network** consists of several elements of the surface waters (as defined in the WFD) that form an *uninterrupted connection*. Because the WFD is rather general and technical in its definitions, this section will have a closer look at the details of the surface water network. The International glossary of Hydrology [IGH 2013] has been used in this section and its subsection (esp. 1.1.2.5) to find the appropriate terms for the different elements of the surface water network.

Rivers form the main component of the surface water network[2]. Under the term "river" we understand a natural watercourse that flows towards the sea or as a tributary towards another larger river. There are different synonyms for rivers like stream, waterway, watercourse, burn, beck, brook, ditch, rivulet, creek, wash, run, and so on[3]. The stream

[2] Yes, I know there are some regions in our world where there are more lakes than rivers. C.f. [BRITTON 2002] esp. the section about the Finnish coding system

[3] The different types can show the approximate size of a river when used regionally, but they do not offer an exact and universal way of classification, because their meaning is different between the regions.

flow in rivers depends on precipitation (see Figure 1–1), but only a very small amount of the rain (or snow) falls directly into the river.

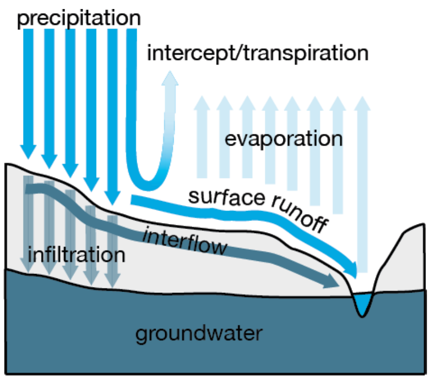


Figure 1–1 Formation of the stream flow in river channels (modified after [ZEPP 2002], p. 114)

Portion of the precipitation is intercepted by the vegetation or transpires from its leaves. The part of the water reaching the earth's surface becomes either surface runoff, infiltrates into the ground or evaporates from the surface. The fraction of the water that infiltrates into the ground becomes the subsurface water. Some of it flows more or less parallel to the surface as interflow the rest seeps deeper and becomes part of the groundwater. Portions of the subsurface water provide the baseflow of the river, which is the reason why there is water in a river during dry seasons.

Every river can be divided into **river segments**. A river segment is a river stretch that has no points where another river stretch joins. The starting point of a river segment is either a starting point of the river or a point where another river stretch joins. A **surface water network system** ideally has many starting points and one **endpoint**, towards which all the water flows (e.g. the point at the coastline where the river empties into the sea). The endpoint of a river segment is either a point where another river stretch joins or the surface water networks endpoint.

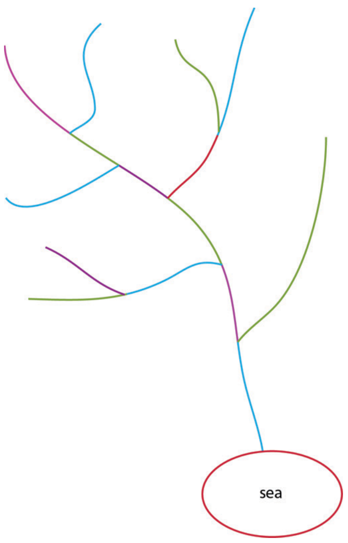


Figure 1–2 Simple hydrologic network - To distinguish the river segments they are drawn in different colours

This ideal simple form of a surface water network works fine for wide parts of the earth's surface water networks, but some exceptions/additions will be described in the following subchapters.

1.1.2.1 Divergence, bifurcation, braided river - main reaches vs. subordinated reaches

In some parts of a surface water network the flow splits into branches. From one point in the network the water can therefore flow in two different directions. This is the case with alluvial rivers (which are rivers that have their bed in sediments rather than in bedrock and can therefore easily be relocated) in flat areas (like in the headwater regions where there are areas of glacial or fluvioglacial deposits; or in areas where fine sediments are accumulated like in floodplains). If the "parallel" stretches are not of the same size, a main reach (major branch) versus a subordinated reach (minor branch) can be distinguished.

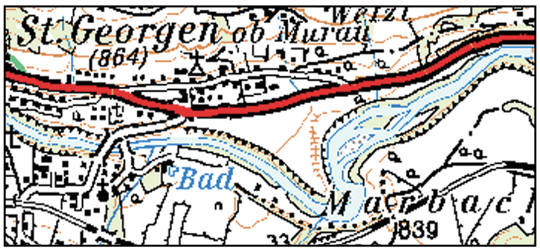


Figure 1–3 Braided river stretch (Source: Austrian Map 3D - Bundesamt für Eich- und Vermessungswesen BEV)

For the hydrologic network this means that there are parallel river segments. Most of them will enclose islands because they usually flow together again some distance downstream from the splitting point. Near the estuary mouth some of them will directly empty into the sea, meaning that the flow that comes from one river system into the sea has to be summed up for all the river mouths of one river system.

Rivers with several branches can grow into rather complex systems, especially when the branches receive other rivers before joining the main flow again.

1.1.2.2 Disappearing and underground rivers

Some rivers vanish from the earth's surface, which can happen either through natural causes (e.g. sinkholes in karst areas; large sediment areas that form large aquifers with no overlaying impermeable layer; draughts) or due to anthropogenic reasons (e.g. overbuild rivers or channel rivers underground). Those anthropogenic-modified rivers, even if they are underground, cannot be categorized as diffuse groundwater. They usually have known flow paths, which mostly surface at some point. Thus, the segment(s) underground can be more or less easily retraced. This is possible only in some cases with rivers that disappeared because of natural causes. Some underground rivers in karst regions (e.g. in the Moravian karst in the Czech Republic) formed big cave systems where the river can be used for boating (or at least canyoning[4]) and therefore can also be retraced easily.

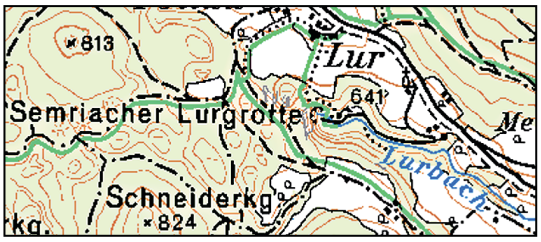


Figure 1–4 "Disappearing" river. (Source: Austrian Map 3D - Bundesamt für Eich- und Vermessungswesen BEV)

[4] In the U.S.: canyoneering

In other karst areas (e.g. the Dachstein area in Austria and in the Slovenian karst) tracing methods (either colour, spores or some other radioactive substance) have been used to find either the underground flow paths or at least the corresponding points where a disappeared river surfaces again, but in some cases it may never be entirely clear how those underground drainage systems work, because there is a variation that can highly depend on the precipitation input. For Austria, as well as some other countries, there exists a database of tracer experiments [TRACER] that can help to find the underground links between different parts of the river network.

In case of sediment bed (e.g. gravel and sand bodies in alluvial plains) a disappearing river could join the groundwater that in many cases joins the surface water flow downstream from its disappearance point in a diffuse way as baseflow.

Concerning the hydrologic network, attention has to be turned to whether the underground part forms a retraceable stretch of water, or if at least a link can be made from the disappearing point to the point where the water appears again in a fuzzy manner, or if there is no connection at all that can be established for sure, thus it is only known that the water has to surface somewhere, but it is not clear where. In the Austrian River Network for Reporting, which will be used in later chapters such links can be found.

1.1.2.3 Lakes

Lakes are special items in the surface water network. They can be of different origin, whereas the lakes formed in a glacial environment are, seen worldwide, probably as the biggest group. But lakes can also form because of tectonic processes, through the erosion or accumulation and even because of the wind. Man-made lakes are a further category (see 1.1.2.5.).
Lakes are an area element with zero to many contributing rivers and zero to many runoff points and a shoreline, which usually differs from the main current flow. Closed lakes (that have no runoff) lose their water either by evaporation or by seepage to the groundwater. They form a small surface water system of their own and are not directly part of a greater surface water network surrounding them. If there is a contribution to the network flow it will be diffuse via the baseflow.

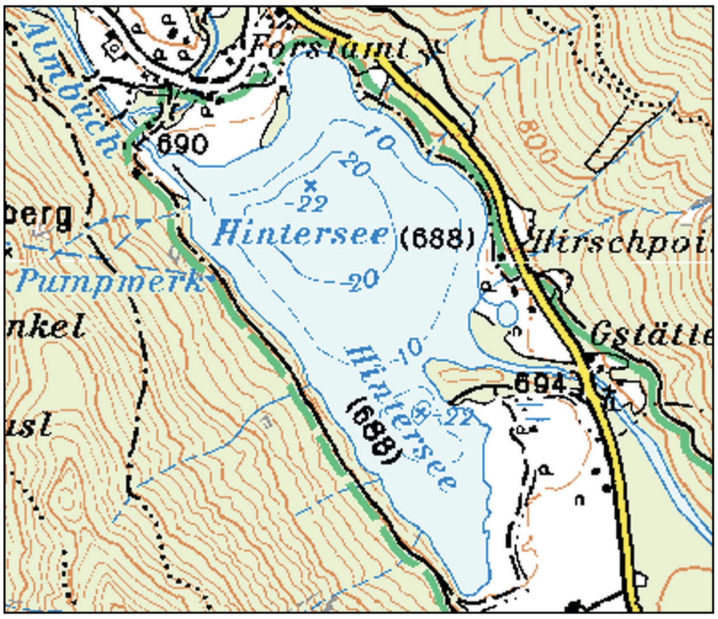


Figure 1–5 Example of a lake with more than one contributing river and one outflow (Source: Austrian Map 3D - Bundesamt für Eich- und Vermessungs-wesen BEV)

1.1.2.4 Wide rivers

Wide rivers - always dependent on the mapping scale - comparably to lakes form an area element. Sometimes only parts of the river show a compound cross section (this is where the river is much broader than the average of the river). The main current flow line has to be found. River bank and shoreline are the lines where other rivers join in the real world and differ considerably from the flow line.

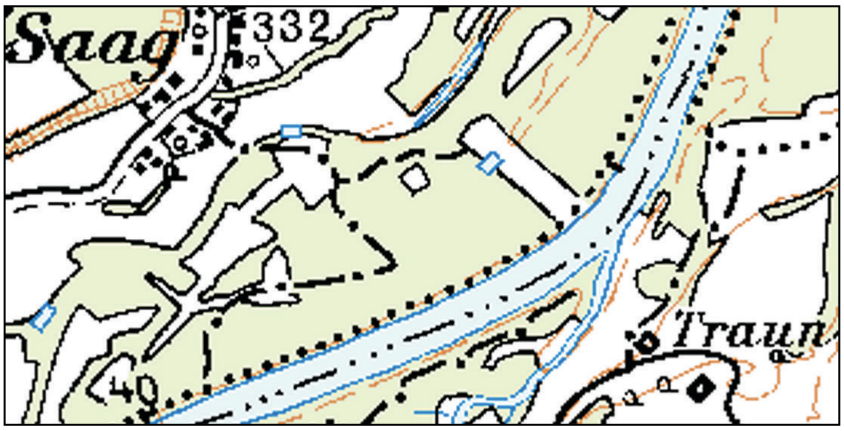


Figure 1–6 Example of a wide river, one tributary from south (Source: Austrian Map 3D - Bundesamt für Eich- und Ver-messungswesen BEV)

1.1.2.5 Anthropogenic interventions

Anthropogenic changes in the environment are omnipresent. Many hydraulic structures are already available, and some are yet to be built, to utilize water resources and to help to protect against negative effects of water. In the hydrological environment artificial watercourses are generated, watercourses are changed, and their flow is modified. The following list shows some of the greater structures that influence the hydrological network:

- **Channel**: A channel is an artificial watercourse. It either periodically or continuously contains moving water or forms a connecting link between two natural or artificial bodies of water (c.f. [IGH]).
- **Floodway**: A special sort of channel that is used to divert flows from a point upstream of a region to a point downstream of this region (c.f. [IGH]).
- **Diversion**: Transfer from one watercourse into another (c.f. [IGH]).
- **Weir**: A weir is an overflow structure that may be used for controlling the upstream water level or for measuring discharge or for both. This is used for run-of-the-river hydroelectric power generation or the feeding of millstreams of watermills, navigational purposes, fish breeding, recreation, ... (c.f. [IGH, BDS]).
- **Dam**: A dam is a structure that serves the primary purpose of retaining water. There are several types of dams, like arch dams, buttress dams, gravity dams, embankment dams, mostly depending on the type of valley (narrow V-shape or wide valley types) and the underground (rock or soil) (c.f. [BDS 2013, INTERNATIONAL RIVER 2013]). Conventional hydroelectric power stations need a dam and reservoir.
- **Barrage**: A barrage is a barrier across a river, sometimes with gates. They are often used to control and stabilize water flow for irrigation systems. Barrages that are built at the mouth of rivers or lagoons to prevent tidal incursions or utilize the tidal flow for tidal power are known as tidal barrages.
- **Reservoir**: Are water storages either behind dams or as bank-side reservoir somewhere along or near the river, where an embankment is used to encircle the water withdrawn from the river. Reservoirs can be created for different purposes, which are: hydroelectricity, water supply, irrigation, snowmaking, recreation, flood control and flow balancing. A special type is the pumped storage, which is a storage reservoir for hydroelectricity into which water is pumped (sometimes over some distance from watercourses not nearby) (c.f. [IGH 2013, BDS 2013]).
- **Crossover**: Through canalization of a river it becomes possible that two watercourses are crossing each other without having contact.

1.1.3 Time variance in flow

Many headwaters have time varying starting points that depend on the available surface runoff, which is strongly influenced by precipitation (see above). Therefore, it is not always clear where a river has its starting point or upper segment.

> *If one follows a river upstream, the flow becomes less and less, and the straem* [!] *forks into a network of drainage channels which lose themselves perhaps in a multitude of small gulleys. Which of these then constitute "first order stream segments"?* [SCHEIDEGGER 1966]

If the rainfall is steady and ample, the rivers start at higher altitudes. If there is not enough rainfall or none at all, the rivers' variable starting points shift to lower altitudes, because it takes more time until enough water is gathered to form a stream. Some reaches may fall dry in periods of no precipitation and exist only some time of the year, because there is limited baseflow or a highly permeable underground. They are therefore called intermittent rivers. Rivers that are potentially perennial are called perennial or permanent rivers. Rivers containing alternating stretches of perennial and intermittent flow are called interrupted streams. (About stream building c.f. [STRAHLER 2006, ZEPP 2002]).

There is variation possible concerning not only the starting point but also in the course of a river. Some rivers shift their streambed with time, and depending on the discharge the current flow line of a river also can change. (About channel stability c.f. [CHORLEY 1984], p. 302 ff.)

For the hydrologic network this means that there is a time factor to be considered. The network's continuity is dependent on the availability of a flow. Consequently in humid seasons the network may be considerably longer than in dry seasons.

1.1.4 Changes in flow direction

In flat areas changes in flow direction can happen depending on the discharge of the main river. This mostly concerns channels or river segments in estuaries or channels in the vicinity of big rivers flowing on flat plains.

1.2 The representation of the surface water network

There is a set of issues concerning the crossover of the surface water network to its representations (either the analogue form or the hydrological network). The most important are:

a) The representation of the physical elements of the hydrologic network in a model in general:

> – The producers of comprehensive data (e.g. the national land survey) already have expert knowledge on these representation topics – diffuse boundaries around lakes and along rivers have to be considered.

b) The mapping scale (see 1.2.1)

c) Complicated parts of the surface water network:

> - Where the catchment of a river segment can only be insufficiently identified (c.f. 1.1.2.2) - in these areas the discharge is often split or unclear, which has to be considered in a quantitative way in hydrological models.
> - Where there have to be defined virtual flow lines (e.g. through lakes) to create a continuous flow pattern.
> - Where there are anthropogenic interventions in the natural course of the rivers or the administrative requests of the river basin management, e.g. the segmentation of a river at dams, weirs or other river engineering measures. (c.f. 1.1.2.5)

d) The time variance in the surface water network - changing starting points in the hydrological network have to be considered (c.f. 1.1.3 and 1.1.4)

1.2.1 Scale dependency

The representation of the physical surface of the earth has been the topic of cartographers for a long time. Although interests have shifted a lot and techniques have developed since the beginnings of cartography, there is still the question about scale to be answered when planning a map and collecting the data for this map. This is not only valid for map data sets but for all spatial data sets in general:

It is clear (and widely accepted ;-) that a representation is not a 1 : 1 copy but can only be a model of "reality". Therefore, some kind of generalization is to be expected when creating a representation. If we have a closer look to the crossover process between the real world object and its representation, a decision has to be made on the degree of detail (c.f. [MULLER 1991], p. 457 seq.), which leads to some introductory questions:

- What is the size and form of the smallest elements, which must be represented?
- How important is the natural form of the elements?
- What about locational and attribute accuracy?
- When is the dataset complete?

In classical cartography this is the point where discussion about the scale fractions between reality and model and the thresholds of perception takes place. Because in a map the storage of the information is identical with its visualization - it is the map graphic that holds both. "*The threshold of perception is the minimum size of a graphic element which can be seen with the naked eye under normal circumstances*."([ROULEAU 1984] p. 104) These minimum sizes are necessary to be able to depict the singular element in the texture of a map. Arnberger gave a list of threshold values [ARNBERGER 1975]:

Isolated graphical form	Size in mm when form is black on white background	Size in mm when form is coloured or on tinted background with proper differentiation of brightness and colour weight
Line	0.05	0.07
Double line (line/spacing)	0.05/0.20	0.07/0.20
Point	0.24	0.30

Table 1 Minimum sizes for a viewing distance of 25 cm and normal lighting (translated excerpt from [ARNBERGER 1975], p. 227)

The combination of scale and minimum sizes shows explicitly what can be represented in a map and what cannot be represented. We can look at this as a sort of best case scenario for mapped content, because although these threshold numbers have existed for some time there is still the factor "map maker" who may omit information even though it would be representable at a certain scale, due to other reasons (e.g. aesthetics, economic reasons, ...) and may add important information that is smaller than the minimum size (thus displace other information).

In maps the "real world" surface water networks are represented in a discrete way, either as an illustration of the centreline, or as an areal illustration. Both types of illustration can occur in one map and are dependent on the size of the real world object and the map scale.

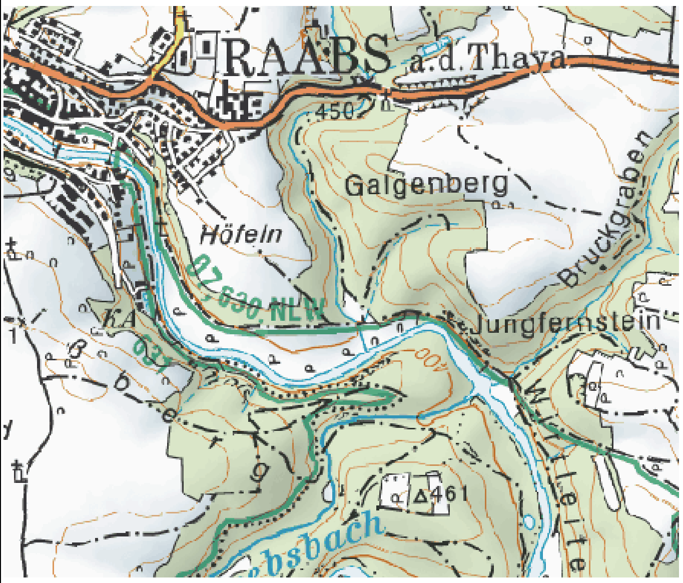
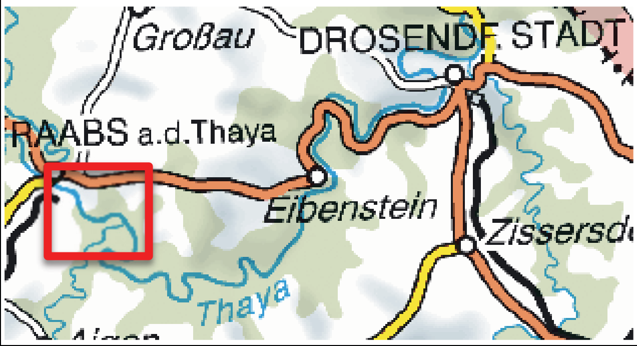
 RAABS a.d.Thaya 450 Galgenberg Höfeln Bruckgraben Jungfernstein KA Δ461 bsbach	1:25,000 - River Thaya is represented as areal element. Smaller rivers can only be represented as centrelines.
 Großau DROSENDF. STADT RAABS a.d.Thaya Eibenstein Zissersd Thaya	Original scale 1:500,000 enlarged to 1 : 250,000 - River Thaya is represented as line. The red rectangle shows the extent of the 1:25,000 example. Small rivers and intermittent rivers are omitted.

Figure 1–7 Example of the differences in the representation of rivers depending on scale. (Source: Austrian Map 3D - Bundesamt für Eich- und Vermessungswesen BEV)

Although I was not to happy that a widely known GIS specialist as Michael Goodchild uses the ambiguity of the term scale as lead for his articles about scale [GOODCHILD 2001, GOODCHILD 2011], because this ambiguity simply occurs out of the terms use in different professions (cartographer and environmental scientist), which happens a lot when people of different professions meet, I agree that the classic cartographic use of scale as ratio between real world, and stored data has to be adapted for the GIS environment.

In the GIS environment the representation of the data is (and this applies to vector data even more than to raster data) split in a storage and a visualization part. The separation of data storage and data visualization makes it possible to present a data set at a different number of scaling ratios between real world and mapped data (sometimes disregarding all cartographic principles). When it is not stored with the data (e.g. as metadata) then there is usually no information about the detail that is represented in the dataset (and this applies again more to vector than to raster data, because the information about the first two dimension is explicit in the raster data itself).

To describe detail for digital data the term **resolution** is often used. The resolution in raster case is for the first two dimensions explicit in the size of the raster cells[5], and it even shows in a map, when a raster dataset is used for scales not appropriate for the used map scale (e.g. when one easily can depict the single raster cell in the map). The resolution of the third dimension has to be found out from the data description.

The (minimum possible) distance between points is often used to describe the vector resolution. But vector-mapped datasets are almost always based on a world-view that discretizes the objects. Thus, the more points are set to distinguish an object from its surroundings (e.g. a river), the higher its resolution would be.

> *Unfortunately this often creates the mistaken impression that vector data sets have infinitely fine resolution*. ([GOODCHILD 2011], p. 6).

It has to be kept in mind that the step to discretize the object - e.g. the shoreline of a river - has already taken place and has introduced generalization. Even if the basis of the mapping is the reality "*Any observation of the physical occurs within the context of some model or abstraction*" [RAPER 1995].

The idea of fractals gives an additional interesting starting point to thoughts about detail. (c.f. [GOODCHILD 1987, KLINKENBERG 1994, MANDELBROT 1967, MARITAN 1996, TARBOTON 1988, WHEATCRAFT 1986])

> "*One of the most important features of a fractal object is that its "degree of irregularity" is independent of scale. A simple example of this is a coastline.*

[5] At least if there never was any downscaling process included in the data lineage.

When viewed from space, a coastline appears very irregular, but it appears just as irregular at an altitude of 1000 feet. Even when walking along the beach, the water meets the sand in a complex irregular pattern that is not a straight line." *([WHEATCRAFT 1986], p. 568)*

If one has a closer look at the ideas that exist around the term "detail" in the scientific articles written on fractals, it appears possible to go into infinite detail when mapping data. Especially in the 1980s and 1990s articles about fractals concerning measurements in hydrology and geomorphology were published and showed that in a world, where data can be so easily obtained and combined in a GIS, it is extremely important to keep the fractal nature of many natural elements (e.g. rivers) in mind.

Thus, although visual representation is not the first interest of the work with the network itself, generalization is an issue, because it influences not only the visual appearance and ability to be integrated with other data, the measured lengths and comparability of the elements of surface water network, but it can also introduce changes in the networks' structure. Scheidegger as far back as 1966 has already stated this in connection with stream ordering:

Thus the choice of what is a first order stream segment, in fact, has in reality already been made by the map maker. Perhaps, a different map maker would have followed the "blue lines" a little farther upstream, through an additional fork, and thus the stream ordering should have been different than what was first arrived at. Similarly, if the scale of the map is increased (say, from 1: 100,000 to 1: 10,000), it stands to reason that more tributaries in a river system will become visible so that all the stream orders become affected ([SCHEIDEGGER 1966], p. 57)

Change the term "map maker" in Scheidegger's statement against "data collector" and the scale specifications against specifications of detail, and this statement becomes valid for digital and analogue data. Not only the starting points and the number of tributaries are different, but also the topology in the surface water network may be changed, e.g. because three rivers may come together at the same point in a dataset with low detail, which may not be the case in a more detailed version of the data of the same rivers; or small connections may be missing in low detail data but are available in high detail. To work with stream data on more than one scale (e.g. river basin management vs.

management on catchment level) a **level of detail** approach similarly to the ones used in virtual realities is advisable. The generalization has to follow a strict hierarchy based procedure that allows a reference from certain low-level river nodes or rivers to high-level ones to ensure forward and backward compatibility.

1.2.2 Representation as a digital network

> *Network data structures were one of the earliest representations in geographic information systems (GIS), and network analysis remains one of the most significant and persistent research areas in geographic information science. (...)*
>
> *Network analysis in GIS rests firmly on the theoretical foundation of the mathematical subdisciplines of graph theory and topology. Any graph or network (...) consists of a set of vertices and the edges that connect them. ([CURTIN 2007], p. 103)*

With a closer look at a surface water network, many terms for surface water network elements can be mapped directly to the terms of graph theory [the following terms from graph theory are based on the works of [AHUJA 1993, BONDY 1976, CHARTRAND 1977, DIESTEL 2006, KÖNIG 1950, LUCE 1952, WEISSTEIN].

A **graph G** is a **set V** of items (called **vertices)** connected by a set of **edges E**. An **incident function ψ** defines which edge e_k is associated with which 2-element Subset of V.

$$V(G) = \{v_1, v_2, ..., v_n\}$$

$$E(G) = \{e_1, e_2, ..., e_n\}$$

$$\psi(e_k) = \{(v_i v_j) | \forall v \in V(G)\}$$

An edge (v_i, v_j) is usually written as $v_i v_j$.

Between the vertices and edges a binary relation is defined (that is called **adjacency**).

Whenever two vertices are associated with the same edge they are called **adjacent vertices**, when two edges are associated with the same vertex they are **adjacent edges**.

If the graph is **undirected**, the adjacency relation defined by the edges is symmetric. This means that for every pair $v_i v_j$ there also exists a pair $v_j v_i$. Otherwise the graph is called a **directed graph** (or oriented graph).

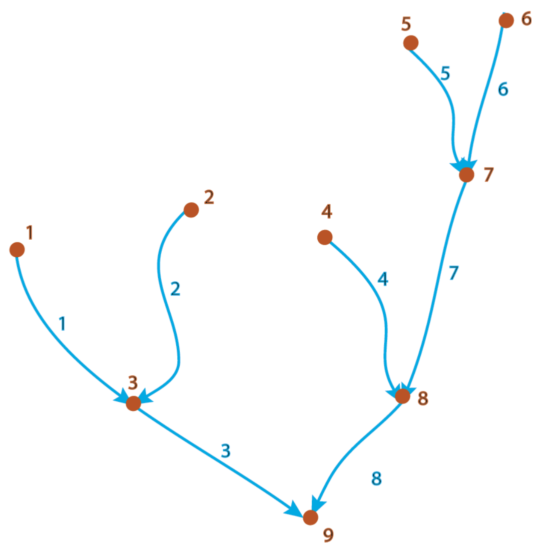


Figure 1–8 Line representation of a simple surface water network

Figure 1–8 shows a line representation of a simple ideal surface water network. The orange dots that represent the beginnings and endpoints of watercourses like sources, confluences and river mouth are the graph theoretical vertices. The blue lines that are the representation of the watercourses would be graph theoretical representation of edges, the arrows indicate a direction. As only the connection of the vertices matters, there is no immediate need to make a generalized version of the rivers, as often seen in graph representations.

Each set of data (vertices and edges) has its distinct unique values. One edge with identical start and end vertex ($v_i \equiv v_j$) is called a **loop**, when the ends differ ($v_i \neq v_j$) this is defined as a **link**. If the graph does not allow loops, adjacency is **irreflexive**. If a sequence of adjacent edges has the same start and end vertex this is called a **cycle**. A graph without cycles is an **acyclic** graph.

Usually the river water runs in a certain direction (which is downhill). Therefore, edges are usually only passed through in one direction. Thus it is a **directed graph**. Both being irreflexive and acyclic should be expected from the surface water network graph.

In a surface water network the used number of vertex pairs is only a small fraction of what would be available of pairs, because most of the vertices are adjacent to no more than three other vertices - in very rare cases, when there is one confluence of three rivers, there may be four vertices connected or in an extensively braided river the connection may be even made by up to 4 or 5 vertices. This would be called a sparse graph, which is a rather vague definition and can be seen as the opposite of a dense graph that is described as a graph, which uses most of the possible pairs of vertices to form its edges. The formal definition of sparse (and dense) graphs can be found in [PREISS 2000]:

A *sparse graph* is a graph G=(V, E) in which $|E| << O(|V|^2)$.[6]

A *dense graph* is a graph G=(V, E) in which $|E| = O(|V|^2)$.

The following text on graph representations is based on [AHUJA 1993, CHARTRAND 1977, CORMEN 2011, PREISS 1998, PREISS 1999, SAAKE 2004, TURAU 1996, VORNBERGER 2009, ZHAN 1998].

A node-node **adjacency matrix A** (also called connection matrix) of a graph G, with its edge set E, labels all adjacent vertex pairs v_iv_j with 1 and all other possible vertex pairs with 0.

$$A_{ij} = \begin{cases} 1 & (v_iv_j) \in E \\ 0 & otherwise \end{cases}$$

A is a square matrix with n rows and n columns: $\mathbf{O(|V|^2)}$.

[6] About O-Notations see 4.1.1.2 Asymptotic notation, which does not only apply to define maximum running time but also to define the order of the maximum needed space.

	to-nodes									
from-nodes		1	2	3	4	5	6	7	8	9
	1	0	0	1	0	0	0	0	0	0
	2	0	0	1	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	0	0	1	0
	5	0	0	0	0	0	0	1	0	0
	6	0	0	0	0	0	0	1	0	0
	7	0	0	0	0	0	0	0	1	0
	8	0	0	0	0	0	0	0	0	1
	9	0	0	0	0	0	0	0	0	0

Table 2 Example of an adjacency matrix for the directed graph in Figure 1–8

It is very easy to discover if a connection exists from v_i to v_j with node-node adjacency matrices. This can be done in constant time. But the matrix is quadratic in size, and although most of the connections between v_i and v_j do not exist because our graph is sparse and therefore the matrix is sparse too, it would use a huge amount of disk space.

Other values concerning the network, like costs or capacities, can be added by using a similar matrix for each variable. That means that for every additional variable we need additional space of **$O(|V|^2)$.**

$$C_{ij} = \begin{cases} c(v_i v_j) & (v_i v_j) \in E \\ 0 & i = j \\ \infty & otherwise \end{cases}$$

	to-nodes								
from-nodes		1	2	3	4	5	6	7	8	9
	1	0	∞	250.5	∞	∞	∞	∞	∞	∞
	2	∞	0	307.2	∞	∞	∞	∞	∞	∞
	3	∞	∞	0	∞	∞	∞	∞	∞	271.5
	4	∞	∞	∞	0	∞	∞	∞	278.4	∞
	5	∞	∞	∞	∞	0	∞	194.5	∞	∞
	6	∞	∞	∞	∞	∞	0	189.7	∞	∞
	7	∞	∞	∞	∞	∞	∞	0	299.5	∞
	8	∞	∞	∞	∞	∞	∞	∞	0	190.2
	9	∞	∞	∞	∞	∞	∞	∞	∞	0

Table 3 Example of a cost matrix belonging to the adjacency matrix in Table 2 - the values are the costs (in our example we use distances). When there is no connection between two nodes the costs are infinitely high. If from-node is equal to-node then one stays in the same location, thus the costs are zero.

Besides the node-node adjacency matrix shown above, it is also possible to create an edge-edge adjacency matrix. The edge-edge adjacency matrix (again for directed graphs) shows for each edge (from-edge) if its endpoint is the starting point of another edge (to-edge). Advantages and disadvantages of edge-edge adjacency matrices are comparable with those of node-node adjacency matrices (see above).

	to-edge								
from-edge		1	2	3	4	5	6	7	8
	1	0	0	1	0	0	0	0	0
	2	0	0	1	0	0	0	0	0
	3	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	1
	5	0	0	0	0	0	0	1	0
	6	0	0	0	0	0	0	1	0
	7	0	0	0	0	0	0	0	1
	8	0	0	0	0	0	0	0	0

Table 4 Example of an edge-edge adjacency matrix

An alternative to an adjacency matrix concerning the data volume is the adjacency list. Since the early 1970's adjacency lists have been seen as a way to store the nonzero elements of given sparse matrices (c.f. [TEWARSON 1973], p. 4).

A singly-linked list is simply a sequence of dynamically allocated objects, each of which refers to its successor in the list. ([PREISS 1998] - http://www.brpreiss.com/books/opus5/html/page97.html)

In an adjacency list pointers lead to the neighbouring vertices of each vertex. In case of a directed graph those are the vertices where the edges that have their starting point at this certain vertex end. Need of space for an adjacency list is therefore O(|V| + |E|). Our simple example would have one list node for each node except the last one (id number 9) that directly points at NULL, which is the end of the list.

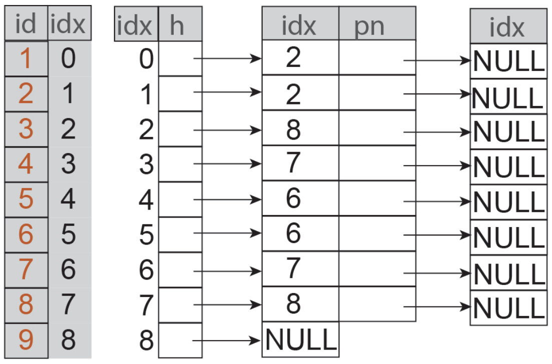


Figure 1–9 Example of an adjacency list for the directed graph in Figure 1–8

id	...unique node id
idx	...node index number
h	...list head
pn	...pointer to next node

The grey parts in Figure 1–9 (and also in Figure 1–10, Figure 1–11, Figure 1–12, and Figure 1–14) are added to achieve more clarity. They are not part of the data structure itself.
Additional values can be added to the lists by simply expanding the list structure with the necessary fields. This can be done very space-efficiently. We only need **O(|E|)** extra space for each additional variable.

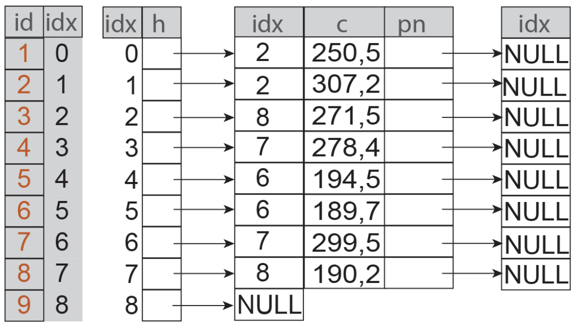


Figure 1–10 Example of an adjacency list with additional cost-field (in this case the arc length).

c	... cost

The other headers are like in Figure 1–9.

In the node-node adjacency list only the connections between the vertices are listed. It would also be possible to create an edge-edge adjacency list to show the connections between the edges. To establish the connection between the vertices and edges, the data can either be saved in an **incidence matrix** or an **incidence list**. The incidence matrix of a network contains one column for each node of the graph and one row for each edge of the graph. The starting nodes (tails) are assigned the value -1. The ending nodes (heads) are assigned the value 1.

$$I_{ke} = \begin{cases} -1 & e = (v_i v_j), k = i \\ 1 & e = (v_i v_j), k = j \\ 0 & otherwise \end{cases}$$

It is easy to determine incoming (e where I_{ke} = 1) and outgoing edges (e where I_{ke} = -1) for a node and to determine the number of incoming (= number of 1's in a row) and outgoing edges (= number of -1's in a row) at O(|E|). However, the incidence matrix is space inefficient. The matrix is of **O(|V|*|E|)** and sparse because each arc is only connected to two vertices. Each column has only two non-zero values (either -1 or 1). Thus only 2*|E| values in the matrix are not zero, which makes (**|V|*|E|)-2*|E|** zero.

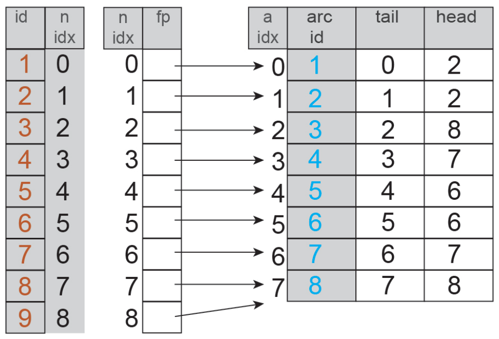
	edges								
nodes		1	2	3	4	5	6	7	8
	1	-1	0	0	0	0	0	0	0
	2	0	-1	0	0	0	0	0	0
	3	1	1	-1	0	0	0	0	0
	4	0	0	0	-1	0	0	0	0
	5	0	0	0	0	-1	0	0	0
	6	0	0	0	0	0	-1	0	0
	7	0	0	0	0	1	1	-1	0
	8	0	0	0	1	0	0	1	-1
	9	0	0	1	0	0	0	0	1

Table 5 Example of an incidence matrix for the directed graph in Figure 1–8

Additional variables for cost or capacity values can be obtained in creating a separate matrix with variables*edges. Its space is of **O(|E|)** for every variable.

In an incidence list representation edges and their corresponding starting nodes and ending nodes are stored. A special version is also known as **Forward Star Representation**.

The Forward Star Representation uses fixed-size arrays that contain the arcs' indices and each arc's tail and head (this is the incidence list part of the data structure). From each from-node there exists a pointer to the first arc that emanates from the node.



id	n idx
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8

n idx	fp
0	
1	
2	
3	
4	
5	
6	
7	
8	

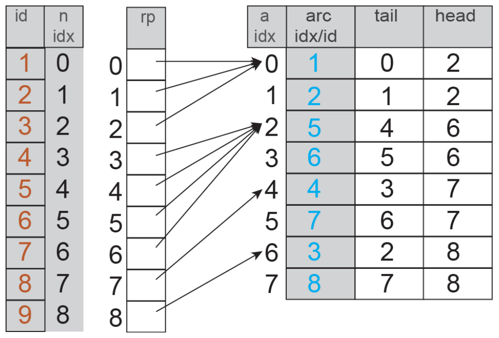
a idx	arc id	tail	head
0	1	0	2
1	2	1	2
2	3	2	8
3	4	3	7
4	5	4	6
5	6	5	6
6	7	6	7
7	8	7	8

Figure 1–11 Example of a Forward Star Representation for the directed graph in Figure 1–8.

fp	... forward pointer
a idx	... arc index (edges)
arc id	... unique arc id (edges)
tail	... starting node index
head	... ending node index

Other headers are like in Figure 1–9.

The Forward Star Representation provides information about the emanating arcs of each node. To determine the incoming arcs of a node the **Reverse Star Representation** can be used.



id	n idx
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8

	rp
0	
1	
2	
3	
4	
5	
6	
7	
8	

a idx	arc idx/id	tail	head
0	1	0	2
1	2	1	2
2	5	4	6
3	6	5	6
4	4	3	7
5	7	6	7
6	3	2	8
7	8	7	8

Figure 1–12 Example of a Reverse Star Representation for the directed graph in Figure 1–8.

rp	... reverse pointer

The other headers are like in Figure 1–11.

Additional variables (costs, etc.) can be added after the tail and head fields. Its space is of **O(|E|)** for every variable.

The surface water network can contain braided river stretches, where there can be more than one edge between two points on the river stretch. These edges are called **multiple edges**. If the multiple edges are oriented towards the same vertex, as we can expect in downhill flowing rivers, they are **parallel edges**. In graph theory the graph is called a **multigraph**.

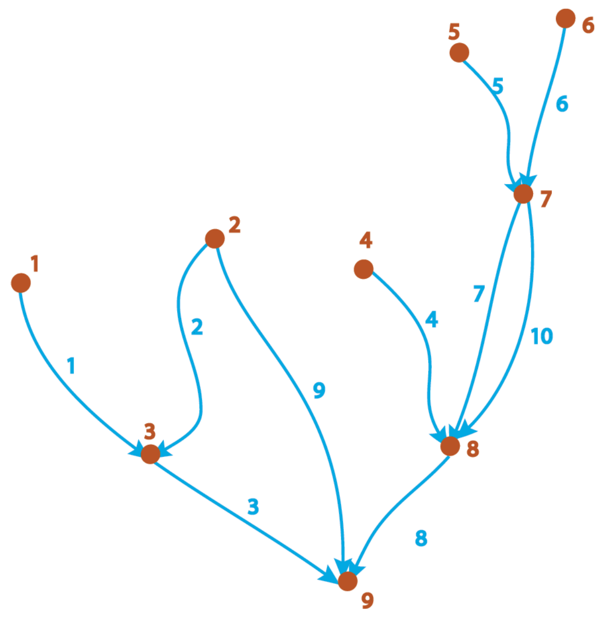


Figure 1–13 Line representation of a network with multiple edges (7 and 10)

Multiple edges cause problems in the adjacency matrix representation as shown in Table 6 (yellow cell). The marked value represents more than one connection between vertices. It is possible to show with an adjacency matrix that two vertices are adjacent, but it is not possible to show how many connections there are. In an adjacency list the problem of multiple edges can be handled better by pointing more than once to an adjacent vertex. Even the addition of different values for costs/capacities would not cause a problem.

	to-nodes								
		1	2	3	4	5	6	7	8	9
from-nodes	1	0	0	1	0	0	0	0	0	0
	2	0	0	1	0	0	0	0	0	1
	3	0	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	0	0	1	0
	5	0	0	0	0	0	0	1	0	0
	6	0	0	0	0	0	0	1	0	0
	7	0	0	0	0	0	0	0	1	0
	8	0	0	0	0	0	0	0	0	1
	9	0	0	0	0	0	0	0	0	0

Table 6 Example of an adjacency matrix for the directed graph in Figure 1–13

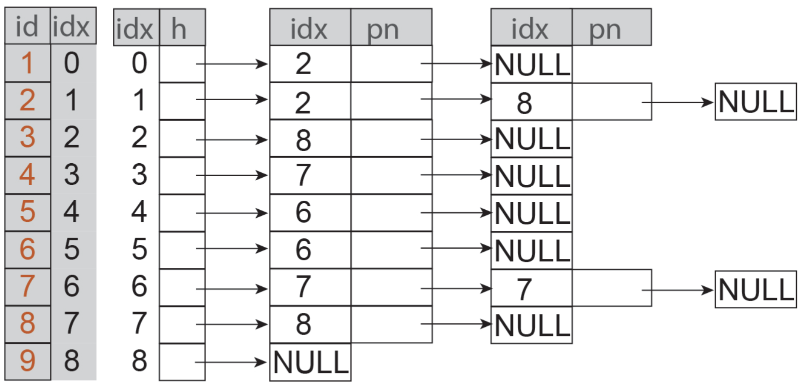


Figure 1–14 Example of an adjacency list for the directed graph in Figure 1–13

	edges									
nodes		1	2	3	4	5	6	7	8	9	10
	1	-1	0	0	0	0	0	0	0	0	0
	2	0	-1	0	0	0	0	0	0	-1	0
	3	1	1	-1	0	0	0	0	0	0	0
	4	0	0	0	-1	0	0	0	0	0	0
	5	0	0	0	0	-1	0	0	0	0	0
	6	0	0	0	0	0	-1	0	0	0	0
	7	0	0	0	0	1	1	-1	0	0	-1
	8	0	0	0	1	0	0	1	-1	0	1
	9	0	1	1	0	0	0	0	1	1	0

Table 7 Example of an incidence matrix for the directed graph in Figure 1–13

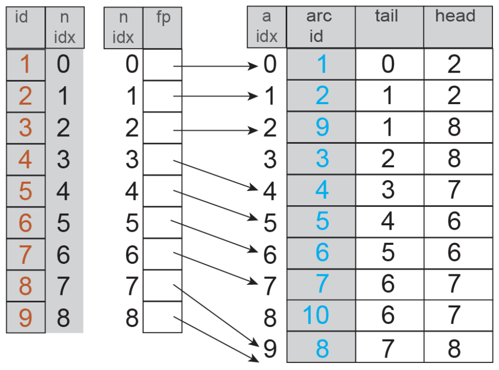


Figure 1–15 Example of a forward star representation for the directed graph in Figure 1–13. Due to branching there is more than one edge per node.

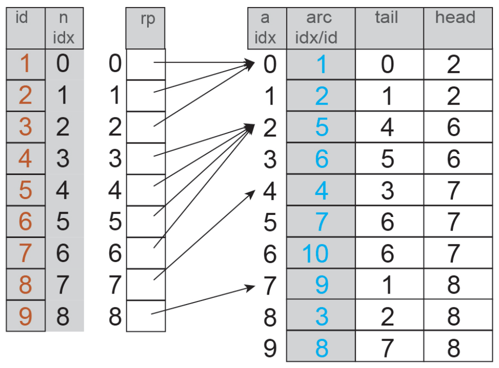


Figure 1–16 Example of a backward star representation for the directed graph in Figure 1–13.

In case the surface water network contains crossovers (which has to be verified), it may be possible that the graph cannot be drawn in the plane without two edges crossing each other. Thus the surface water network could therefore be a **nonplanar graph.** Both having a nonplanar graph and a multigraph has consequences in graph theory.

If time variances are drawn into conclusion we talk about a **dynamic graph.** The dynamic would influence the data model that makes it possible to derive data sets for special cases (like floods or changes in flow direction).

A flow network is a special form of a directed acyclic graph. It has weights on the edges that define the flow capacity of the edges, and two kinds of specially marked nodes can be distinguished:

- the **sources**: vertices with no incoming but one or several outgoing edges
- the **sinks**: vertices with no outgoing but several incoming edges

There are many Geographic Information Systems that provide a network model. They range from the commercial GIS products to freeware. The finer implementation of the network in the GIS system is in some cases made clear, but mostly hidden behind interfaces, thus difficult to come by. In general the network in GIS is based on the vector geometry. Network junctions are created from the vector geometry nodes. The edges themselves with all their vertices or curve functions are relevant for the establishing of the topological connection and for the capacity and weight values they contribute. In different connectivity tables, which vary in their structure between the different GISs, the connection between the nodes, and the nodes and edges, as well as between the edges is described.

The following example shows the realization of a network model in ESRI ArcGIS SDE (source: http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html# /na/002n0000007r000000/)

(Text in *italic* is the original text from the homepage)

- ***GDB_ITEMS — Geometric networks are tracked in here***
- ***GDB_ITEMTYPES— Stores a value indicating that the object is a geometric network.***
- ***GDB_ITEMRELATIONSHIPS— stores information on how the network and the feature dataset it is in are related.***

Tables in the geodatabase that begin with N_ store information about networks.

- **N_<ID>_PROPS - Contains a summary description of a network's properties, such as element counts and maximum EID values.**
- ***N_<ID>_DESC— table describes the elements of a network. This is a normalized table whose row count is equal to the number of junctions and the number of edges in a geometric network. Each network element has a unique element identifier EID and the ELEMENTTYPE field shows if the network element is a junction (1) or an edge (2)***

The following tables use BLOB-pages to store the information. The detailed structure of the data is therefore hidden (BLOBs can store a wide range of formats, even executables).

- ***N_<ID>_E<#>—Describes network edge weights; # = 2, 3, 4, or 5***
- ***N_<ID>_EDESC—Describes the edges in a network***
- ***n_estatus—Describes the status of each edge including its deleted and disabled states***
- ***N_<ID>_ETOPO—Describes the network edge topology or connectivity***
- ***N_<ID>_FLODIR—Describes the network flow direction***
- ***N_<ID>_J<#>—Describes network junction weights; # = 0 or 1***
- ***N_<ID>_JDESC—Describes the network junctions***
- ***N_<ID>_JSTATUS—Describes the status of each network junction including its deleted and disabled states***
- ***N_<ID>_JTOPO—Describes the connectivity of junction elements with edge elements***
- ***N_<ID>_JTOPO2—Describes the connectivity of junction elements with edge elements when there are multiple edges connected to a single junction***
- ***N_<ID>_T<#>*—Describes the weight values of each turn element***
- ***N_<ID>_TDEFN*—Defines each turn element by listing the edges and junctions that make up the turn***
- ***N_<ID>_TDEFN2*—Overflow table for the turn element definition; for example, if multiple edges make up a turn***
- ***N_<ID>_TDESC*—Describes the turns in a network***
- ***N _<ID>_TSTATUS*—Describes the status of each network turn, including its deleted and disabled states***

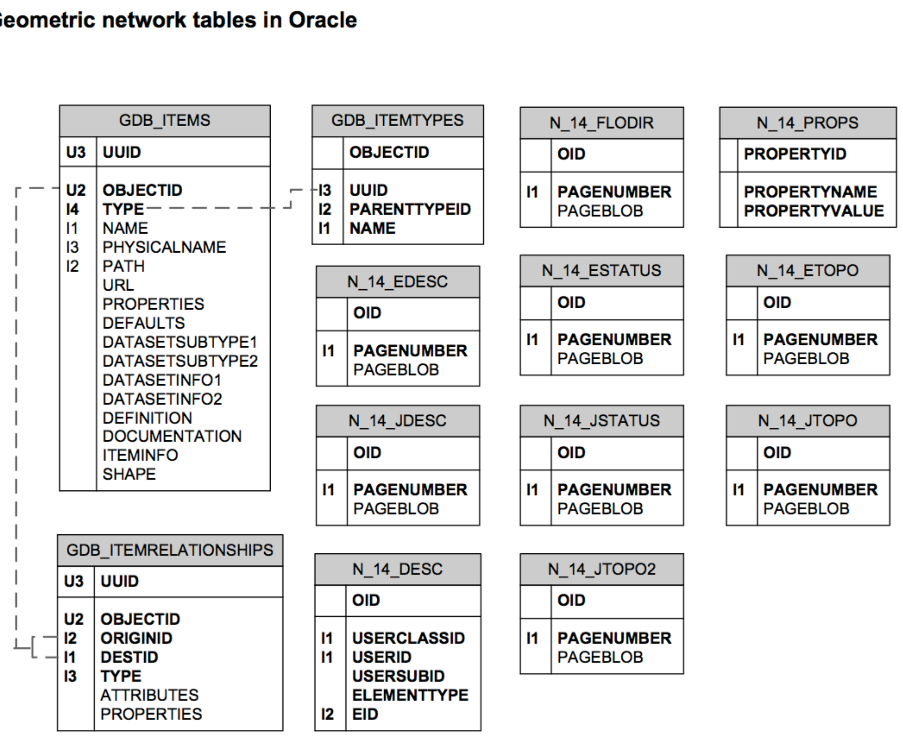


Figure 1–17 ESRI SDE Network Model

Dashed lines show implicit relationships between columns

2 Catchment area and hydrologic network

2.1 The hydrologic catchment

The hydrologic catchment (or drainage basin) "*is the entire area providing runoff to, and sustaining part or all of the streamflow*" of a surface water network (c.f. [GREGORY 1973], p. 37). Ideally it is an area with one outlet point (for coastal catchments it is a coast line) for the surface water runoff. The catchment boundary (or drainage divide or watershed divide) is the line that divides the water runoff. It can be determined with a topographic map or with a digital elevation model, in finding the highest points between two rivers/runoff lines and connecting them to form a polygon.

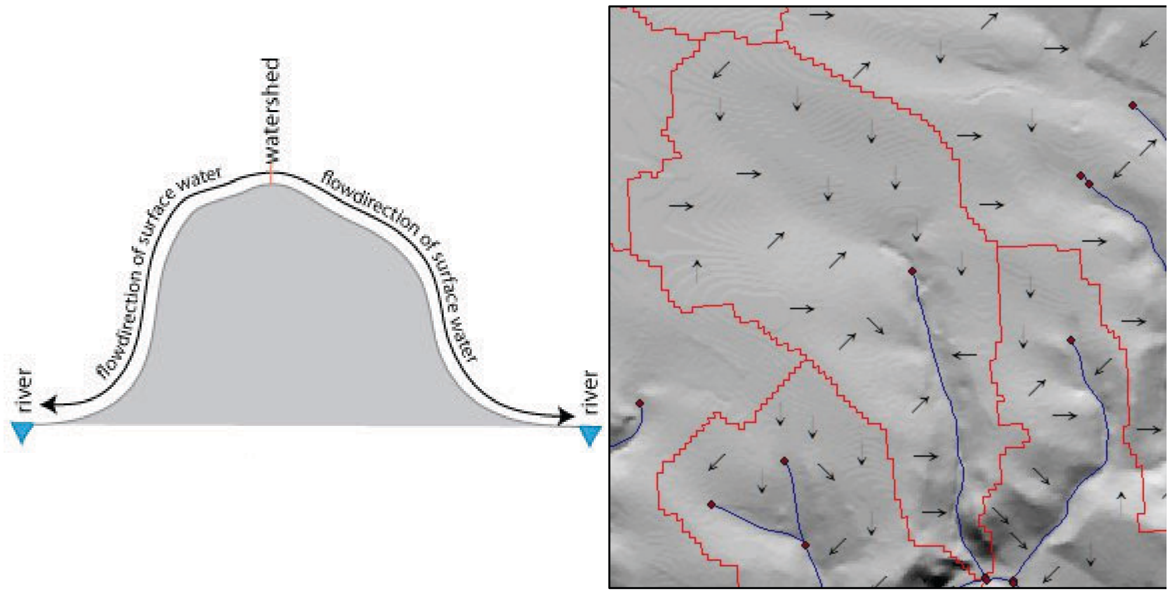

watershed
flowdirection of surface water
flowdirection of surface water
river
river

Figure 2–1 Watershed indicated in red - Flow direction of the surface water shown with black arrows

The catchment boundary is - comparable to the river segments - strongly influenced by its fractal dimension. The more detail it is surveyed in, the longer will be its length, which has to be taken into account whenever the perimeter of the catchment is included into models (see also [BREYER 1991] and section 3.1.3.2)

2.2 The connection between surface water network and catchment

In the ideal typical connection between surface water network and catchment there is a distinct association between the single surface water network element and a catchment area. For a simple surface water system this means that a river segment ends usually at the point where three (or more) catchments touch - those are (1) the catchment of the river segment, (2) the catchment(s) of the other confluence(s) and (3) the catchment where both (1 and 2) drain to. In direction of the flow, the areas can be aggregated to form one non-overlapping catchment area for a river's drainage system.

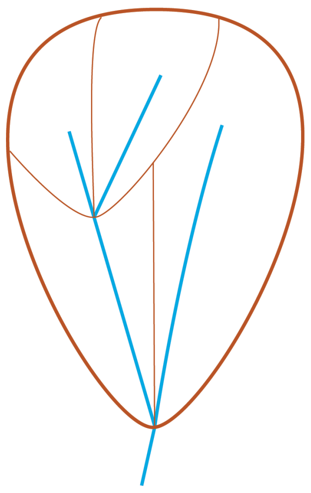


Figure 2–2 Each segment of the surface water network should have exactly one catchment area

2.2.1 Segmented/Overlapping Catchments

Problematic exceptions of the surface water network have already been mentioned in the sections 1.1.2, 1.1.3 and 1.1.4. They have an influence on the definition of the watershed divide and can happen in small as well as in large environments. Ideally each watershed divide surrounds exactly one catchment area without area overlaps. But when there are anthropogenic interventions like crossovers and diversions, or natural causes like rivers flowing underground and other rivers forming atop of them, then it can happen that a catchment may be segmented in more than one area or overlap with catchments of other surface water elements.

In some cases the catchments have to be divided artificially into sub catchments because there are anthropogenic structures (weirs, dams, barrages...) for which it is desirable to know the catchment area for hydrologic/hydrometric aspects.

2.2.2 Groundwater

The groundwater flow forms the baseflow for many surface waters.

> *As the groundwater flows, particularly for small catchments, often cross the topographical divides, the surface water divides (overland flow, drainage flow) and the groundwater divides in some cases differ significantly. [HENRIKSEN 2003]*

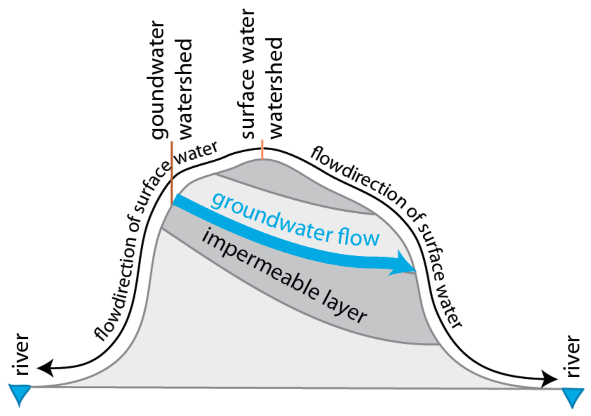


Figure 2–3 Difference between surface and groundwater catchment

To ignore the groundwater bodies is therefore a strong simplification of the surface water catchments. The true area of a catchment would have to be defined in a much more complicated procedure than only using a DEM or appropriate map. It would have to include geologic and hydrologic factors that would exceed the frame of this thesis.

2.2.3 Floods

In flat areas with only a few meters difference in altitude between drainage divide and river bed, a flood can cause temporary changes in catchments, because the drainage divide may be overflown, and waters that usually flow separately are joined - and so are their catchments.

2.3 The representation of the catchment

The catchment is generally understood and therefore represented as areal feature. If we stay in the plane it is a common way to use topological functions to check for overlaps and gaps in areal features. With a look on the surface water drainage only, topology is therefore used to control the area and make sure that there are no uncovered parts and no overlaps. Additionally, there will have to be corrections made for those areas that cannot be represented in the plane, meaning the cases discussed in the sections 2.2.1, 2.2.2 and 2.2.3. For those cases we will have to either introduce a third dimension and allow the overlap of areas or to accept catchments that consist of more than one part.

The connection between the catchment and the network will be based on the vector data's topology. The better the network and the catchment correspond, the easier it is to describe their connection based on the topology without need to set manual relations between the two levels. The connectivity between the catchments and between catchment and surface water network can be derived from their position or represented by connectivity tables.

Due to the existence of catchment with no digitally defined flow lines but with a known connection to the other catchments, this has to be specially considered when creating a data model. As the catchments have connections to each other similar to the rivers, it suggests itself to use a similar data representation for their connectivity and create a catchment network. This could be done for example in representing the catchment polygons with points and the connections with edges, like in a river network or just in form of a node adjacency list, as described in 1.2.2.

3 Where and how is surface water network data (incl. catchments) used?

Surface water network analysis, which includes the catchments, is not constricted to the interests of one science or one administrative body. Many different interest groups in the WFD environment seem to deal with surface water networks. This is an attempt to address the different groups and find out how surface water networks are used. The conclusions at the end of this chapter will serve as a basis to find or formulate expedient algorithms for the surface water network and its corresponding catchments. The information was compiled from scientific articles and by talking to and working with people of different sciences and administrative bodies.

3.1 Surface water network/catchment uses in earth sciences

Many of the earth sciences are connected to research of the surface water network. Geomorphology as scientific study of the landforms and the processes of landform shaping and hydrology as the study of the movement, distribution (quantitative hydrology), and water quality (qualitative hydrology) have scientific questions that deal with surface water networks and their corresponding catchments. They both use hydrography[7], which refers amongst others to the mapping or charting of water's topographic features. It involves establishing the morphology of rivers, lakes and seas, and measurements of the water bodies (lengths, volumes, depths, tides, currents) and the regional water balance. Geomorphology and hydrology themselves are practiced within other disciplines like physical geography and geology.

> *"The investigation of drainage network systems would seem to address the essential focus of a geographical approach to landform analysis. Drainage networks identify organized transport systems in space, and also operate as dynamic space-filling systems in temporal landscape evolution. The drainage*

[7] The term hydrography is used differently in different countries, the meaning used in this thesis is rather common in Austria and Switzerland. In other countries (e.g. US, Germany,...) some of the defining elements of the Austrian meaning of the word "hydrography" would be associated with "hydromorphology" (esp. the morphology), or with "hydrology".

network defines a drainage basin, and thus provides both a framework for integrating spatial elements of the land surface, and also a structure for dissecting sampling space into functional areas." *([JARVIS 1977], p. 271)*

The scientific interests in surface water networks and their catchments that arise in the earth sciences can be grouped as follows (see also [GREGORY 1973], p. 39):

- The distinct surface water element: length, depth, volume, profile, sinuosity,...
- The network: shape, topology, drainage density, stream length
- The network and its relation to the catchment: area tributary to streams
- The catchment: length, perimeter, shape

3.1.1 Orders and coding

This subchapter about river ordering and coding is a recapitulation of different sources, mainly [HORTON 1945, SCHEIDEGGER 1965, SCHEIDEGGER 1970, SCHMIDT 1984, SHREVE 1966, STRAHLER 1957, VERDIN 1999, WIMMER 1994])

River ordering is a way to examine the distribution of rivers and their tributaries in a watershed and to find scientific regularities in these distributions. River ordering was a big discussion topic of geographers and geomorphologists in the 1960s and 1970s. Although the ordering systems are not on the main research agendas any longer, they are still in use as access points to watershed research (besides more interesting topics like watershed connectivity).

Horton defined the different orders in connection with the stages in stream development. A stream is of first Horton order at the end of the first stage of its development, and with every stage gains one order (c.f. [HORTON 1945], p. 239 ff.). That means that rivers in the same Horton order are in the same stage of development. Because of this, the order numbers serve as the basis for the comparison of different fluvial systems. River order computation is a process that has a big topological component, which makes networks the ideal tool to calculate them.

The order numbers are always established for a river or river segment (depending on the ordering system) and are usually also assigned to the related catchment area. Appendix 8.1 contains an overview of the prevalent river ordering systems. One of the main differences in the ordering systems is the starting point, which means whether it is top down (order number one is at the headwaters) or bottom up (order number one is at the estuary).

River coding that is in some variations very close to river ordering (e.g. the Pfafstetter system) helps to locate river segments.

A special version of low order rivers (1 and 2 after Horton/Strahler) are the **adventitious rivers**, those connect directly to the main stream even though they are usually of a much later stage of the rivers development when the development of the river system approaches maturity. The number or the adventitious rivers is therefore of interest to the researchers (c.f. [HORTON 1945], p. 342).

The **bifurcation ratio** R_b ([HORTON 1945], p. 286) is a parameter that is derived from Horton/Strahler orders. R_b conveys how the number of rivers per order decreases with increasing order - to put it simply: the higher the order, the lower the number of rivers that can be counted for that order. R_b is the factor that gives the decrease of rivers from one order to the next higher order. The more elongated a catchment is (compared with others of the same size), the higher the average R_b usually is. The R_b is influenced by the number of adventitious streams.

$$R_b = \frac{n_i}{n_{i+1}}$$

n	... Number of river segments
i	... Order of river segments

	number of river segments	**R_b**
1st order	20	
2nd order	6	3.33
3rd order	2	3
4th order	1	2
sum	29	8.33
average R_b		2.78

Table 8 Example for the calculation of the bifurcation ratio

Other factors e.g. growth of area, growth of length per increase of order number are calculated analogue to R_b.

3.1.2 Extracting information about distinct surface water elements

A substantial description of a surface water network in hydrology and geomorphology is usually given though the different morphometric variables of the distinct surface water elements and the whole surface water network. Those are usually researched in relation to the coding.

Most of the parameters used for the description of the rivers and the catchments are usually calculated on a plane view. For some questions the relief may have to be taken into account. The area tributary to streams, for example, is dependent on the relief. Most of the parameters mentioned below will produce different values when calculated with the relief (e.g. for the length we then talk about "true" length...)

3.1.2.1 River length and gradient

The surface water network is dynamic (c.f. 1.1.3 Time variance in flow). Therefore, the length of a river can vary. When a starting point, a course and an endpoint is ascertained, then the length of a river segment for this certain state of the network can be calculated based on its geometry (or if necessary, based on the measures that are combined with the segment). All length calculations are strongly influenced by scale (c.f. 1.2.1). Thus it is only legitimate to compare length values of geometries created on the same scale and with the same method. This will therefore have to be a point for further investigations concerning a European-wide dataset of rivers where different countries contribute their data.

The length itself is a variable of the network edges' geometry. Involvement of the topological part of the network is not necessary to calculate the unique edges lengths either in 2D or 3D. In some cases when geometry is not seen as exact enough, the length values will be calibrated through survey points along the river and used as measures, this being one way to deal with different scales.

The edge lengths can be used as costs of a network analysis when calculating special length variables, which are:

- The **Path Length**: The distance from the downstream end of a segment to the network termination point.
- **Longest Flow Path**: The path with the greatest length from a certain point upstream.
- **Arbolate Sum**: The sum of lengths of all segments that flow to the downstream end of a segment. This is an aggregation parameter that can be refined with flow data to hold the volume of water in a (sub-)system at a certain time step.

The length values are often related to other values that are seen as important for hydrologic research:

- **River Gradient**: Length of a stretch (can be segment length, path length or other) related to the difference in height

$$G_i = \frac{L_i}{\Delta H_i}$$

... G_i gradient of the stretch i

... L_i length of the stretch i

... ΔH_i difference in height between the starting point and the end point

The gradient can vary in the river, and should be calculable dynamically for arbitrary parts of rivers. The gradient is a variable that has a big influence on the flow velocity and because of this on the flood hydrograph.

- **Network Density**: The arbolate sum related to the catchment area.

$$D_j = \frac{\sum_{i=1}^{n} L_{ij}}{A_j}$$

... D_i density of rivers in Catchment i

... L_{ij} length of the stretch i in catchment j

... A_j area of catchment j

3.1.2.2 Shape of the network

The shape of a network is a combination of topological and geometrical parameters. The angle of the confluences is one parameter of the networks' shape description that can be calculated based on the networks topology and geometry. Another parameter would be the network's overall orientation.

3.1.3 Variables of the Catchment

"*Morphometric characteristics of drainage basins provide a means for describing the hydrological behaviour of a basin.*"*([BÁRDOSSY 2002], p. 931)*

Many articles have been written about meaningful catchment variables (or drainage basin variables as they are often called). Gardiner and Park [GARDINER 1978] created a thorough review of ideas about catchment up to the publication time of their article. The set of problems concerning the meaningfulness of variables and the implications of scale (c.f. also 1.2.1) are valid up to this day and are re-assessed in later articles (e.g. [BÁRDOSSY 2002]).

3.1.3.1 Area

"*In hydrology, the drainage area of a basin (...) is needed whenever a member of the water balance equation is to be quantified in volume units for the basin as a whole, or for parts of it. Thus, uncertainties in the basin area will lead to uncertainties of the same order of magnitude in the water balance calculations.*" *([BÁRDOSSY 2002],* p. 933)

3.1.3.2 Perimeter

Perimeter P is a problematic parameter, like all other length variables with a fractal dimension. Related with the area the perimeter serves as a variable that can be used to give some information about the shape of the catchment.

> "*Basin perimeters have ever-increasing length as resolution of view increases, and as a result, the perimeter measurement employed in calculation of a number of morphometric indices has no unique value. If there is no explicit compensation for this in experimental design, there is a real potential for misinterpretation of results, particularly in studies comparing basins of different*

size or data sets of basin characteristics obtained from different map sources." ([BREYER 1991], p.156)

Breyer and Snow further state that when using the perimeter in shape analysis, it should be measured using a coarse resolution that is proportional to the catchments size. Using the perimeter of the catchments should therefore always happen with great caution when there are different catchment data sources in use.

3.1.3.3 Length of the catchment

There are different length variables in catchment research. One very common is the air-line distance from the confluence point (that is at the lower end of the catchment) to the point farthest away on the catchment's boundary. The value usually works very well for small catchments, but larger catchments do not always have a straight teardrop form. They tend to have bends in their course. This means that the air-line distance line will not run along the valley. It could even be possible that the line crosses the catchment boundary. Therefore, usually other values are used - like the valley length of the longest tributary where the valley is extended to the catchment boundary.

3.1.3.4 Shape of the catchment

3.1.3.4.1 Compactness

Compactness of shapes is calculated as the area of the shape in relation to the area of a circle (as the most compact shape). As early as in 1914 Gravelius published a form parameter, which he called "Entwicklung der Wasserscheide" (expansion of the watershed):

"*Ist U km die Länge der Wasserscheide eines Flußgebietes von der Größe f qkm, so wird man sie vergleichen mit dem kleinsten Umfang, den diese gegebene Fläche f überhaupt haben könnte. Nun ist der Kreis diejenige Figur, welche bei gegebenem Flächeninhalt f den kleinsten Umfang u besitzt. Aus* $\mathrm{f} = \pi \mathrm{r}^2$ *und* $\mathrm{u} = 2\pi \mathrm{r}$ *findet man* $\mathrm{u} = 2\sqrt{\pi \mathrm{f}}$.

Dieses u kann man also leicht für jedes gegebene Flußgebiet f finden; und es ist dann das Verhältnis U:u=e welches interessiert und welches als Entwicklung der Wasserscheide bezeichnet wird."[8]([GRAVELIUS 1914], p. 14)

This parameter is now mostly known as Compactness C ([GREGORY 1973], p. 51, [BÁRDOSSY 2002], p. 932):

$$C = \frac{P}{2\sqrt{\pi A}}$$

P ... Perimeter of the catchment

A ... Area of the catchment

If C = 1, then the catchment would be completely round. The larger C is, the larger is the perimeter compared to the catchments size. This can be because the catchment is very long and narrow (or broad and short) but also because the catchment is deeply jagged.

3.1.3.4.2 Basin Circularity

A very similar factor was defined in 1953 by Miller. The Basin circularity c is the ratio of basin area to the area of circle, having the same perimeter as the basin. ([BÁRDOSSY 2002], p. 932, [GREGORY 1973], p. 51)

$$c = \frac{4\pi A}{P^2} = \frac{1}{C^2}$$

A ... Area of the catchment

P ... Perimeter of the catchment

C ... Compactness

[8] "If U (km) is the length of the watershed of a river system of size f (km^2), one will compare it with the smallest perimeter this given area f could have. Now the circle is the one figure that has the smallest perimeter u of a given area f. From $f = \pi r^2$ and $u = 2\pi r$ one finds $u = 2\sqrt{\pi f}$.
This u can easily be found for any given river basin f, and it is then the ratio U: u = e which is interesting and which is called the expansion of the watershed."

3.1.3.4.3 Form Factor

In 1932 Horton created the Form Factor *F* ([BÁRDOSSY 2002], p. 932, [GREGORY 1973], p. 51), which is a parameter where he related the catchments' area with the square of the catchments' length. Its advantage over the compactness parameter and the basin circularity is, that it is not influenced by a fractal dimension, at least when the length of the catchment is calculated as a straight line from the catchments mouth to the most distant point on the watershed. Horton criticized the compactness ratio for not considering the point of the stream outlet, which also applies to the basin circularity.

$$F = \frac{A}{L^2}$$

A ... Area of the catchment

L ... Length of the catchment

If $F = 1$ then the catchment has a quadratic form. The smaller F becomes, the longer or more jagged the catchment is. The main discussion point is how the length of the catchment is to be calculated.

3.1.3.4.4 Basin Elongation

In 1956 Schumm created the basin elongation, a ratio of the diameter of a circle with the same area as the basin to the basin length.

$$E = \frac{2\sqrt{A}}{L\sqrt{\pi}}$$

A ... Area of the catchment

L ... Length of the catchment

3.1.3.4.5 Lemniscate Ratio

Because the catchment shape is nearly never circular but most times more teardrop like, Chorley compares the basin of such a shape with the Lemniscate Ratio *K*

([BÁRDOSSY 2002], p. 932, [CHORLEY 1959], p. 341). Before applying K to river basins, Chorley used it to describe the shape of Drumlins[9].

$$K = \frac{\pi L^2}{4A}$$

A ... Area of the catchment

L ... Length of the catchment

3.2 Surface water network/catchment uses in biology and ecology

3.2.1 Hydromorphology

Hydromorphology is a very recent term that has been created for the European Union Water

Framework. It originates from soil science but has been given a new and specific meaning, which is “the hydrological and geomorphological elements and processes of water body systems.”

"*The field of hydromorphology is introduced to respond to the myriad of scientific and engineering challenges created by the wide range of natural and anthropogenic influences that have literally “morphed” the hydrologic cycle at all spatial and temporal scales.*" ([VOGEL 2011], p. 148)

"*Whilst traditionally poorly quantified, the link between physical habitat and ecological response in rivers is widely recognised, and is currently rising up legislative and policy agendas. In Europe, this is reflected in the Water Framework Directive which dictates that ‘hydromorphological’ condition of water bodies should be capable of supporting ‘Good Ecological Status’. Methods are developed that integrate river system hydrology, geomorphology and ecology (and the complex interplay between these three variables).*" *([ORR 2008], p. 32)*

[9] A Drumlin is a small hill created by glacial ice impacting glacial sediments. It has the form of a turned spoon without handle.

Hydromorphology could therefore be described as geomorphology with a strong intertwining with ecology and biology. In ANNEX V the WFD lists the hydromorphological elements that are seen as important quality elements for the classification of ecological status - the *river continuity*[10] is directly connected with the structure of the river network but includes other information like the behaviour of species. Others of the listed elements (*hydrological regime, quantity of water flow*) can use the topological and geometrical information represented in the digital surface water network.

3.3 Surface Water network and their corresponding catchments used for WFD questions

The WFD (but also other directives of the EC dealing with surface water, e.g. the EU Floods Directive, the EU Flora-Fauna-Habitats and Birds Directives) has a strong reporting component. If elements reported have a spatial quality, then they should be presented in their spatial context. Thus, all the administrative bodies connected with the preparation of the data for reporting about the conditions of surface waters, get (more or less intensive) in contact with the surface waters GIS datasets (thus also with the network):

> *1 SURFACE WATERS*
> *1.1. Characterisation of surface water body types*
> *Member States shall identify the location and boundaries of bodies of surface water and shall carry out an initial characterisation of all such bodies in accordance with the following methodology. Member States may group surface water bodies together for the purposes of this initial characterisation.*
> *(i) The surface water bodies within the river basin district shall be identified as falling within either one of the following surface water categories – rivers, lakes, transitional waters or coastal waters – or as artificial surface water bodies or heavily modified surface water bodies.*
> *(ii) For each surface water category, the relevant surface water bodies within the river basin district shall be differentiated according to type. These types are those defined using either "system A" or "system B" identified in section 1.2.*
> *(iii) If system A is used, the surface water bodies within the river basin district shall first be differentiated by the relevant ecoregions in accordance with the geographical areas identified in section 1.2 and shown on the relevant map in Annex XI. The water bodies within each ecoregion shall then be differentiated by surface water body types according to the descriptors set out in the tables for system A.*

[10] River continuity can be defined for different purposes. The ecological view of continuity is closely connected with the ability of different species to migrate along the river.

(iv) If system B is used, Member States must achieve at least the same degree of differentiation as would be achieved using system A. Accordingly, the surface water bodies within the river basin district shall be differentiated into types using the values for the obligatory descriptors and such optional descriptors, or combinations of descriptors, as are required to ensure that type specific biological reference conditions can be reliably derived.
(v) For artificial and heavily modified surface water bodies the differentiation shall be undertaken in accordance with the descriptors for whichever of the surface water categories most closely resembles the heavily modified or artificial water body concerned.
(vi) Member States shall submit to the Commission a map or maps (in a GIS format) of the geographical location of the types consistent with the degree of differentiation required under system A. [WFD 2000]

Several working groups were/are on the way to achieve the tasks set by the WFD. EU working groups and international river basin working groups created guidance papers for the national level to coordinate how the tasks have to be fulfilled. National working groups and Federal states working groups prepare inputs for guidance papers and consolidate the data.

The WFD-guidance papers are at the moment kept at the CIRCABC (Communication and Information Resource Centre for Administrations, Businesses and Citizens)-server.

- No. 9. Implementing the Geographical Information System Elements (GIS) of the Water Framework Directive
- No. 10. Rivers and Lakes - Typology, Reference Conditions and Classification Systems
- No. 22. Updated WISE GIS guidance

These sources give a lot of detailed information on how to deal with surface waters. With regard to the surface waters, it is eye-catching that most of the data is presented on **different levels of detail** than it is usually collected. Thus, ways for the aggregation of data (e.g. from single catchment level to the level of grouped catchments (100 km^2, 500 km^2)) have to be planned and kept in mind.

Another thing that attracts attention is the huge amount of data that has to be collected and stored in the spatial context. This cannot happen directly within the digital network - because the network would be cut into multiple small pieces by doing so. The suggested way by the GIS-Guidance documents to deal with this is the use of **dynamic**

segmentation. The digital network will not only have its geometrical and topological information, but it will also be combined with routing information that makes it possible to reference a lot of different data onto the network.

3.4 Surface water networks in river engineering and navigation

River engineering attempts to understand fluvial geomorphology, implement a physical alteration, and maintain public safety. Networks with a medium to high resolution can therefore help to gain some insight in the headwaters of the location to be engineered. The accumulation of catchment areas above a certain point can for example be used to calculate worst-case scenarios for flooding or to give basic information for any kind of regulation work. This usually is only possible if the network and catchment model is combined with a high resolution DEM and additional data (profiles, rainfall, anthropogenic risk locations...).

Engineering works to increase the navigability of rivers only make sense when they are undertaken in large rivers with a moderate fall and a high enough discharge in the dry seasons. The surface water networks can therefore not only be used to hold data on the navigability of rivers, which can be queried accordingly, but they can, in combination with other data (river widths, river depths, river profiles, land use, DEM, ...), help to estimate the amount of engineering that has to be done (channelization, ...) to reach the goals.

3.5 Conclusion on the use of digital surface water networks

From the different uses of digital surface water networks listed above, the main following uses can be derived:

Ordering and coding: the networks' topology is used to create river ordering, which leads to hierarchies in the data set that are used for data comparison as well as administrative purposes

Locating (tracing): the networks' topology is used to easily find locations and segments that influence (because they are above) a certain point in the network or that are influenced by (because they are beneath) a certain point in the network.

Getting information about the network and its elements: network information can be based on topology or geometry.

Getting information on continuity: The network geometry and topology, along with information from biology and hydromorpholgy, are used to calculate continuity of the river for several purposes (navigation, fish migration, other species migration, and flow).

Accumulating: Along the network data is accumulated. This can be area (catchment data), length, flow and dynamically segmented data (counts ...). The continuity of the river may have to be kept in mind, as well as the type of connectivity between catchments and rivers.

Relating catchment data to rivers and river data to catchments

Relate research data to rivers: How to use a network based on natural conditions and apply data to it, (e.g. by dynamic segmentation) and still be able to query that network in sufficient time.

4 Algorithms for the use of digital surface water networks

4.1 Introduction to Algorithms

4.1.1 Basics

(c.f. [CORMEN 2009, DUMKE 2001, KNUTH 1997, KUNZINGER 2002, OTTMANN 1998, OTTMANN 2002, SAAKE 2004, SEDGEWICK 2011, VORNBERGER 2008])

The term "algorithm" goes back to the period around 825: Muhammad Ibn Musa **Al-Khwarizmi** wrote the book "Kitab al-jabr wa'l-muqabala"("Rules of restoring and equating") [KNUTH 1997 S.1]. His name and the Greek "arithmos" for number led to the term algorithm. Algorithms exist already since ancient times. Just think of Euclid's (around 300 BC) algorithm for finding the greatest common divisor gcd in its 7th Book of Elements - an explanation of the individual steps of computing the greatest common divisor of two numbers. This is already evidence that the process is more or less independent of the computer, which should be used and also independent of the programming language. An algorithm is the description of all steps to be executed in order for a given problem to be solved.

Ein Algorithmus ist eine präzise (d.h. in einer festgelegten Sprache abgefasste) endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte. ([SAAKE 2004] S. 16)[11]

It is often in the textbooks for algorithms, however, that the term is used only in connections with computers[12].

Informally, an algorithm is any well-defined computational procedure that

[11] An algorithm is a precise (that means composed in a determined language) finite description of a general procedure by using executable elementary (processing) steps.

[12] It should be noted that each person is confronted with algorithms outside the computerized world, be it by a cooking recipe or by a manual. The intuitive grasp of the concept "algorithm" is therefore usually correct, because the concept is familiar.

takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. ([CORMEN 2001] S.5)

Algorithms should have the following qualities:

- **Correctness**: An algorithm terminates with the correct solution in all possible input cases.
- **Determinacy**: Different cycles of an algorithm have to produce the same result if the instance and conditions stay the same.
- **Definiteness:** An algorithm specifies the sequence of events. This means that for every point in the procedure there is only one possibility to go further. This includes details of each step and including how to handle errors
- **Interpretability:** The individual steps must consist of clearly interpretable instructions. To avoid language barriers in this steps a formalized language or pseudocode is used where the semantic of the step is clearly described. The implementation in a certain programming language would follow this description.
- **Feasibility:** Each step of the procedure must be executable. Wherein the execution of the individual steps may happen sequentially or parallel and can be controlled by conditions.
- **Finiteness:** An algorithm terminates after a finite numbers of steps.
- **Efficiency:** The implementation of the algorithm has to happen to such an extent that time (running time) and cost (e.g. storage space, processing power ...) issues have a well-defined scope. When a large program is to be created, there should advance a large portion of the work in the formulation of the problem. Usually the problem is split up into small sub-problems, which can then be implemented. Since for solution to a problem there often are several approaches, the best algorithm for a particular problem has to be chosen. This selection can be a complicated process, perhaps involving sophisticated mathematical analysis. The selection of the algorithm is essential for the running time of a program and has a greater impact in the same, as the computational implementation.

A good algorithm should be the result of optimizing these qualities. In general it will be efficiency where much of the time will be invested, at least when the problem is big enough, because we mostly are interested in questions like "How long will it take?" or "Will my hard drive be big enough?" or "Why does the process run out of memory?".

4.1.1.1 Time efficiency

In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem ([CORMEN 2001], p. 5)

Although the running time of an algorithm increases when the instance increases, the rate of increase can differ between two algorithms (even if the outcome may be the same). Table 9 shows different increase functions. Taken the rounded up integers of the shown values, the numbers give the necessary processing steps for an instance n when the running time of an algorithm follows the certain increase function.

Thus for big datasets it becomes very important that the growth of increase functions is as small as possible. For big problem sizes it can be stated that those algorithms are the fastest which have the slowest growing increase functions.

instance	**increase functions**			
n	**$\log_2(n)$**	**$n*\log_2(n)$**	**n^2**	**n^3**
1	0.000	0.000	1	1
10	3.322	33.219	100	1,000
100	6.644	664.386	10,000	1,000,000
1000	9.966	9,965.784	1,000,000	1,000,000,000
10000	13.288	13,2877.124	100,000,000	1,000,000,000,000
100000	16.610	1,660,964.047	10,000,000,000	1,000,000,000,000,000
1000000	19.932	19,931,568.569	1,000,000,000,000	1,000,000,000,000,000,000

Table 9 Comparison of different functions

Figure 4–1 shows graphs of some increase functions. Although for the modest input of only 100 items, the differences between the functions can clearly be seen.

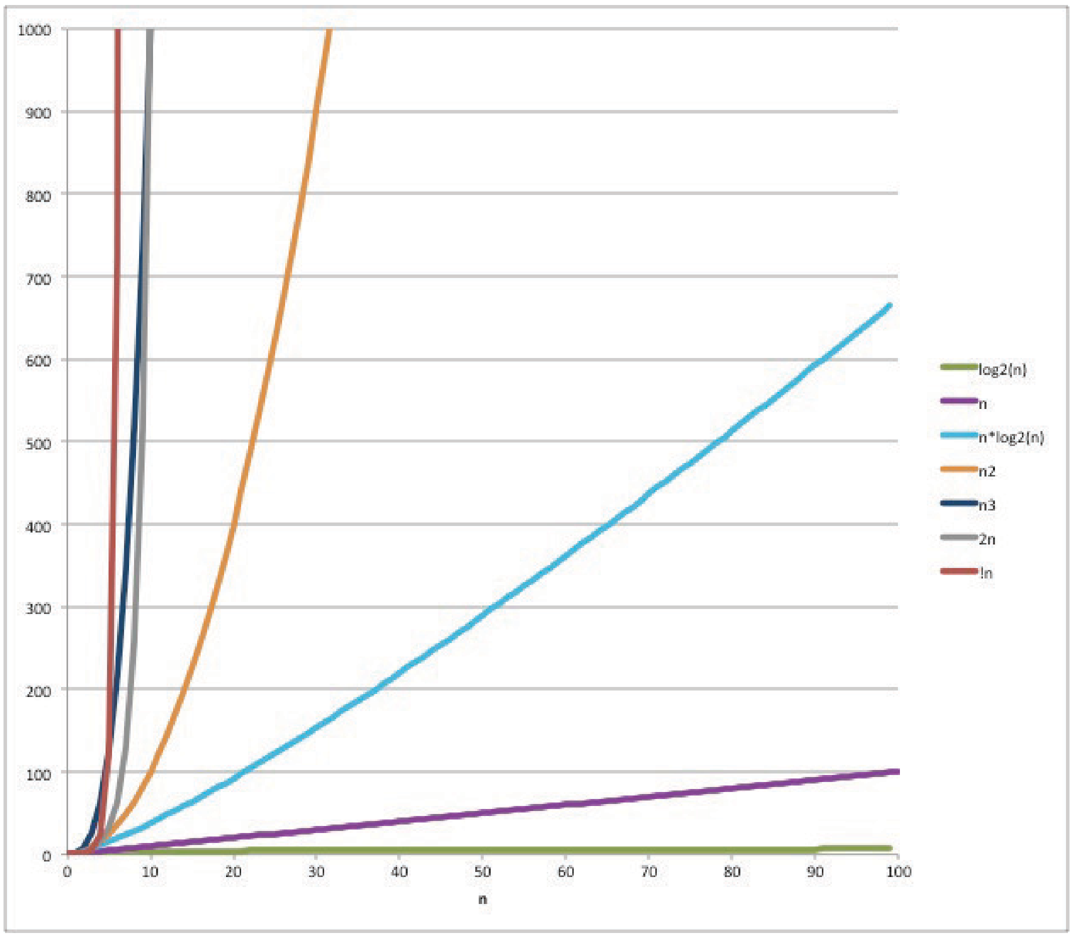


Figure 4–1 Graphs of different increase functions

To make the implications even more clear, Table 10 shows how long the processing of an input of size n would take with a certain increase function, when we assume that the computer performs one trillion[13] (10^{12}) instructions per second[14]. Some cells are not even calculable any more with normal spread sheet programs - the red coloured cells are all above 1 year. Increase functions like n^3, 2^n and n! hence should only be "accepted" for small input sizes, otherwise the calculation may even take more than a lifetime. [15]

[13] This is noted in short scale and would be a billion in long scale (e.g. used in Austria)

[14] Intel Polaris can perform up to 1.82 trillion instructions per second; the Intel Core i7 (Dual Core) Arrandale of my 2010 notebook, which I used to write this thesis with, has a maximum performance of 42170 MIPS (million instructions per second)

[15] To compare this values with the running time of my notebook with the Intel Core i7 (Dual Core) Arrandale: n=100.000 with n3-algorithm would have a running time of nearly 7 days, n=1 mio with n3 would take 275 years

n —— f(n)	10	20	30	40	50	100	1000	10000	100000	1000000
$\lg_2 n$	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000
n	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000001
$n \lg_2 n$	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000002	🟩 0.000020
n^2	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000001	🟩 0.000100	🟩 0.010000	🟩 1.000000
n^3	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000000	🟩 0.000001	🟩 0.001000	🟩 1.000000	🟦 16.666667	🟪 11.574074
2^n	🟩 0.000000	🟩 0.000001	🟩 0.001074	🟩 1.099512	🟦 18.764998	🟥 4.0E+10	🟥 3.4E+281	🟥 $>10^{300}$	🟥 $>10^{300}$	🟥 $>10^{300}$
n!	🟩 0.000004	🟪 28.158588	🟥 8.4E+12	🟥 2.6E+28	🟥 9.6E+44	🟥 3.0E+138	🟥 $>10^{300}$	🟥 $>10^{300}$	🟥 $>10^{300}$	🟥 $>10^{300}$

🟩	seconds
🟦	minutes
🟪	days
🟥	years (with 365 days each)

Table 10 Times for processing of an input of size n under the assumption that the computer performs one trillion (1,000,000,000,000) high level instructions per second (c.f. [KLEINBERG 2006], p34)

The running time T(n) of algorithms is in most cases also dependent on the quality of the input data (e.g. if the data is to be sorted with a sorting procedure then it takes usually less time, if the data already has a preliminary sorting, compared to a data set that is completely random). This means that two instances of the same size put into one and the same algorithm can show different running times. It is therefore not expected that an algorithm under different conditions (even with the same problem size) works equally well in every case. With mathematical means the worst case (the longest running time that can occur - the upper bound of running times), best case (the shortest running time - the lower bound) and possibly the average case (the running time one finds usually with that kind of data) can be decided.

worst case:	$T_{worstcase}(n)=\max(T_i(n))$	where $i \in$ {possible running times for algorithm}
best case:	$T_{bestcase}(n)=\min(T_i(n))$	where $i \in$ {possible running times for algorithm}
average case:	$T_{average}(n) = \sum(T_i(n)*p_{i)}$	where $i \in$ {possible running times for algorithm}
	and p is a probability value	

4.1.1.2 Asymptotic notation

"The order of growth of the running time of an algorithm (...) gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms." ([CORMEN 2001), p. 43)

Sometimes the precise running time of an algorithm can be determined, but it is not always possible and sometimes the extra effort is not necessary and not productive (this is because the steps in a high level language can be counted, but those steps are split into a number of more primitive steps when compiled and their actual number is dependent on the architecture used). Therefore, usually the **asymptotic efficiency** of an algorithm is studied. This means one tries to find a function to which the real runtime polynomial approximates when using big input sizes.
Donald E. Knuth compiled in 1976 [KNUTH 1976] a list of definitions (O Omicron, Ω Omega, Θ Theta) for the asymptotic analysis that are still valid and used today:

O(f(n)) denotes the set of all g(n) such that there exist positive constants c and n0 with $g(n) \leq cf(n)$ for all $n \geq n0$ [16]
Ω(f(n)) denotes the set of all g(n) such that there exist positive constants c and n0 with $g(n) \geq cf(n)$ for all $n \geq n0$.
Θ(f(n) denotes the set of all g(n) such that there exist positive constants c, c', and n0 with $cf(n) \leq g(n) \leq c'f(n)$ for all $n \geq n0$.
([KNUTH, 1976], p. 19)

g(n) can be seen as the function describing the actual running time *T(n)*.
These three numbers he *[KNUTH 1976]* described further verbally as,

O "*order at most f(n)*"
- "fewer than or the same as" iterations
- upper bound of f(n) -> there will be at most so many iterations

Ω "*order at least f(n)*" -
- "more than or the same as" iterations
- lower bound of f(n) -> there will be at least so many iterations

Θ "*order exactly f(n)*"
- "the same as" iterations.
- tight bound of f(n)

[16] This means that g grows not faster than f

There exist stricter versions of these notations:

- little o denotes "fewer than" iterations ($T(n)=o(f(n))$ if and only if for all constants $c>0$, $\exists$ a constant n_0 such that $T(n) \leq c*f(n)\ \forall n \geq n_0$),
- little ϖ denotes "more than" iterations,

but they are not as widely used as the big O Omicron, big Ω Omega, and big Θ Theta numbers.

To put this into practice the first step is to find the dominating term in the running time function and omit multiplicative constants[17], e.g. $T=3n^2+5n$ is therefore $O(n^2)$.

4.1.1.3 Typical upper bounds for running times

The following overview was compiled using ([KLEINBERG 2006, SEDGEWICK 2011, VORNBERGER 2009])

O(1) Time (constant time) is found with algorithms that do not depend on the input size. Thus every time the algorithm is executed the number of steps will be the same.

```
public static int countN(int[] n) {
return n.length;
}
```

In the above example the input integer array n can have a different number of items per run, but the number of steps will always be 1.

O(log n) Time (logarithmic time) [18] is found with algorithms in which for every growth of n (the input size) the problem size gets reduced to $\log_2 n$. Such algorithms are those in which the problem is divided into two halves and one of the two is selected. The question is, how many times can n be divided by 2 until the result is = 1?

$$\mathbf{n = 2^x \Longleftrightarrow x = \log_2 n}$$

An example for $O(\log_2 n)$ is binary search (e.g. used in StrahlerLanfear.java).

O(n) Time (linear time) is found with algorithms that have a running time being a constant factor times the number of input data (e.g. find river segments longer than 1000 m; compute the maximum of n numbers, merge two sorted lists).

[17] Exponential (e.g. 4^n (4^n dominates 3^n, 3^n dominates 2^n)) dominates polynomial. If it is polynomial find the highest exponent (e.g. n^3 dominates n^2). Logarithms are dominated by polynomials and exponentials.
[18] Usually the $\log_2$ is used.

```
public static int countN(int[] n) {
  for(int i = 0; i<n.length;i++){
    System.out.println("Value of n at " + i + " is " + n[i]);
  }
}
```
In the above example the number of steps is the same as the number of items in the input array.

O(n log n) Time (linearithmic time) is typical for algorithms that divide the input in two and work recursively and use linear time to merge the outcome - thus typical divide and conquer algorithms. (e.g. Mergesort)

Up to here are the time orders we strive to achieve, because they do not grow disproportionally high when the input grows.

O(n^2) Time (quadratic time) is typical for algorithms that work with pairs in a brute-force way (e.g. finding the closest pair of points). These algorithms usually contain nested loops on the input data.

```
for(i=0; i<n;i++){
  for(j=0;j<n;j++){
    //Here follows what to do in each loop
    //with the n cases of input data
    //using the indices i and j
  }
}
```
Some of this algorithms with quadratic order can be rewritten to O(n log n) algorithms using a more clever way than the brute-force algorithm does (e.g. in sorting the points based on their coordinates and only compare those which are close enough. Doing this in a recursive way leads to O(n log n)).

O(n3) Time (cubic time) is for example typical for algorithms that do schoolbook matrix multiplication, or Gauss-Jordan-elimination for matrix inversion. These algorithms usually contain three nested loops of the following sort:

```
for(i=0; i<n;i++){
  for(j=0;j<n;j++){
        for(k=0;k<n;k++){
        //Here follows what to do in each loop
        //with the n cases of input data
        //using the indices i, j, and k
        }
      }
   }
```
If there are even more nested loops over all the input data, then the algorithm has **O(n^k)** where k is the number of nested loops.

O(2^n) Time (exponential time) denotes algorithms whose growth will double with each additional element in the input data sets (e.g. find the maximum size independent set in a graph).

O(n!) Time (factorial time) denotes algorithms whose growth will rise with i for the i-th additional element.

$$n! = 1*2*3*4*5*.......*n$$

4.1.2 The design of algorithms

The ways to design algorithms already fill many books. Searching for the right strategy can be time consuming and many times personal preferences for one (known) strategy will prevail. Given a problem to be solved there are different strategies to create suiting algorithms.

Preiss [PREISS 1999] states the following five strategies:

1. **Direct solution strategies**

Brute force algorithms are straightforward approaches directly based on the problem statement and definitions. They proceed in a simple and very obvious way. Their advantage lies in their simplicity. They are easy to implement, but they rarely are time efficient, because they usually do much more work than more clever algorithms. For small instances they have usually good enough running times and it may not be worth finding a better algorithm. But mostly they are just the yardstick for better algorithms, some that are cleverer and therefore more efficient.

Greedy algorithms build up a solution step by step, always choosing the next step in a way that for every step it offers the most immediate benefit (c.f. [DASGUPTA 2008, KLEINBERG 2006]). Once made decisions are never reconsidered. The advantage of greedy algorithms is, that they are faster than brute force algorithms, but that it is much harder to prove their correctness. Some greedy algorithms only prove to be right given certain preconditions (e.g. Dijkstra's shortest path algorithm is greedy by weights and has the precondition that all weights have to be positive).

2. **Backtracking strategies**

Simple backtracking algorithms try to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between alternatives to the next component of the solution, it simply tries all options in a depth-first manner, when the solution is not found in one option (i.e. a branch that starts from a node in the solution space) it tracks back and tries the next one.
Branch-and-Bound algorithms are similar to the simple backtracking algorithms but use a breadth-first approach. The nodes of the solution space are put in a queue and are processed in a first-in-first-out (FIFO) order. If a cost criterion is available, it

will be used to decide which node in the queue (the one with the best cost) to expand next (this is the "branch" part of the algorithm). The costs will also be used to discard those nodes from the queue that are too expensive (this is the "bound" part of the algorithm) - thus the solution space becomes smaller and there are less solutions to be considered. (c.f. [PREISS 1999]).

3. Top-down solution strategies

Divide-and-Conquer algorithms break the input into ever smaller non overlapping parts (usually two) of the same size (that's the divide), solves the problem in each part recursively and combines the solutions to the sub-problems into a solution for the total problem (that's the conquer part). The divide part usually runs in c*log n time. In many cases it is possible to combine the data in c*n steps thus getting an overall running time order of O(n log n). A very well-known algorithm using Divide-and-Conquer is the merge-sort algorithm.

4. Bottom-up solution strategies

Dynamic-programming is a strategy that breaks the problem into smaller and smaller overlapping sub-problems, which are then solved. The small sub-problems solutions are then combined into the solutions to larger and larger sub-problems like in the divide-and-conquer approach. In contrast to divide-and conquer, the sub-problems overlap which means that sub-problems share sub-problems (c.f. [CORMEN 2009], p. 359). Thus the challenge in dynamic-programming *is to understand the set of sub-problems and how the sub-problems depend on one another* ([CORMEN 2009], p. 367). Dynamic-programming avoids to calculate the same values over and over again. It first recursively defines the value of an optimal solution and then computes the value in a bottom up way. All necessary sub-solutions are memorized (c.f. [DASGUPTA 2008, KLEINBERG 2006]). An example for a dynamic-programming algorithm is the Bellman-Ford-Moore algorithm for computing single-source shortest paths in weighted directed graphs (c.f. [BANG-JENSEN 2007], p. 55).

5. **Randomized strategies**

Based on some probabilistic concept the algorithm decides on what step to take next. "*Thus the role of randomization is purely internal to the algorithm [...]*" *(*[KLEINBERG 2006], p.705).

Surprisingly – almost paradoxically – some of the fastest and most clever algorithms we have rely on chance: at specified steps they proceed according to the outcomes of random coin tosses. These randomized algorithms are often very simple and elegant, and their output is correct with high probability. ([DASGUPTA 2008] p. 38)

For this thesis different strategies were tried. In a first step the questions out of the "Conclusion on the use of digital surface water networks " (see section 3.5) were split up in manageable sub-problems. For those a solution was created, if nothing better was available already, in a brute-force-way. This algorithm was then tested. The tests with the brute-force algorithm usually revealed a lot of "special cases" that had to be considered.

Most of the problems in networks demand to visit every arc (or node or catchment) at least once. For many of the algorithms eventually the depth first search approach (DFS, backtracking) has proved most useful and should therefore be stated in more detail in the following. The idea of DFS is not a new one. Since Robert Tarjan's article [TARJAN 1972] it is a widely used concept of traversing graphs.

There are different ways to implement DFS, either with nested recursions or with a stack (this is a data sequence where the last data put in, is the first data to take out again - LIFO). The recursive way is very elegant, but with very deep trees it can happen that the computer's memory will not suffice, because the recursions are nested in each other and are kept in memory. In this case the implementation would have to be made with an iterative approach using a stack.

In the following examples the two ways to implement a DFS for directed graphs are shown as code stumps in a Java-like pseudocode - the vertices are only visited, but nothing happens - the effective code for calculations would have to be put at the places where you find a comment "//do something with node".

// ... marks comments - this would be lines that are not executed

void ... means that the method does not give anything back to the caller

Node ... it is assumed that a class Node exists

incomingNodes(Node node) ... it is assumed that a method incomingNodes exists which has a node as parameter and gives back a collection of the upper adjacent Nodes of that node

The DFS implementation with nested recursion for Nodes:

```
public void DFSNode(Node node) {//method DFSNode
  //do something with node
  //get the nodes upstream neighbours
  node.visited=true
  if(incomingNodes(node).isEmpty()){
  // do something - this is a most upper Node
  } else { // the node has upstream neighbours
  // do something
  // get upstream neighbours and apply
  foreach(inNode : incomingNodes(node).inNodes){
    if(node.visited==false){
    Node upperNode = inNode;
      DFSNode(upperNode); // recursive Call
    }
  }
  }
  //do something with node
}
```

An iterative DFS implementation for Nodes using a stack:

```
public void DFSNodeIterative(Node node) {
  //mark the starting node visited
  node.visited=true
  // do something with the first node
  // make new empty stack of type Node
  Stack<Node> nodeStack = new Stack<Node>();
  // push all incoming nodes onto nodeStack
  nodeStack.push(incomingNodes(node))
  // do as long nodeStack is not Empty
  while(!nodeStack.isEmpty()) {
   Node n = nodeStack.pop() //pop node from nodeStack
   // do something with the node
   // do for each upstream neighbour
   foreach(inNode : incomingNodes(n).inNodes){
      if(inNode.visited==false){
      inNode.visited=true
      nodeStack.push(inNode)
      }
   }
   }
}
```

An implementation to run through the edges would look very similar (recursive approach) - the parameter edge is the lowest edge in the network:

```
public void DFSEdge(Edge edge) {//method DFSNode
   //do something with edge
   edge.visited=true
   //get the from node of the edge
   fromNode = edge.getFromNode()
   //get the fromNode's incoming edges
   upperEdges = fromNode.getIncomingEdges()
   if(upperEdges.isEmpty()){
   // do something - this is a most upper Edge
   } else { // the edge has upstream neighbours
   // do something
   // get upstream neighbours and apply
   foreach(inEdge : upperEdges){
      if(node.visited==false){
      Edge upperEdge = inEdge;
         DFSEdge(upperEdge); // recursive Call
      }
   }
   }
}
```

4.1.3 Quality management for the algorithms

4.1.3.1 Verify the correctness of Algorithms

To verify the correctness of an algorithm, the problem has to be understood. An algorithm's correctness can only be assessed in reference to the problem that has to be solved. A problem is a collection of problem instances. An algorithm is correct for a problem instance if it produces the correct answer for this instance. An algorithm is correct for a problem if it is correct for all instances of the problem (e.g. a shortest path algorithm is correct if it always (this means for every graph and for every two arbitrarily chosen start and end point) computes the right answer). In some algorithms additional preconditions concerning the instance have to be stated to be able to guarantee correctness.

There are two strategies to verify if an algorithm really solves the problem:

1. Testing: The algorithm is tested with different input data (different instances of the problem). This is a simple approach. Its disadvantage lies in the fact that testing can never cover all possible instances of input data. The arising question therefore is: "How much testing is enough?" With testing only it cannot really be guaranteed that the algorithm is correct.
2. Proving: With a formal analysis it has to be proved that the algorithm is correct for every possible instance. This guarantees the correctness of the algorithm. The disadvantage of proving is the difficulty to find a proof. For very complex algorithms the algorithm may have to be split into several sub problems, which then have to be proved separately.

To prove the correctness one has to
... identify the preconditions. What kind of data can be used with the algorithm? (E.g. positive values only, values not bigger than X ...)
...identify the postconditions. What must the output be like?
...prove that starting from the preconditions and executing the steps in the algorithm the postconditions are achieved.

To prove recursive algorithms one can either use proof by induction or proof by contradiction.

To prove the correctness of iterative algorithms a proof with a loop invariant can be made (which is a lot like induction - first prove a base case, then an inductive step): A loop invariant is a property that is valid at the beginning (initialization), before every loop iteration (maintenance), and at loop termination.

4.1.3.2 Analyse implementation

"*Beware of bugs in the above code; I have only proved it correct, not tried it.*" [Donald E. Knuth (http://www-cs-faculty.stanford.edu/~knuth/faq.html)]

Algorithm idea and implementation are different things. The only way to really test an algorithm is via its implementation. Even though in this thesis the algorithms are always shown in their implemented Java code form, this may not be the only way to implement them. The implementation of an algorithm follows from the algorithm's idea and sometimes gives good hints how to make the algorithm better and avoid errors (esp. with real data where numerous unexpected cases can occur). The downside of an implementation is that it can introduce new errors not based on the algorithm. Thus also the different parts of the implementation - e.g. data input, data output, method calls, calculations have to be tested thoroughly. To do this it always has to be stated what should be the outcome of this part of the implementation. With dummy data (for ideal, normal and special cases) many errors can be detected before the implementation is tested on big real world data sets. During the work for this thesis a collection of dummy data was created for river edges and junctions that helped to test if the implementations produced the right output. Figure 4–2 shows an example for testing the correctness of implementations when bifurcations in networks exist. After every addition or change to an implementation, the implementations were tested with the dummy data. The dummy data is available on the CD-ROM in the testgeodatabase.gdb. Only when the implementation with the dummy data succeeded the tests were also made with real world data. Code that proved to work properly and produced the right output was saved in form of code snippets and reused if possible to reduce test times.

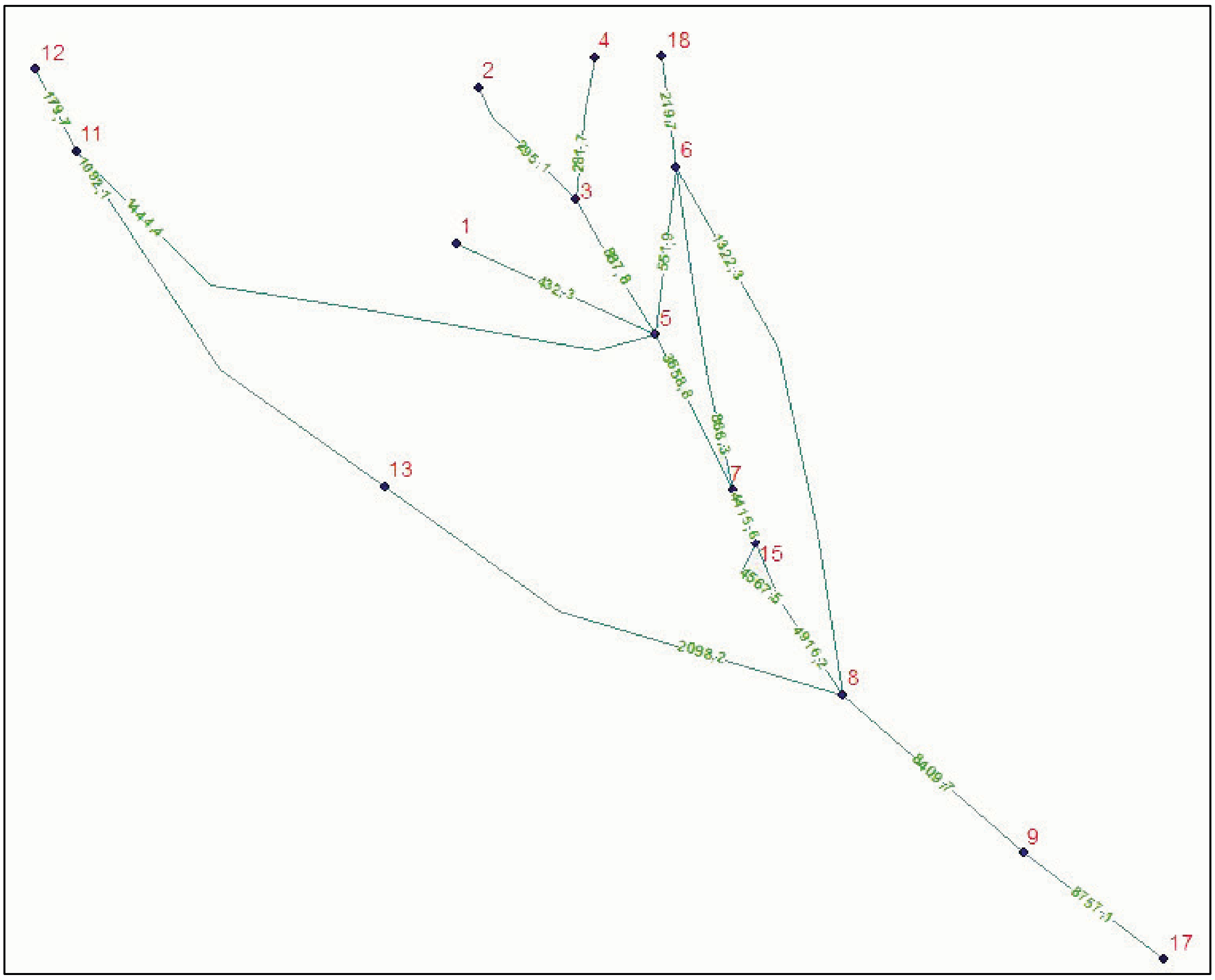


Figure 4–2 Example for a dummy data set to test braided networks

4.1.4 Data structures for the use with the algorithms

Algorithms go hand in hand with data structures-schemes for organizing data that leave them amenable to efficient processing by an algorithm. ([SEDGEWICK 2011], pos. 282)

The data modelling should always be seen in context with the algorithms that should use the data. The choice of a certain data structure is dependent on the tasks to accomplish. Data structures are useful to organize data so that it can be accessed quickly and usefully. There are many different abstract data types, which support different sets of operations and are suitable for different types of tasks, e.g. lists, sets, stacks, queues, heaps, search trees, hash tables, map... or the more specialized ones for networks as

already mentioned in sections 1.2 and 2.3. The Java language provides an extensive Collections Framework that covers many of the mentioned basic types [JAVA 2012].
As the creation of a data structure consumes resources (running time and disk space), it may not be smart to create a "best qualified for the task"-data structure but to find a (minimal) data structure that supports all the operations needed by the task. On the other hand it may be "cheaper" to create a special data structure, when the handling of a more general structure causes a very large constant number of steps for an iteration.

What proved to be especially helpful during computation of DSFs were data maps (to find out in constant time if a junction/edge/catchment had inflowing elements) and in some implementation simple arrays (to define which of the elements in the DSF were already visited).

All the used basic data structures have to be calculated from the GIS-dataset (which mostly happens in linear time) and happen to exist only during the computation in the computers RAM. For this calculations the GIS-dataset must have a data structure that allows reading the adjacent features, thus some form of adjacency list (or matrix) must exist. That is the case for the used network object model of ArcGIS. The calculations output is written into the GIS table (therefore to the hard disk) that is addressed by the certain algorithm.

Some of the algorithms make use of special fields in the GIS-dataset (e.g. the NEXTDOWNID in the catchments are used for preparing the catchment network).

4.2 Which Algorithms are necessary/helpful/beneficial for the use with water Networks?

4.2.1 The DRWaterNetworkAddin

GIS already provide a lot of standard tools to prepare and analyze data. Some of the points from the conclusions in chapter 3.5 can easily be handled by standard tools: these are "Getting information about the network and its elements" and the points about relating data. There are many tools to prepare a network, but in my case those weren't sufficient in every detail, therefore the following section holds additional algorithms for network preparation. Further sections show examples for ordering and data accumulating. To make the text more readable I put some of the descriptions into the Annex 8.2 Supportive Java Add-ins when the implementation does not offer any new insights but is just a useful byproduct.

The following section gives a short overview of the algorithms and the Java implementations that hold them. To easily use and test them in ArcGIS they are prepared as an ArcGIS Add-In called DRWaterNetworkAddin. Eclipse Galileo and JDK 1.6 served very well during the development. To use this Add-In a Java Runtime Environment JRE6 must be installed. If this is ensured, copy "DRWaterNetworkAddin" onto the computer and add the path in the "ArcGIS Add-In Manager Options", if you run ArcGIS 10.0. For newer versions of ArcGIS you may have to compile them once again (e.g. using Eclipse and the suitable ArcGIS Eclipse Plugin and JDK and install a suitable JRE) - in some cases it may be necessary to exchange deprecated classes/methods against newer versions.

The assembly of algorithms has a very simple structure: a toolbar (GN Toolbar), which is subdivided by using menus, makes the Add-Ins available.

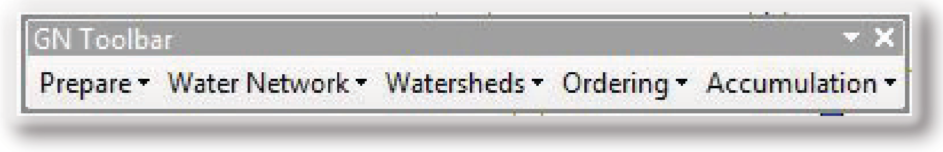


Figure 4–3 GN Toolbar

Prepare:

- Create and check the HydroID (CheckHydroID.java)
- Find Multipart Features (Multipart.java)

Water Network:

- Calculate From/To Nodes (FromToNode.java)
- Set Flow Direction (SetFlowDir.java)
- Calculate Junction Valence (JunctionValence.java)
- Find circles in the Dataset (FindCircles.java - uses the Kosaraju-Sharir algorithm)
- Find Backflow elements (FindBackflows.java)

Watersheds:

- Create Watershed Network (CreateWatershedNetworkGP.java)
- Calculate Basin Circularity (CalculateBasinCircularity.java)

Ordering:

- Calculate Strahler orders with K. Lanfears algorithm (StrahlerLanfear.java)
- Calculate Strahler orders with A Gleyzers algorithm (StrahlerGleyzer.java)
- Calculate Shreve orders (Shreve1.java)

Accumulation:

- Accumulate River Data (AccumulateRivers.java)
- Accumulate Catchment Data (AccumulateCatchment.java)
- Aggregate Watersheds (array version) (AggregateWatersheds.java)
- Aggregate Watersheds (in memory recursion) (AggregateWatersheds1.java)
- Aggregate Watersheds (feature class version)(AggregateWatersheds2.java)

The complete Java source files can be found on the CD in the DRWaterNetworkAddin/src/gn_pack folder.

The implementations were made with special view to the Austrian River Network for Reporting [AUSTRIAN RIVER NETWORK 2012] thus if there are fieldnames or dataset names addressed when reading or writing data, they are from this dataset. I did not bother to make those names variable.

4.2.2 Network preparation

Network preparation is a major part of working with networks. Many of the network algorithms implementations mentioned in the next subchapters only work in a proper way if the network has the following properties:

- Network must be a geometric network - a geometric network can be constructed from available river data using the ArcGIS software
- Network must have flow directions - with nonbraided rivers this can be very simply achieved by using GIS software tools (e.g. ArcGIS Utility Network tools) and a pour point (sink), with braided rivers a field called FLOWDIR can be added to the edges, which will be used by the SetFlowDir.java ArcGIS Java Add-In (see 8.2.2).

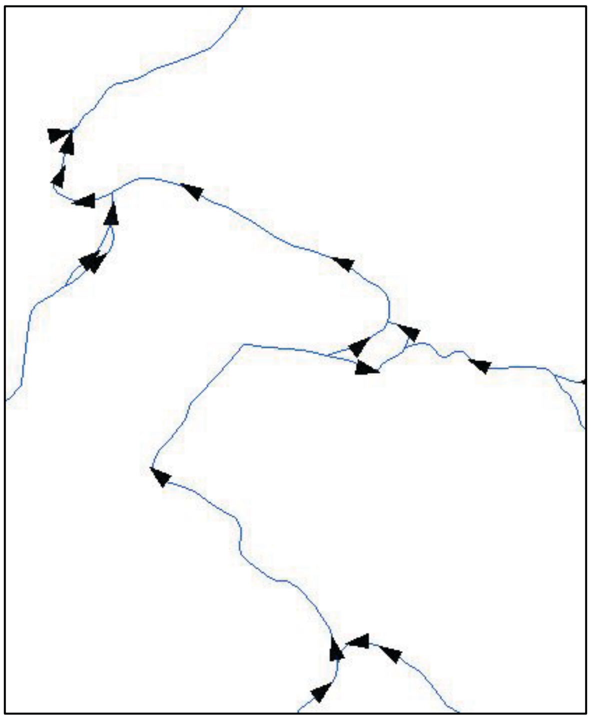
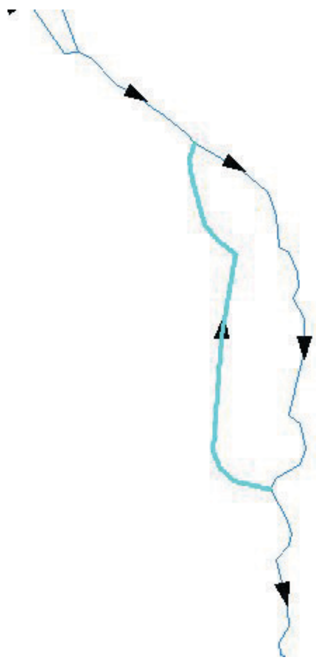
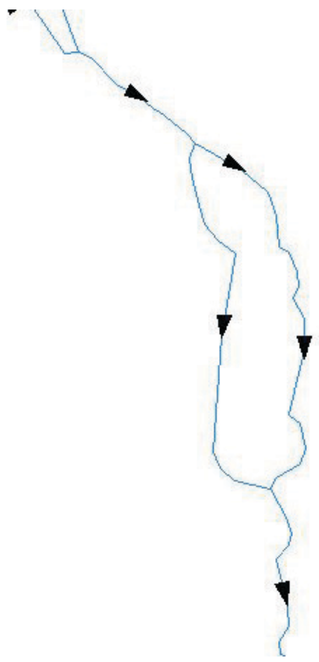


Figure 4–4 Example of flow direction in a braided network

- Because the OBJECTID, which is the standard unique feature number in ArcGIS could be changed, when the database is rebuilt, there exists a field HydroID for all the hydrological/hydrographical features in the database [AUSTRIAN RIVER NETWORK 2012]. To ensure the uniqueness of the HydroID for every data set and facilitate the tracking of changes, the CheckHydroID Add-In checks for not unique HydroIDs and archives old numbers of features that got a new number because being not unique at first. (c.f. 8.2.1). Before calculating a catchment network this Add-In can be used to check for ID-errors in the data set.
- The FromToNode Add-In (see 8.2.3) uses the HydroID to describe from point and to point of each edge in the network. The HydroID numbers of the edges endpoint are written to its feature table and can be addressed there directly.

4.2.2.1 Find backflows

River and catchment networks must not contain circles - they can easily be detected in edge data in linear running time using the FindBackflows ArcGIS Java Add-In or the FindCircles ArcGIS Java Add-In (see 4.2.2.2.). Based on the generated list of circle or backflow elements the value in the flow direction field can be corrected and with a new run of the SetFlowDir Add-In for the selected elements the wrong directions can be quickly and easily corrected.

 Figure 4–5 Backflow in the network	 Figure 4–6 Correction of backflow element

FindBackflows.java returns the backflow river segment from every circle. Because the backflow segment is usually the segment that has to be corrected, there is no need to mark the whole circle (this could for example be done if necessary when backtracking - one would have to keep track of the fromNode and add all edges to a stack, then retrieve them until the fromNode of the current arc is the same as the from Node of the backflow segment).

Data preconditions:

- Networks pour points
- River segments with network topology

Detail Description:

The central method is called "checkForBackflows" and uses a DFS on edges. The trick is to mark the branch nodes in the upward phase of the DFS and if the marked nodes are hit again in the upward phase they must be circles. When backtracking again the mark has to be removed - because otherwise parallel edges will be seen as circles when coming to the same node.

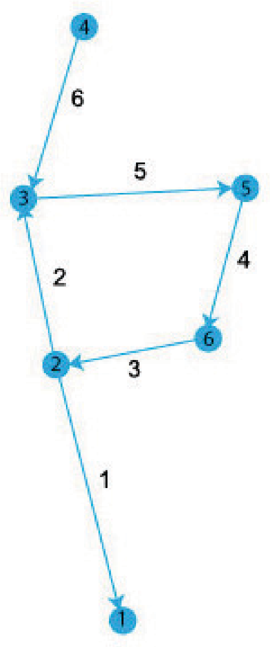
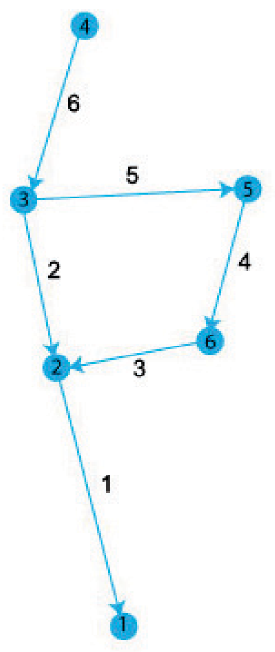
	**Pour point: 1**	---->checkForBackflows(2) //Edge 2
	checkForBackflows(1) //Edge 1	visited={1,2,3,4,5}
	visited={1}	fromNode=2
	fromNode=2	backflow={2}
	inflowingArcsPerNode(2)={3}	inflowingArcsPerNode(2)={3}
	outflowingArcsPerNode(2).size()=2	// Edge 3 was already visited
	braided={2}	outflowingArcsPerNode(3).size()=1
	->checkForBackflows(3) //Edge 3	braided={2}
	visited={1,3}	<---- // Edge 2 - no change in braided
	fromNode=6	---->checkForBackflows(6) //Edge 6
	inflowingArcsPerNode(6)={4}	visited={1,2,3,4,5,6}
	outflowingArcsPerNode(6).size()=1	fromNode=4
	braided={2}	inflowingArcsPerNode(4)={}
	-->checkForBackflows(4) //Edge 4	outflowingArcsPerNode(4).size()=1
	visited={1,3,4}	braided={2} // no change in braided
	fromNode=5	<---- // Edge 6 - no change in braided
	inflowingArcsPerNode(5)={5}	<--- // Edge 5 - no change in braided
	outflowingArcsPerNode(5).size()=1	<-- // Edge 4 - no change in braided
	braided={2}	<- // Edge 3 - no change in braided
	--->checkForBackflows(5) //Edge 5	// Edge 1
	visited={1,3,4,5}	braided={}
	fromNode=3	---
	inflowingArcsPerNode(3)={2,6}	---
	outflowingArcsPerNode(3).size()=1	**Output: backflow={2}**
	braided={2}	
	**Pour point: 1**	outflowingArcsPerNode(6).size()=1
	checkForBackflows(1) //Edge 1	braided={}
	visited={1}	-->checkForBackflows(4) //Edge 4
	fromNode=2	visited={1,2,3,4,6}
	inflowingArcsPerNode(2)={2,3}	fromNode=5
	outflowingArcsPerNode(2).size()=1	inflowingArcsPerNode(5)={5}
	braided={}	outflowingArcsPerNode(5).size()=1
	->checkForBackflows(2) //Edge 2	braided={}
	visited={1,2}	--->checkForBackflows(5) //Edge 5
	fromNode=3	visited={1,2,3,4,5,6}
	inflowingArcsPerNode(3)={6}	fromNode=3
	outflowingArcsPerNode(3).size()=2	inflowingArcsPerNode(3)={6}
	braided={3}	// Edge 6 was already visited
	-->checkForBackflows(6) //Edge 6	outflowingArcsPerNode(3).size()=2
	visited={1,2,6}	braided={3}
	fromNode=4	<--- // Edge 5
	inflowingArcsPerNode(4)={ }	braided={}
	outflowingArcsPerNode(4).size()=1	<-- // Edge 4 - no change in braided
	braided={3}	<- // Edge 3 - no change in braided
	<-- // Edge 6 - no change in braided	// Edge 1 - no change in braided
	<-// Edge 2	
	braided={}	---
	->checkForBackflows(3) //Edge 3	---
	visited={1,2,3,6}	**Output: backflow={}**
	fromNode=6	
	inflowingArcsPerNode(6)={4}	

Figure 4–7 Find Backflows-examples - arrows to right indicate depth first part- arrows to left indicate backtracking

The Java-method checkForBackflows reveals the details of the algorithm.

```
public void checkForBackflows(int ArcID){
   visited.put(ArcID, true); //as soon as arc is visited it is marked
   //get fromNode
   int fromNode = fromNodePerArc.get(ArcID);
   //get count of inflowingArcsPerNode
   int inflowingArcs = inflowingArcsPerNode.get((Integer)fromNode).size();
   // define a boolean braidadd
   boolean braidadd;
   //use lookup to find out if the from node is a starting point for braids
   // -if yes try to add it to the global set "braided"
   // -if it can't be added, because it is already in the set it is a
   // circle - add it as backflow to the global set "backflow"
   if(outflowingArcsPerNode.get((Integer)fromNode).size()>1){
   braidadd=braided.add(fromNode);
   if(!braidadd){
      //then node was already in the set - thus there is a circle
      backflow.add(ArcID);
   }
   }
   //An edge can only be part of a circle if it has upper adjacent edges thus
there   //is no need to consider a base case (most upper edges)
   if (!(inflowingArcs == 0)){
   //if the river with ArcID is no upper arc
   //get the inflowingArcsPerNode
   Iterator<Integer> it1;
   it1=inflowingArcsPerNode.get((Integer)fromNode).iterator();
   int arc;
   //for every inflowingArc do the following:
   while(it1.hasNext()){
      arc = (Integer)it1.next();
      //if the arc has never been visited
      if(!visited.containsKey(arc)){ //only if incoming arc was never
                 //visited it is called again
      checkForBackflows(arc);   //recursive part
      }
   }//while ends
   }// if ends
   // when backtracking do not forget to remove the braided entry
   // otherwise all parallel rivers would be marked as backflows
   if(braided.contains(fromNode)){
   braided.remove(fromNode); // the braided entry is removed
   }
   return;
}
```

Running time:

The Add-in uses DFS and only visits every arc once. Its running time is therefore O(E+V), thus O(n).

Time Complexity	Steps
O(n)	Once visit every edge, thus n is the number of edges
Space Complexity	
The implementation creates the following maps: inflowingArcsPerNode (number of nodes), outflowingArcsPerNode (number of nodes) and fromNodePerArc (number of arcs). Space complexity is thus O(n).	

4.2.2.2 Find circles

FindCircles.java uses the Kosarju-Sharir algorithm for finding strongly connected components as described in detail in [SEDGEWICK 2011].

"*Two vertices v and w are strongly connected if they are mutually reachable: that is, if there is a directed path from v to w and a directed path from w to v. A digraph is strongly connected if all its vertices are strongly connected to one another.*" ([SEDGEWICK 2011], pos. 10348)

Returns a list of junctions of the rivers that form a circle (or cycle as it is also called).

Data preconditions:

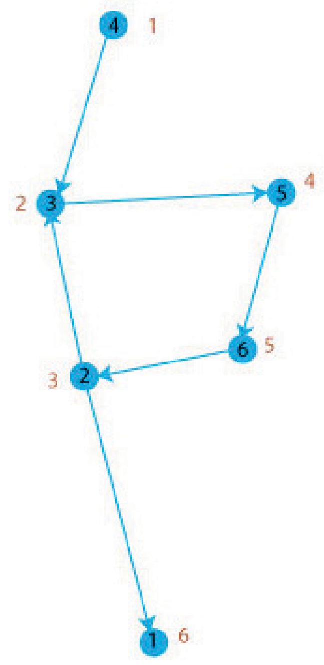
- Networks pour points
- River segments with network topology

Detailed Description:

FindCircles.java creates a directed graph and calls the Kosaraju-Sharir algorithm on it. The Kosaraju-Sharir algorithm uses a DSF, with a special numbering of the vertices as they appear in the network, instead of using the vertices standard id. It makes use of the fact that the transpose graph has the same strongly connected components as the original graph (because in a connected component every node is connected to another either directly or via a sequence of other nodes - when reversing the direction does not introduce a change in the connection). Proof can be found in [CORMEN 2009].

The steps of the Kosaraju-Sharir algorithm are the following (c.f. [SEDGEWICK 2011], [CORMEN 2009], p. 617):

1. Make a DFS on the graph and number the vertices in the same order as their recursive calls are completed (see Figure 4–8 top).
2. By reversing the direction of every arc in G construct a new directed graph $G_{reversed}$.
3. Perform another DFS on $G_{reversed}$, start with the vertex that had the highest number assigned. Should this DFS not reach all numbered vertices, then choose the highest numbered vertex from the remaining vertices and run a DFS (see Figure 4–8 bottom).
4. Each tree in the resulting spanning forest is a strong component of G.

	DFS on the graph (original vertex numbering in black) produces a new numbering (brown)

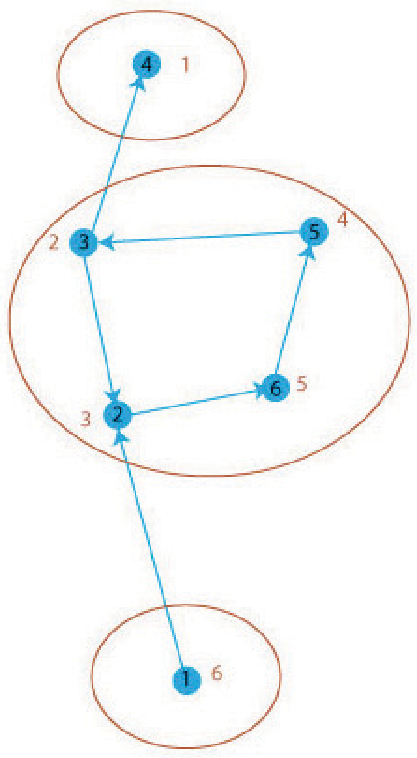
	DFS on reversed graph: - starts at new number 6 (because it is the highest) ⇒ no inflowing arc - thus DFS stops 6 is one component - start at new number 5 (because it is the highest after 6 was removed) ⇒ 5, inflowing arc from node 3 ⇒ inflowing arc from node 2 ⇒inflowing arc from node 4 ⇒ node 5 is already visited - thus 5, 3, 2, 4 are one strongly connected component - start at new number 1 (because it is the highest after the others were removed) ⇒ no more inflowing arcs, thus 1 is one component

Figure 4–8 Example of the progression in the Kosaraju-Sharir algorithm

Running time:

The Add-in uses two DFS, which both visit every edge once. Its running time is therefore O(V+E).

Time Complexity	Steps
O(V+E)	Once run through the original graph from its source node and once on the reverse graph.

4.2.3 Tracing rivers and catchments

Tracing rivers upstream and downstream is already well established in many GIS and should therefore only be mentioned here for the sake of completeness. The same functions can be used on catchments provided that the catchments have a sort of network structure, like the "NextDownID" in the Austrian River Network for Reporting.

In ArcGIS therefore, in addition to the built-in functions of creating geometric networks from edges and junctions (as used for the river data) for the catchment data, a catchment network topology has to be established. A simple and permanent way to create a catchment network topology is using the catchment centroids as junctions and create a topology using information about how the catchments are connected (thus the NextDownIDs). CreateWatershedNetworkGP.java (see 8.2.5) uses the available NextDownIDs of the Austrian River Network for Reporting and creates centroids for the catchment polygons. Between every pair of adjacent centroids[19] ,a straight line is created and centroids and lines are forged into a Geometric Network using ArcGIS geoprocessing tools.

4.2.4 Calculating stream orders and adventitious streams

The main characteristics of the stream ordering systems are described in Appendix 8.1 Stream order systems. The group of stream order systems using the top down approach are similar to each other. Algorithms already exist for some (Strahler and Shreve) of them and are implemented in other GIS-systems (e.g. the Strahler and Shreve Avenue Routines by **Duncan Hornby** from 2003 which are based on [LANFEAR 1990], or his VisualBasic version of the Gleyzer [GLEYZER 2004] algorithm in the RivEX Software for ArcGIS 9.3 from 2010 (http://www.rivex.co.uk). Because during writing of this theses none were implemented for ArcGIS 10.x., which is used by many contributors to the Austrian surface water dataset for reporting as well as to the European river dataset, the known algorithms ([LANFEAR 1990, GLEYZER 2004]) are therefore implemented as Add-ins for ArcGIS 10.0.

Similar stream orders were tackled with the same approach.

All of the algorithms concerning river ordering more or less have the same preconditions:

- River network topology exists (rivers are connected, ...)
- Each river segment has to have a unique number
- The stream order will be written in a certain column for each river segment
- River segments should be digitized in flow direction or at least flow direction in relation to digitized direction has to be known

[19] Adjacency is derived from the NextDownID.

- End points and/or starting points may have to be marked specially (this does explicitly not apply for the Lanfear stream ordering algorithm and could also be packed in pre-calculation)

The central ordering method in all of the implementations is a DSF based strategy, which works straightforward for nonbraided networks. Braiding is the main cause why each of those implementations needs an extra method. In this method the braiding is addressed, otherwise errors would be introduced. Figure 4–9 shows what would happen when braids are not handled. The segment that has Shreve order number 37 should have Shreve order number 19 because Shreve order number is the number of all head waters above a certain edge, a simple Shreve algorithm would just add up the numbers at every junction, which would in this case result in an error.

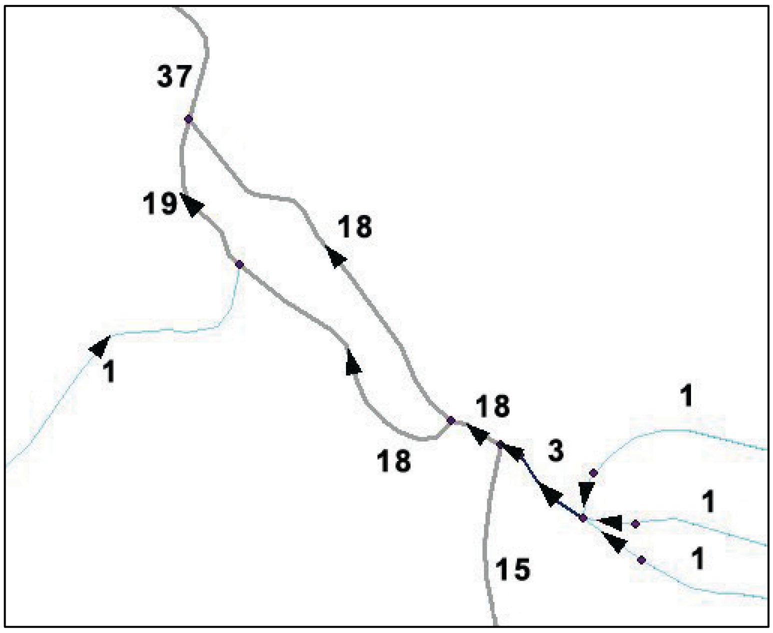


Figure 4–9 Error in Shreve ordering if braiding is not addressed properly

For the ordering systems based on a bottom up strategy further information is needed on how to distinguish the main rivers from the others: this could be the throughflow or the aggregated length or the aggregated area for each river reach. In 4.2.4.3 the strategy to implement such a bottom up system is drafted.

Adventitious streams do not need to be calculated through a new algorithm, but they can for example be calculated in short enough time via a simple spatial query available in many GIS[20].

The same is also valid for the **bifurcation ratio** for certain catchments. The implementation of the Gleyzer stream-ordering algorithm also calculates segments that can easily be summed up for a catchment using the standard tools provided in a GIS.[21]

4.2.4.1 Strahler-Algorithms

4.2.4.1.1 Lanfear's Strahler algorithm

StrahlerLanfear.java (an Add-In for ArcGIS 10.0) calculates Strahler stream order, and is based on the algorithm of Keneth Lanfear [LANFEAR 1990]. The calculation of the stream order is done in an iterative way and the iteration only stops when there are no more changes to be made (this is dependent on the number of segments of the longest path from headwaters to mouth).

Data preconditions:

- River segment with a unique id number
- River segments have a calculated from and to node (e.g. use 8.2.3 FromToNodes.java)

Running time:

Lanfear gives as Execution time $n*\log_2 n*p$, where n is the number of overall segments, p is the number of segments in the longest path from headwaters to mouth.

The calculation of the order values is dependent on the already calculated confluences. In the worst case (i.e. when the lower parts of the rivers are processed before the headwaters) it will take as long as the highest count of segments from one of the headwaters to the outlet before all rivers can be calculated. In the best case (i.e. when the rivers are topologically sorted from headwater to outlet) it will only take two

[20] For example select all features with a Strahler number smaller than 1 (or 2 depending on the used definition for adventitious streams), and query the output against an intersection with the selection of Strahler number > 4. This results in the lowest segments of adventitious streams.

[21] For ArcGIS this would involve the following steps: select the rivers of the catchment to be calculated then summarize over order and segment_id.

executions to calculate the Strahler numbers, because the upper rivers have already their right numbers and the last (in this case the second) execution is only done to make sure that nothing has yet to be calculated.

Time Complexity	Steps
constant	Initialize document and create the necessary fields and indices
c1*n	Create 2 arrays of river data (first for ID, FromNode, ToNode, StrahlerOrder, NodeOrig; second for ToNodes) and fill it with the contents of geodatabase river segment attribute table.
c2*n*log$_2$n	Sort arrays using ToNode - Arrays.java uses a tuned quicksort adapted from [BENTLEY 1993]
c3*n*log$_2$n*p	Calculate Strahler order for every river (n) * c3 (=number of rivers flowing into the segment - this is counted as a constant because there are never more than low number of rivers (usually 2, seldom 3, very seldom more than 3) flowing into a river) - find the corresponding node (log$_2$n) - repeat this p times (p)
c4*n*log$_2$n	Sort arrays using ID - Arrays.java uses a tuned quicksort
c5*n	Write the Strahler orders back into the river segment attribute table.
O(n*log$_2$n*p)	
Space Complexity	
Space for the data array and space for an ArrayList holding the Strahler number and node origin for each river flowing into a certain river (- this ArrayList is empty for headwaters, for the average confluence it holds two items and in some cases it holds three items). The space complexity is of order O(n)	

The implementation of Lanfear's Strahler algorithm deals with pseudo nodes and braided rivers of a low complexity. Strahler numbers of braided rivers with a high complexity (those are braided rivers which very long parallel river segments which themselves have other contributing streams and are split again ...) have to be viewed with a critical eye. Those zones of high complexity have usually an anthropogenic touch and it can be necessary to "clean" the data set beforehand, especially if there are a lot of channels, which should strictly speaking not be ordered (at least when following Horton's idea of stream ordering).

4.2.4.1.2 Gleyzer's Strahler algorithm

StrahlerGleyzer.java (an Add-in for ArcGIS 10.0) calculates Strahler stream order and is based on the algorithm of Alexander Gleyzer, Michael Denisyuk, Alon Rimmer, and Yigal Salingar [GLEYZER 2004] for braided rivers. It works like a Depth-First Search with nested-recursion (see method streamOrdering). The ordering part (getOrder) is nearly the same as in Lanfears algorithm. Additionally river segments are calculated, without having to implement an extra loop.

The data dictionaries in the algorithm were all implemented as hashMaps, which allow getting and putting data in constant time. All but the inflowingArcsPerNode hashMaps are purely Integer hashMaps. The inflowingArcsPerNode has a generic type <Integer, ArrayList<Integer>> whereas the ArrayList<Integer> is used to hold all the inflowingArcs.
Therefore, in the recursive loop the additional local data dictionary upstreamOrders is implemented as an ArrayList, because it again holds data of inflowingArcs.

Data preconditions:

- Networks pour points
- River segments with network topology
- No backflows

Running time: Gleyzer et.al. indicate O(n), where n is the number of overall reachable segments (edges).
Code details of the recursion part reveal that this is right and works for all river networks without backflow (that is a circle in the river network or a reflective arc):

(To facilitate readability of the essential part some code is omitted and replaced by)
The streamOrdering method is called for every pour arc in the network. Only if an arc was not called already it will be called via the streamOrdering method.

```
...
for(int i = 0; i<pourEdgeCount;i++){
   pourEdgeFeature=pourSJFeature.getEdgeFeature(i);
   pourStartArc=(IFeature)pourEdgeFeature;
   intStreamOrder=streamOrdering(pourStartArc.getOID); //calls the
            // streamOrdering for every pour arc
}
...


public int streamOrdering(int ArcID){
   visited.put(ArcID,true); // as soon as an arc is visited it's
visited
             // status is set true
...

   //if the arc has never been visited
   if((visited.get(arc))==false){
   upstreamOrder.get(i).add(streamOrdering(arc));//find its order - 0
   ...

   //if it already has been visited
   }else{
   upstreamOrder.get(i).add(streamOrder.get(arc));//order         -0
   ...
   }
```

Claim 1: streamOrdering is called exactly once for each reachable arc.

Proof: streamOrdering(e) starts at a pour arc (the arc that has no outflow). It finds all the inflowing arcs for this arc and calls streamOrdering for them only if they were not visited yet. QED.

Claim 2: The call of getOrder is executed exactly once for each edge.

Proof: getOrder is called once per execution of streamOrdering. And following claim 1 the streamOrdering is executed exactly once for each reachable arc. QED.

Therefore, running time of streamOrdering is O(n) .

The initialization of the segment array in this implementation follows the idea, that the maximum number of orders in a network is $\log_2$ (n+1) or smaller. This is because for any node with a certain Strahler number i there must be at least two nodes with a Strahler number i - 1, thus four nodes with Strahler number i - 2, eight nodes with Strahler number i - 3, etc. - c.f. Figure 4–10 Ideal Structure of Strahler-ordering.

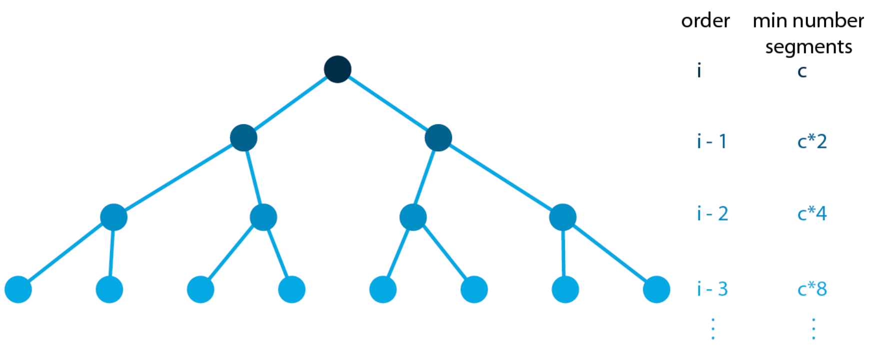


Figure 4–10 Ideal Structure of Strahler-ordering.

Thus the minimum number of river segments for an ideal network (which would be a binary tree) is 2^{order} - 1. This leads to the conclusion that given a number of rivers the possible maximum number of orders is $\log_2$ (n+1). This facilitates the usage of an array without using too much extra space but with its advantage of its easy handling.

As the implementation of the recursion in Java has a return statement for both the base case (all upper arcs with no inflow) and the other cases, the additional river segment calculation is made before both return statements and not only as suggested in the algorithm pseudo code once at the end.

Time Complexity	Steps
constant	Initialize document and create the necessary fields and indices
c*n	Create hashMaps (visited, streamOrder, fromNodePerArc, originatingNode, inflowingArcsPerNode, segments) and fill them with data
c	Create and fill segment array
c*n	Calculate Strahler order and river segment number for every river
c*n	Write the Strahler orders and river segment numbers into the river segment attribute table.
O(n)	
Space Complexity	
>12 n	Extra space for 6 hashmaps (hashmaps have to be designed large enough to avoid to much rehashing, thus for every hashmap a small extra amount of space has to be planed). The order of space complexity is O(n).
$\log_2$(n+1)	Extra space for river segments maximum number per order array - thus negligible.

The implementation of the Gleyzer's Strahler algorithm deals like the above implementation with pseudo nodes and braided rivers of a low complexity. Again the Strahler numbers of braided rivers with a high complexity (those are braided rivers which very long parallel river segments which themselves have other contributing streams and are split again, ...) have to be viewed with a critical eye and it may be necessary to "clean" the data set before calculation, especially if there are a lot of channels, which should strictly speaking not be ordered (at least not when following Horton's idea of stream orders).

Central methods in StrahlerGleyzer.java

```
public int streamOrdering(int ArcID){
  visited.put(ArcID,true);
  int fromNode = fromNodePerArc.get((Integer)ArcID);
  get count inflowingArcsPerNode
  int inflowingArcs =
    inflowingArcsPerNode.get((Integer)fromNode).size();
  if (inflowingArcs == 0){//Base case
  streamOrder.put((Integer)ArcID, 1);
  int node = originatingNode.get(ArcID);

  //stream segments
  if (!(segmentIDsPerOriginatingNode.containsKey(node))){
    int j = 0;//because order start with 1, but arrays start at 0
    segmentID[j]=segmentID[j]+1;//get the segmentID value of the order
                               //and increase it
    segmentIDsPerOriginatingNode.put(node, segmentID[j]);
  }
  segments.put(ArcID,segmentIDsPerOriginatingNode.get(node));
  return 1;

  }else{//if the river with ArcID is no head water
  //get the inflowingArcsPerNode
  Iterator<Integer> it1;
  it1=inflowingArcsPerNode.get((Integer)fromNode).iterator();
  int arc;
  int i=0;
  ArrayList<ArrayList<Integer>>upstreamOrder=new
           ArrayList<ArrayList<Integer>>();
  //for every inflowingArc do the following:
  while(it1.hasNext()){
    arc = (Integer)it1.next();
    //upstreamOrder=new ArrayList<ArrayList<Integer>>();
    upstreamOrder.add(new ArrayList<Integer>());
    //if the arc has never been visited
    if((visited.get(arc))==false){
    upstreamOrder.get(i).add(streamOrdering(arc));//find its order - 0
    upstreamOrder.get(i).add(originatingNode.get(arc)); //origin   - 1

    //if it already has been visited
    }else{
    upstreamOrder.get(i).add(streamOrder.get(arc));//order         -0
    upstreamOrder.get(i).add(originatingNode.get(arc));//origin   -1

    }
    i=i+1;
  }//while end

  return  getOrder(upstreamOrder, ArcID);
  }
}

public int getOrder(ArrayList<ArrayList<Integer>> upstreamOrder, int
            ArcID){
  int maxOrder=0;
  int maxOrderCount=0;
  int order=0;
  int origin;
  int maxOrderOrigin=0;

  for(int i = 0;i<upstreamOrder.size();i++){
  order=upstreamOrder.get(i).get(0);
  origin=upstreamOrder.get(i).get(1);
  if(order>maxOrder){
    maxOrder=order;
    maxOrderCount=1;
    maxOrderOrigin=origin;
  }else if (order==maxOrder){
    if (maxOrderOrigin!=origin){
    maxOrderCount=maxOrderCount+1;
    }
  }//do nothing if maxOrder less than order
  }

  if (maxOrderCount > 1){
  order=maxOrder+1;
  streamOrder.put(ArcID, order);
  originatingNode.put(ArcID, fromNodePerArc.get(ArcID));
  }else{
  order =maxOrder;
  streamOrder.put(ArcID, order);
  originatingNode.put(ArcID, maxOrderOrigin);
  }
  // Calculation of the river Segments: if there is no segment Id
  // from the originatingNode of edge ArcID then add one
  int node = originatingNode.get(ArcID);
  if (!(segmentIDsPerOriginatingNode.containsKey(node))){
  int j = order - 1;//because order start with 1, but arrays start at
0
  segmentID[j]=segmentID[j]+1;//get the segmentID value of the order
                              //and increase it
  segmentIDsPerOriginatingNode.put(node, segmentID[j]);
  }
  segments.put(ArcID,segmentIDsPerOriginatingNode.get(node));
  return streamOrder.get(ArcID);
  }
```

4.2.4.2 Shreve/Scheidegger/Rzhanitsyn algorithm

A brute-force-algorithm for calculating Shreve orders would be to visit every node and make a DFS to find all the headwater nodes and count them. This would be n* n/2 if we assume a perfect binary tree, thus be of O(n^2).

Shreve.java (an Add-in for ArcGIS 10.0) calculates Shreve stream magnitude and is based on the recursive algorithm of Gleyzer et.al [GLEYZER 2004]. The streamOrdering-method, and the getOrder-method had to be changed to fit Shreve's concept [SHREVE 1966]. The approach is quite straightforward for rivers without braiding. In this case its runs in O(n). The braiding introduces an extra component into the code that makes sure, that every headwater is counted only once. This was accomplished using a set approach.

For every braided river the origin node and all upper braid nodes (these are all those nodes where a river is branching out), are saved in a hashMap, and for every branching origin node a current count is maintained. If a braiding ends, then this braiding's origin node (or nodes) is (are) not added any more to the next down river(s). For the low-land reaches of rivers these braiding reaches can become rather complex sets, but seen overall they usually involve only a certain portion of the river data set (e.g. in the Austrian surface water dataset for reporting there are about 10% rivers branching out, only about a third of them is involved in branching sets with more than one branching). Thus using a recursive approach is still a quick way to calculate the Shreve numbers.

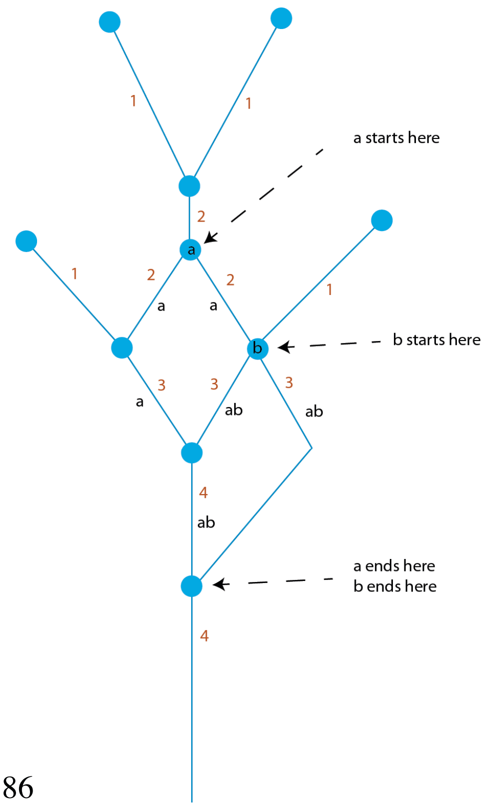


Figure 4–11 Branching schematic

Details of calculating Shreve orders:

In the preprocessing phase the following data maps are created for the edges:

visited [ArcID]... empty at the beginning - will hold all edges at the end

streamOrder[ArcID]...initialized with 0

fromNodePerArc[ArcID]...the FromNode-ID of the edges

inflowingArcsPerNode[Arc's FromNodeID] ... the list of the inflowing edges

outflowingArcsPerNode[Arc's FromNodeID] ... the list of the outflowing edges

Nodes where braiding starts are marked and their outflowing edges are counted:

If | outflowingArcsPerNode[Arc's FromNodeID] | > 1,

then the count is put in the data map origNodeCnt[Arc's FromNodeID]

origNode[Arc's FromNodeID] ... the FromNode-ID of the edges

Starting with the pourPoints the streamOrdering method is called once for every edge comparable to Gleyzers Strahler algorithm.

```
public int streamOrdering(int ArcID){
  //set visited
  visited.put(ArcID,true);
  //get fromNode
  int fromNode = fromNodePerArc.get((Integer)ArcID);
  //get count of inflowingArcsPerNode
  int inflowingArcs =
inflowingArcsPerNode.get((Integer)fromNode).size();
  if (inflowingArcs == 0){//Base case
  for(int x : outflowingArcsPerNode.get(fromNode)){
    streamOrder.put(x, 1);
    visited.put(x, true);
  }
  return 1;
  }else{//if the river with ArcID is no upper arc
  //get the inflowingArcsPerNode
  Iterator<Integer> it1;
  it1=inflowingArcsPerNode.get((Integer)fromNode).iterator();
  int arc;
  int i=0;
  ArrayList<ArrayList<Integer>>upstreamOrder=new
            ArrayList<ArrayList<Integer>>();
  //for every inflowingArc do the following:
  while(it1.hasNext()){
    arc = (Integer)it1.next();
    upstreamOrder.add(new ArrayList<Integer>());
    //if the arc has never been visited
    if((visited.get(arc))==false){
    upstreamOrder.get(i).add(streamOrdering(arc));//find its order - 0
    upstreamOrder.get(i).add(arc);// arcOID    - 1
    //if it already has been visited
    }else{
    upstreamOrder.get(i).add(streamOrder.get(arc));//order         - 0
    upstreamOrder.get(i).add(arc);// arcOID    - 1
    }
    i=i+1;
  }//while end
  return  getOrder(upstreamOrder, ArcID);
  }
}
```

public int getOrder(ArrayList<ArrayList<Integer>> upstreamOrder, int

```
ArcID){
  int sumOrder=0;
  int order=0;
  int upstreamRiver;
  int fromNode = fromNodePerArc.get(ArcID);
  int outCount = outflowingArcsPerNode.get(fromNode).size();
  int upstreamFromNode=0;
  int temp;
  int temp1;
  int temp2;
  int temp3;
  int max=0;
  //create local HashMaps and sets
  Set<Integer> tempNodeSet=new HashSet<Integer>();
  Set<Integer> unionNodeSet=new HashSet<Integer>();
  HashMap<Integer,Integer>nodeSetMapCnt=new
HashMap<Integer,Integer>();
  Set<Integer>nodeSet=new HashSet<Integer>();
  HashMap<Integer,Set<Integer>>nodeSetMap=new
           HashMap<Integer,Set<Integer>>();
  HashMap<Integer,Integer>localOrigNodeCnt=new
HashMap<Integer,Integer>();

  for(int i=0; i<upstreamOrder.size();i++){
  //get the next inflowing upstreamRiver of ArcID
  order = upstreamOrder.get(i).get(0);
  upstreamRiver=upstreamOrder.get(i).get(1);
  sumOrder=sumOrder+order;
  if(max<order)max=order;
  upstreamFromNode=fromNodePerArc.get(upstreamRiver);
  //find out if it is a branching upstreamRiver
  if(origNode.containsKey(upstreamFromNode)){
    //if it is a branching Arc
    //get its fromNode
    nodeSet.add(upstreamFromNode);
    if(!nodeSetMap.containsKey(upstreamFromNode)){
    nodeSetMap.put(upstreamFromNode, origNode.get(upstreamFromNode));
    localOrigNodeCnt.put(upstreamFromNode, 0);
    }else{
    //if the node is already in the nodeSetMap
    //then the order has to be corrected
    sumOrder=sumOrder - streamOrder.get(upstreamRiver);
    // every time the node appears again - the
    // nodeCount will have to be reduced by one
    temp=localOrigNodeCnt.get(upstreamFromNode) + 1;
    localOrigNodeCnt.put(upstreamFromNode, temp);
    }
  }
  }
  // if nodeSetMap is empty then there is nothing to do than
  // to put the streamOrder into the streamOrder Map
  if(nodeSetMap.isEmpty()){
  if(outCount>1){
    for(int x : outflowingArcsPerNode.get(fromNode)){
    streamOrder.put(x, sumOrder);
    visited.put(x, true);
    }
  }else {
```

```
  streamOrder.put(ArcID, sumOrder);
}
return sumOrder;
}else{
//if nodeSetMap contains items then there were
//branching nodes in the input
// localOrigNodeCnt contains up to now only the keys of the from
// upper nodes - but all upper keys have as well to be put there
// for reasons of orderCnt corrections
for(int e:nodeSet){
  if(!(localOrigNodeCnt.get(e)==0)){
  temp1=localOrigNodeCnt.get(e);//Cnt of the nodes
  tempNodeSet = origNode.get(e);
  for(int f:tempNodeSet){
    if(localOrigNodeCnt.containsKey(f)){
    temp=localOrigNodeCnt.get(f)+ temp1;
    localOrigNodeCnt.put(f, temp);
    }else{
    localOrigNodeCnt.put(f, temp1);
    }
  }
  }else{
  localOrigNodeCnt.remove(e);
  }
}
for(Map.Entry<Integer, Set<Integer>> e: nodeSetMap.entrySet()){
  upstreamFromNode = e.getKey();
  tempNodeSet=e.getValue();
  for(int f : tempNodeSet){
  if(nodeSetMapCnt.containsKey(f)){
    temp=nodeSetMapCnt.get(f) + 1;
    nodeSetMapCnt.put(f, temp);
  }else{
    nodeSetMapCnt.put(f, 1);
  }
  }
}
// add nodeSet
for(int e: nodeSet){
  if(nodeSetMapCnt.containsKey(e)){
  temp=nodeSetMapCnt.get(e) + 1;
  nodeSetMapCnt.put(e, temp);
  }else{
  nodeSetMapCnt.put(e, 1);
  }
}
// now exists a list of all the upperNodes that are in
// the node set lists of other upper Nodes
// there are more than one different input
// branching nodes
// loop over all of them and find
// out
// 1 - if there is a branching that ends here
//      -> count all occurences of a node in the origNode sets
//    add the nodeSet at the end
//-> compare every count of a node with its origNodeCnt
//    if not the same -> node has to be put into the
//    origNode set of fromNode
// 2 - if there are branching nodes coming up more than once
//that are not in the nodeSet (because those orders were already
//substracted above!
// ->1
for(Map.Entry<Integer, Integer> e: nodeSetMapCnt.entrySet()){
  upstreamFromNode = e.getKey();
```

```
    temp=e.getValue();//Count
    //find out if it is the next upper branching node
    // repair sumOrder value - every node must only be used once
    temp1= temp;
    // find out if it is one of the next upper branching nodes
    // - otherwise it may be a nested node
    for(Map.Entry<Integer,Integer> f: nodeSetMapCnt.entrySet()){
    temp3=f.getKey();
    if(temp3!=upstreamFromNode){
      tempNodeSet=origNode.get(temp3);
      if(tempNodeSet.contains(upstreamFromNode)){
      temp1=temp1-f.getValue()+1;
      }
    }
    }
    temp1=temp1-1;
    // only substract it if it is the next upper branching node
    if(temp1 > 0){
    sumOrder=sumOrder-temp1 *
  streamOrder.get(outflowingArcsPerNode.get(upstreamFromNode).get(0));
    }
    // repair nodeOrigCnt value
    // first get the nodeOrigCnt
    if(origNodeCnt.containsKey(upstreamFromNode)){
    temp2 = origNodeCnt.get(upstreamFromNode);
    }else{
    temp2 = 1;
    }
    if(localOrigNodeCnt.containsKey(upstreamFromNode)){
    temp=temp+localOrigNodeCnt.get(upstreamFromNode);
    }
    temp1 = temp2 - temp;
    // node ends:
    if(temp1==0){
    origNodeCnt.put(upstreamFromNode, 0);
    }else{
    unionNodeSet.add(upstreamFromNode);
    temp1=temp1+outCount;
    origNodeCnt.put(upstreamFromNode, temp1);
    }
  }
  if(!unionNodeSet.isEmpty()){
    origNode.put(fromNode, unionNodeSet);
    if(unionNodeSet.size()>0){
    countall++;
    }
    if(unionNodeSet.size()>10){
    countGT10++;
    }
    if(unionNodeSet.size()>20){
    countGT20++;
    }
  }
  }
  if (sumOrder<max){
  sumOrder=max;
  }
  //now write the streamOrders
  if(outCount>1){
  for(int x : outflowingArcsPerNode.get(fromNode)){
    streamOrder.put(x, sumOrder);
    visited.put(x, true);
  }
  }else {
```

```
  streamOrder.put(ArcID, sumOrder);
  }
  return sumOrder;
}

//Set Methods
// setX minus setY
public static <T> Set<T> difference(Set<T> setX, Set<T> setY){
  Set<T>temp=new HashSet<T>(setX);
  temp.removeAll(setY);
  return temp;
}
//setX contains setY
public static <T> boolean containedIn(Set<T> setX, Set<T> setY){
  return setY.containsAll(setX);
}
```

The algorithm produces the Shreve order numbers for planar graphs without circles.

Proof A: nonbraided rivers:

Recursion base case: if there are no inflowing rivers, the number is set to one, which is exactly the count of headwaters for one headwater river.

Other cases: if there are inflowing rivers their river numbers are summed up and thus give the numbers of headwaters. Thus after every calculation of stream orders the stream order number is the same as the count of the headwaters. QED

Proof B: braided rivers:

Recursion base case: if there are no inflowing rivers, the number is set to one, which is exactly the count of headwaters for one headwater river.

Other cases: if there are inflowing rivers their river numbers are summed up. Braided rivers may introduce a headwater point twice or even more often and are found out by the algorithm. The count of the headwaters is corrected. Thus after every calculation of stream orders the stream order number is the same as the count of the headwaters. QED

The only possibility for errors is where the graph violates planarity in areas where braiding occurs (in the Austrian surface water dataset for reporting version 8 this occurs exactly once).

To eliminate (most of) the error introduced in these non-planarity cases an additional test was introduced that makes sure that it can never happen that the calculated arc has a smaller order than the maximum order of the inflowing arcs.

Claim 1 and 2 are the same as for the Gleyzer algorithm:

Claim 1: streamOrdering is called exactly once for each reachable arc.

Proof: streamOrdering(e) starts at a pour arc (the arc that has no outflow). It finds all the inflowing arcs for this arc and calls streamOrdering for them only if they were not visited yet. QED.

Claim 2: The call of getOrder is executed exactly once for each edge.

Proof: getOrder is called once per execution of streamOrdering. And following claim 1 the streamOrdering is executed exactly once for each reachable arc. QED.

Claim 3: The time complexity of getOrder grows quadratically with the amount of nested open braidings.

Proof: If there is no open braiding getOrder sums up the inflowing arcs and is done.

If there is open braiding then the braiding nodes are compared with each other in a brute force manner (each with each). Thus the number of comparisons is the square of the number of open braiding nodes.

In the worst case there is only one upper node, which together with every following node branches out. Their braiding only closes at the last node. This would mean $\sum_{i=2}^{n}((i-1)/2))^2$ nesting-depth thus $O(n^2)$ (where i is even because only two outflowing arcs would create a new braid) comparisons. QED.

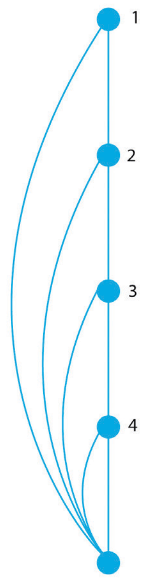


Figure 4–12 Worst case braiding scenario with maximum nesting

As braiding is not the normal case in a river this worst case scenario hardly ever will occur on large scale. Thus in the following calculation the number goes in as b.

Time Complexity	Steps
constant	Initialize document and create the necessary fields and indices
c1*n	Create hashMaps (visited, streamOrder , fromNodePerArc, origNode, inflowingArcsPerNode, outflowingArcsPerNode, segments) and fill them with data
$c2*n + c3*b^2$	Calculate Shreve order for every river segment b is the number of open branchings for this river segment.
c4*n	Write the Shreve orders into the river segment attribute table.
$O(n+ b^2)$	
Space Complexity	
>14 n	Extra space for 7 hashmaps (hashmaps have to be designed large enough to avoid too much rehashing, thus for every hashmap a small extra amount of space has to be planned
f	Extra space for local hashMaps and Sets in the getOrder Method - dynamic size - varies from 0 (in the areas where no branching occurs to the number of active branching nodes). This number is smaller than n.
The order of space complexity is of O(n).	

In tests with the Austrian surface water dataset the highest factor b was 26 for the Danube part of the rivers. From the Danube pour point 35,371 segments could be reached, 2,102 were connected with branching, from those only 130 had a nesting larger than 10, 26 larger than 20 -> $n+b^2$ is approximately 140,000 * c steps. Whereas a quadratic brute force method for calculating all of the rivers would be about $35{,}371^2/2$ steps which is more than 600 mio * c steps.

The algorithm was tested with the data of the three river basins. The output was evaluated through calculating the "JunctionValences" (c.f. 8.2.4.) As the Shreve numbers hold the number of headwaters above a certain river segment, tracing all the junctions of one basin, and reselecting those with a ValenceIn-value of 0, gives back the upper nodes for this certain river segment. Their count has to be the same as the Shreve number for this segment.

The same algorithm can be used to calculate the Scheidegger stream order by multiplying the output with 2 and consequential the Rzhanitsyn stream order by taking the logarithm on base 2 of the Scheidegger stream numbers - these were added as additional calculations to the Shreve.java Add-In, and only change the constant c4.

4.2.4.3 Pfafstetter algorithm

Because of a restricted time horizon and some open questions concerning the interoperability of Java and ArcObjects a Pfafstetter algorithm could not be implemented in Java for this thesis.

Data preconditions:

- Network pour points
- Catchment with network topology
- No backflows/circles
- The last segment before the pour point must not be braided
- Value field (accumulationArea - outcome of the AccumulateCatchments algorithm)

Description:

Steps:

Preprocessing:

- Create hashmaps for tracing (compare accumulateCatchments algorithm).
- Create attributes to hold the Pfafstetter code number for every calculated level.

Main Procedure:

1. Find the main river
    - Create an empty ArrayList "tributaries"
    - Create a linked list "fourLargest" to hold the four largest tributaries and their sizes
    - create a variable "smallest=0" to hold the size of the smallest of the four largest tributaries
    - Trace the watershed network from the pour point upwards - if there is more than one upper watershed choose the one with the bigger accumulationArea value and mark it as main river. Put the other watersheds that are not marked

as main in "tributaries" (id, aggregated area. Put the first four watersheds in fourLargest and save their smallest size in variable smallest. In the next steps compare the watersheds. accumulationArea of the tributary with smallest, if it is larger, the watershed in fourLargest with the size=smallest has to be deleted, the larger watershed will be appended at the end of "fourLargest". Value of "smallest" has to be decided anew.

2. Determine the 4 largest tributaries and interbasins
   - "fourLargest" holds the largest tributaries. According to their order in the list they get the codes 2, 4, 6 and 8.
   - During a new trace from the pour point the values for this Pfafstetter level are written to the table. The catchments marked main and all catchments not in fourLargest get the Interbasin numbers (1,3,5,7,9) according to their position in "tributaries" - from pourPoint to fourLargest(0) they get Pfafstetter code 1, between fourLargest(0) and fourLargest(1) they get Pfafstetter code 3, between fourLargest(1) and fourLargest(2) they get Pfafstetter code 5, between fourLargest(2) and fourLargest(3) they get Pfafstetter code 7, and after fourLargest(3) they get Pfafstetter code 9.

3. If there are more levels to be calculated repeat step 1 and 2.

Running time:

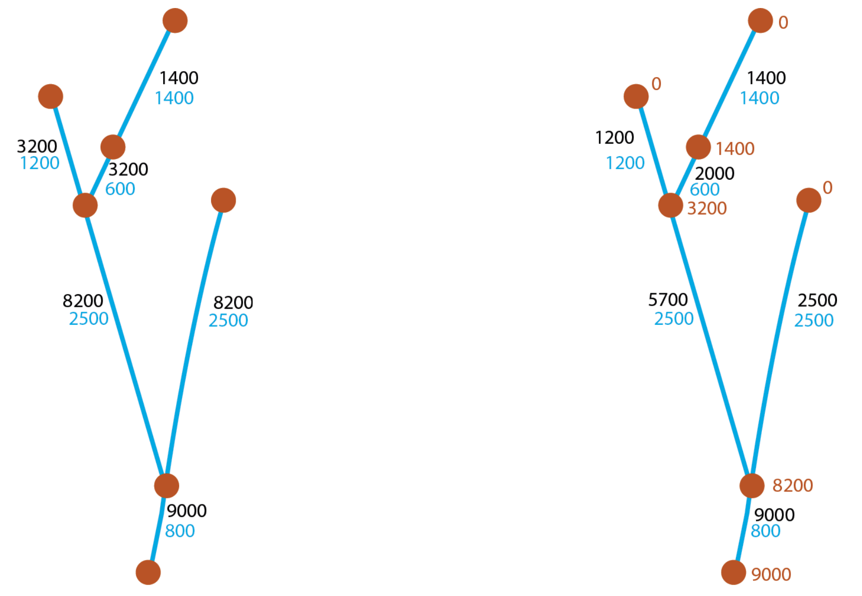
If data preparation (aggregating areas) is not drawn into conclusion, for every Pfafstetter level a main river trace and a code application trace has to be made. The main river trace does not trace the whole network but only the main rivers. To apply the Pfafstetter code a full trace has to be made that visits every catchment exactly once. Thus the time complexity depends on the number of catchments and the number of main rivers and the hierarchy complexity of the catchment. As the number of main rivers is less than the number of all rivers, the calculation of one level means at the maximum 2 runs through all rivers, thus $1*c_1* n + 1*c_2*n$ steps. The largest trace is always as large as the number of catchments, that means a complexity of $O(n*l)$ where n is the number of catchments and l is the number of calculated Pfaffstetter levels. l is usually beneath 100.

4.2.5 Algorithms for accumulating stream and catchment data

4.2.5.1 Accumulating rivers

Accumulation algorithms are necessary for the calculation of the arbolate sums (c.f. 3.1.2.1) for every segment of the river network or for parameters of water quality. The main aim is to calculate the value (e.g. sum of lengths of the river segment and all its upper segments or sum of some other numerical parameter) for all segments in the river network. Parallel rivers have to be kept in mind, because the calculation has to be made for all of them with avoidance of recalculating all parts that have already been calculated. AccumulateRivers.java uses the network structure of ArcGIS and calculates accumulation values for edges (river segments) and network junctions (wells, confluences ...) and keeps track of parallel stretches. The algorithm will only run properly if the prerequisite that the graph is acyclic is ensured. FindBackflows.java and FindCircles.java can both be used to make the corresponding tests.

There exists an Accumulate Attributes-tool from the Arc Hydro Tools [ARCHYDRO 2013] that fulfills a similar task. The outcome is not exactly the same, because there is a small difference in the concepts of accumulation as can be seen in Figure 4–13.



Accumulate Attributes tool (Arc Hydro) **AccumulateRivers.java**

Figure 4–13 Differences in the calculation of river accumulation values (blue numbers are the source values, black numbers are the accumulation values saved with the edges, brown are the accumulation values saved with the junctions)

AccumulateRivers.java calculates the values that are valid for the junctions to a field in the junctions data set and those valid for the lower endpoint of the river to the edge dataset. The Accumulate Attributes-tool of the Arc Hydro Tools only calculates one value (comparable to the value in the junctions of the AccumulateRivers.java).

The Accumulate Attributes-tool is most probably of quadratic order because it can be seen during processing that for the calculation of the accumulation value of an element in the network all upper network elements are traced.

Data preconditions (for AccumulateRivers.java):

- Network pour points
- River segments with network topology
- No backflows/circles
- The last segment before the pour point must not be braided
- Value field (measured or counted)

Description: AccumulateRivers.java calculates the accumulated value for rivers and junctions and saves it to an output field. The algorithm works based on a DSF with nested recursion. For very big datasets it has to be ensured that enough memory is available to hold the double values (the summed up river lengths) during the recursion.
In braided parts the accumulated value holds the sum of all upstream parts. Thus a correction has to take place when a braid ends, to not sum up the same value for a second (third ...) time.

Central methods of AccumulateRivers.java

```
public void accumulateValues(int ArcID){
  //set visited
  visited.put(ArcID,true);
  //get fromNode
  int fromNode = fromNodePerArc.get(ArcID);
  //get count of inflowingArcsPerNode
  int inflowingArcs = inflowingArcsPerNode.get(fromNode).size();
  int outCount=outflowingArcsPerNode.get(fromNode).size();
  if (inflowingArcs == 0){//Base case
    accumValueNode.put(fromNode, new Double(0d));
    //the arc length is already in the accumValue Map
  }else{//if the river with ArcID is no upper arc
    //get the inflowingArcsPerNode
    Iterator<Integer> it1;
    it1=inflowingArcsPerNode.get((Integer)fromNode).iterator();
    int arc;
    double sumValue=0;
    int upstreamFromNode;
    int temp;
    HashSet<Integer>nodeSet=new HashSet<Integer>();
    ArrayList<Integer>upstreamRivers=new ArrayList<Integer>();
    Map<Integer,Integer>localOrigNodeCnt=new
HashMap<Integer,Integer>();

    //for every inflowingArc do the following:
    while(it1.hasNext()){
      arc = (Integer)it1.next();
      //if the arc has never been visited
      if((visited.get(arc))==false){
        accumulateValues(arc);
        upstreamRivers.add(arc);
      //if it already has been visited
      }else{
        upstreamRivers.add(arc);
      }
      sumValue=sumValue+accumValue.get(arc).doubleValue();
      upstreamFromNode=fromNodePerArc.get(arc);
      // check if the origin Node is a braid node - if yes add it
      // to the nodeSet
      if(origNode.containsKey(upstreamFromNode)){
        if(nodeSet.add(upstreamFromNode)){
          localOrigNodeCnt.put(upstreamFromNode, 1);
        }else{
        //There is already the same Node in there
        // save value to correct the localOrigNodeCnt
          temp=localOrigNodeCnt.get(upstreamFromNode) + 1;
          localOrigNodeCnt.put(upstreamFromNode, temp);
        }
      }
    }//while end

    if(!nodeSet.isEmpty()){
      sumValue= sumValue + getValue(nodeSet, localOrigNodeCnt, ArcID,
              fromNode, outCount );
    }
    accumValueNode.put(fromNode, new Double(sumValue));
    double arcValue=0;
    if(outCount>1){
      for(int x : outflowingArcsPerNode.get(fromNode)){
        arcValue=sumValue+accumValue.get(x).doubleValue();
        accumValue.put(x, new Double(arcValue));
        visited.put(x, true);
      }
    }else {
      arcValue=sumValue+accumValue.get(ArcID).doubleValue();
      accumValue.put(ArcID, new Double(arcValue));
    }
  }
}

public double getValue(HashSet<Integer> nodeSet,
Map<Integer,Integer>localOrigNodeCnt, int ArcID, int fromNode, int
outCount){
  double sumValue=0;
  int upstreamFromNode;
  int temp;
  Set<Integer>upstreamOrigins = new HashSet<Integer>();
  HashSet<Integer>originsOut = new HashSet<Integer>();
  Map<Integer,Integer>nodeCount=new HashMap<Integer,Integer>();
```

```
    int factor;
    for(int f : nodeSet){//circle through all incoming nodes
      //direct nodes:
      upstreamFromNode=f;
      upstreamOrigins=origNode.get(upstreamFromNode);
      factor=localOrigNodeCnt.get(upstreamFromNode);
      //upper adjacent nodes:
      if(nodeCount.containsKey(upstreamFromNode)){
        temp=nodeCount.get(upstreamFromNode) + (factor);
        nodeCount.put(upstreamFromNode,temp);
      }else{
        nodeCount.put(upstreamFromNode, factor);
      }

      //origin nodes of upper adjacent nodes:
      if(!upstreamOrigins.isEmpty()){
        for(int e : upstreamOrigins){
          if(nodeCount.containsKey(e)){
            temp=nodeCount.get(e) + (factor);
            nodeCount.put(e,temp);
          }else{
            nodeCount.put(e, factor);
          }
        }
      }
    }

    //Now there exists a map nodeCount with all nodes
    // and their counts
    // - correct the node counts in origNodeCnt
    int upstreamOrigin;
    int cnt;
    for(Map.Entry<Integer, Integer> entry : nodeCount.entrySet()){
      upstreamOrigin=entry.getKey();
      cnt=entry.getValue();
      temp=0;
      //find out if branching is still open
      if(origNodeCnt.containsKey(upstreamOrigin)){
        temp=origNodeCnt.get(upstreamOrigin) - cnt;
      }
      if(temp==0){//branching ends here
        origNodeCnt.put(upstreamOrigin, 0);
        //System.out.println(upstreamOrigin + "ended");
      }else{//branching goes on
        temp=temp+outCount;
        originsOut.add(upstreamOrigin);
        origNodeCnt.put(upstreamOrigin, temp);
      }
    }

    if(!originsOut.isEmpty()){
      origNode.put(fromNode, originsOut);
    }
    //System.out.println("origOut"+originsOut);
    int cor=0;

    //Correct accumulation values
    // - correct direct nodes that appear more than once
    for(Map.Entry<Integer, Integer> entry :
  localOrigNodeCnt.entrySet()){
      upstreamOrigin=entry.getKey();
      cnt=entry.getValue();
      if (cnt>1){
        factor = cnt-1;
      sumValue=sumValue -
(accumValueNode.get(upstreamOrigin).doubleValue()
      *factor);
      nodeCount.put(upstreamOrigin, 1);
      for(int e: origNode.get(upstreamOrigin)){
        cor=nodeCount.get(e) - factor;
        nodeCount.put(e, cor);
      }
    }
  }

  // There may be still upper nodes that are
  // in the sum more often than once
  // a) find out their nesting
  // b) correct nearest node first

  HashMap<Integer,Integer>countCor=new HashMap<Integer,Integer>();
  Set<Integer>corSet=new HashSet<Integer>();
  while(!nodeCount.isEmpty()){
    temp=0;
    cnt=0;

    //a
    for(Map.Entry<Integer, Integer> entry : nodeCount.entrySet()){
      upstreamOrigin=entry.getKey();
      if(temp==0){
        temp=upstreamOrigin;
        cnt=entry.getValue()-1; //minus 1 because one has to be left
      }else{
        if(origNode.get(upstreamOrigin).contains(temp)){
          temp=upstreamOrigin;
          cnt=entry.getValue()-1; //minus 1 because one has to be
left
      }
    }
  }


  // now the temp-Node is corrected and not needed any longer
  // remove it from nodeCount map
  nodeCount.remove(temp);
  //b: if nearest node has any upper nodes
  // correct their count
  if(cnt>0){
    sumValue=sumValue-(accumValueNode.get(temp).doubleValue()*cnt);
    if (origNode.containsKey(temp)){
      //reduce upper nested nodes
      for(int e: origNode.get(temp)){
        if(nodeCount.containsKey(e)){
          cor=nodeCount.get(e)- cnt;
          if(cor<1){//happens in very complex braids (with
crossings)
            cnt=1-cor;

  sumValue=sumValue+(accumValueNode.get(e).doubleValue())*cnt;
            nodeCount.put(e, 1);
            if(!countCor.containsKey(e)){
              countCor.put(e,cnt);
              corSet.addAll(origNode.get(e));
            }
          }else{
            nodeCount.put(e, cor);
          }
        }
      }
      }
   }

   //at last correct the Values that were re-added if
   //they came from nested sets
   if(!countCor.isEmpty()){
      for(Map.Entry<Integer, Integer> entry : countCor.entrySet()){
         upstreamOrigin=entry.getKey();
         cnt=entry.getValue();
         if(corSet.contains(upstreamOrigin)){
            sumValue=sumValue -
               (accumValueNode.get(upstreamOrigin).doubleValue())*cnt;
         }
      }
   }
   return sumValue;
}
```

Running Time

Time complexity is comparable with the Shreve algorithm. If there are no braided rivers time complexity is O(n), only the method accumulateValues is executed and each river is visited only once. If the created nodeSet of braided rivers is not empty, then the getValue method is called to calculate the correction value. The complexity of the getValue method is $O(n_b^2)$, where n_b is the number of currently open braids. The more complex the braiding, the more open braids exist.

Time Complexity	Steps
constant	Initialize document and create the necessary fields and indices
c*n	Create hashMaps (inflowingArcsPerNode, outflowingArcsPerNode, fromNodePerArc,origNodeCnt, origNode) and fill them with data
c_1*n (+ c_2*n_b^2)	Fill hashMaps(accumValue, accumValueNode) - if there are no braided rivers this has a complexity of O(n) because every node is only visited once. If there are braided rivers an extra calculation has to be run.
O(n)	
Space Complexity	
Extra space for 6 hashmaps (hashmaps have to be designed large enough to avoid too much rehashing, thus for every hashmap a small extra amount of space has to be planned). Space complexity is of order O(n).	

4.2.5.2 Accumulating catchments

Comparable to the accumulation of rivers it may also be necessary to calculate accumulation parameters for catchments. For the calculation of the catchment accumulation first a network has to be created (e.g. with CreateWatershedNetworkGP which uses ArcGIS tools to create a network from existing NextDown data) where the network junctions represent the catchment centroids.
AccumulateCatchment.java accumulates the value of a chosen numerical field downstream. The accumulation values are therefore only calculated for the centroids and not for the network edges. In braided parts the accumulated value holds the sum of all upstream parts. Thus a correction has always to take place when a braid ends, to not sum up the same value for a second (third...) time.

A similar existing tool would be the Accumulate Attributes-tool from the Arc Hydro Tools (Arc Hydro Tools for ArcGIS 10.1, Beta version) already mentioned in 4.2.5.1.

Data preconditions:

- Catchment centroids with a network topology
- No backflows/circles
- There must be only one lowest sub-catchment per catchment (thus if there are more than one sub-catchments per catchment that empty into the sea, an artificial catchment has to be created that functions as lowest catchment for all of them)
- Value field (measured or counted)

Detail Description:

Again a DSF strategy proofed to be a quick way to calculate the accumulation of catchment values. Compared with the calculation of river accumulation values, the catchment accumulation only sums up the values of the centroids (which form the network nodes).

```
public void accumulateValues(int nodeID){
  //set visited
  visited.put(nodeID,true);

  //get upper Nodes
  if(!upstreamAdjacentNodes.containsKey(nodeID)){
    System.out.println("NodeID: " + nodeID + " upper ");
```

```
      //do nothing because value is already in accumValueNode
    }else{ //if the node is no upper node
      //get the upper nodes
      System.out.println("NodeID: " + nodeID + " ***** ");
      ArrayList<Integer>upstreamNodes =
upstreamAdjacentNodes.get(nodeID);
      double sumValue=accumValueNode.get(nodeID);
      Map<Integer,Integer>nodeCountMap=new HashMap<Integer,Integer>();
      //Set<Integer>upstreamNodeSet;
      for(int upstreamNodeID : upstreamNodes){
        if(!visited.containsKey(upstreamNodeID)){

          accumulateValues(upstreamNodeID);
          System.out.println("test2");
        }

        //accumulate Values
        sumValue+=accumValueNode.get(upstreamNodeID);
        //get braid-node-set of upstream node
        if(braidNodes.containsKey(upstreamNodeID)){
            nodeCountMap=
              countNodes(braidNodes.get(upstreamNodeID),nodeCountMap);
        }
        //if upstreamNode is also a braid node than add it
        if(downstreamAdjacentNodeCnt.containsKey(upstreamNodeID)){
          nodeCountMap=countNodes(upstreamNodeID,nodeCountMap);
        }
      }//end for
      //now have a look at the nodeCountMap
      //there is only to do something if it is not empty
      if(!nodeCountMap.isEmpty()){
        // get outCount (=number of next Down nodes)
        int outCount=1;
        if(downstreamAdjacentNodeCnt.containsKey(nodeID)){
          outCount=downstreamAdjacentNodeCnt.get(nodeID);
        }
        //adapt downstream adjacent count
        braidNodes.put(nodeID, updateNodeCounts(outCount,
nodeCountMap));
        //- if there are nodes in the nodeCountMap with a count > 1
        //the sumValue has to be corrected
        sumValue = sumValue + correctValue(nodeCountMap);
      }
      accumValueNode.put(nodeID, sumValue);
    }
}
```

```
private double correctValue(Map<Integer,Integer>nodeCountMap){
    double correctionValue=0d;
    Integer upstreamOrigin;
    int count;
    int temp=0;
    //int factor;
    HashMap<Integer,Integer>countCor=new HashMap<Integer,Integer>();
    int tempcnt;
    Set<Integer>corSet=new HashSet<Integer>();
    int cor=0;
    System.out.println(" Correct Values " + nodeCountMap);
    while(!nodeCountMap.isEmpty()){
      temp=0;
      count=0;
      for(Map.Entry<Integer, Integer> entry : nodeCountMap.entrySet()){
        upstreamOrigin=entry.getKey();
        if(temp==0){
          temp=upstreamOrigin.intValue();
          count=entry.getValue()-1; //minus 1 because one has to be
left
        }else{//now check if node is next upper node
          if(braidNodes.containsKey(upstreamOrigin)){
            if(braidNodes.get(upstreamOrigin).contains(temp)){
              temp=upstreamOrigin.intValue();
              count=entry.getValue()-1; //minus 1 because once it has
to
                                //be counted
            }
          }
        }
      }
      // now the temp-Node is corrected and not needed any longer
      // remove it from nodeCount map
      nodeCountMap.remove(temp);
      //b: if nearest node has any upper nodes
      // correct their count
      if(count>0){
        correctionValue=correctionValue   -
                (accumValueNode.get(temp).doubleValue()*count);
        if (braidNodes.containsKey(temp)){
          //reduce upper nested nodes
          for(int e: braidNodes.get(temp)){
            if(nodeCountMap.containsKey(e)){
              cor=nodeCountMap.get(e)- count;
              if(cor<1){//happens in very complex braids (with
crossings)
                count=1-cor;
                correctionValue=correctionValue+
                (accumValueNode.get(e).doubleValue())*count;
                nodeCountMap.put(e, 1);
                if(!countCor.containsKey(e)){
                  countCor.put(e,count);
                  if (braidNodes.containsKey(e)){
                    corSet.addAll(braidNodes.get(e));
                  }
                }else{
                  //tempcnt=nodeCountMap.get(e)+count;
                  tempcnt=countCor.get(e)+count;
                  countCor.put(e, tempcnt);
                }
              }else{
                nodeCountMap.put(e, cor);
              }
            }
          }
        }
      }
    }
    if(!countCor.isEmpty()){
      for(Map.Entry<Integer, Integer> entry : countCor.entrySet()){
        upstreamOrigin=entry.getKey();
        count=entry.getValue();
        if(corSet.contains(upstreamOrigin)){
          correctionValue=correctionValue -
            (accumValueNode.get(upstreamOrigin).doubleValue())*count;
        }
      }
    }
    return correctionValue;
  }
```

```
/**
 *
 * @param nodeSet
 * @param nodeMap
 * @return
 */
private Map<Integer, Integer>countNodes(Set<Integer>nodeSet,
Map<Integer,Integer>nodeMap){
  Integer count;
  for(Integer node:nodeSet){
    if(nodeMap.containsKey(node)){
      count=nodeMap.get(node)+1;
      nodeMap.put(node, count);
    }else{
      nodeMap.put(node, 1);
    }
  }

  return nodeMap;
}


/**
 *
 * @param node
 * @param nodeMap
 * @return
 */
private Map<Integer, Integer>countNodes(Integer node,
Map<Integer,Integer>nodeMap){
  Integer count;
  if(nodeMap.containsKey(node)){
    count=nodeMap.get(node)+1;
    nodeMap.put(node, count);
  }else{
    nodeMap.put(node, 1);
  }
  return nodeMap;
}


/**
 *
 * @param outCount
 * @param nodeMap
 * @return Set<Integer> originsOut
 */
private Set<Integer> updateNodeCounts(Integer outCount,
Map<Integer,Integer>nodeMap){
  Integer upstreamOrigin;
  int count;
  int temp;
  Set<Integer>originsOut = new HashSet<Integer>();
  for(Map.Entry<Integer, Integer> entry : nodeMap.entrySet()){
    upstreamOrigin=entry.getKey();
    count=entry.getValue();
    temp=0;
    //find out if branching is still open
    temp=downstreamAdjacentNodeCnt.get(upstreamOrigin) - count;
    if(temp==0){//branching ends here
      downstreamAdjacentNodeCnt.put(upstreamOrigin, 0);
      //System.out.println(upstreamOrigin + "ended");
    }else{//branching goes on
      temp=temp+outCount;
```

```
        originsOut.add(upstreamOrigin);
        downstreamAdjacentNodeCnt.put(upstreamOrigin, temp);
      }
   }
   return originsOut;
}
```

Time Complexity	Steps
constant	Initialize document and create the necessary fields and indices
c*n	Create hashMaps (visited, accumValueNode, upstreamAdjacentNodes, downstreamAdjacentNodeCnt, braidNodes) and fill them with data
c_1*n (+ c_2*n_b^2)	Fill hashMaps(accumValueNode) - if there are no braids in the catchment network it has a complexity of O(n) because every node is only visited once. If there are braids an extra calculation has to be run with a time complexity of (O n_b^2) and a large constant c_2.
O(n)	
Space Complexity	
Extra space for 5 hashmaps (hashmaps have to be designed large enough to avoid too much rehashing, thus for every hashmap a small extra amount of space has to be planed). Order of space complexity is O(n).	

4.2.5.3 Aggregating catchments

For special questions and for cartographic purposes it is necessary to make a special kind of accumulation namely that of geometry of the catchment polygons. This is called an aggregation. The AggregatingCatchments.java follows the same idea as the algorithm to accumulate catchment values. The starting point is a network where the junctions represent the catchment centroid. Via a relationship between the junctions and the catchment polygons it is possible to address the related polygons during network query and join polygons using topological operations.

AggregatingCatchments.java aggregates the areas of all upstream catchments into a new FeatureClass. For every catchment a new polygon is created that embraces all the upper watershed polygons. At a last step polygons are sorted by their size and written into a new dataset, starting with the biggest. This happens to make sure, that all polygons are visible.

For large catchments this can be a memory intensive progress.

Data preconditions:

1. Catchment centroids and edges with network topology (see 8.2.5)
2. No backflows/circles (check WatershedNetwork for backflows/circles - see 4.2.2.1, 4.2.2.2)
3. No multipart features in the catchments (implementation does not yet deal with multipart features) (see therefore 8.2.6)

Description: Despite the resemblance to the AccumulateCatchments algorithm the AggregateCatchments algorithm had to be implemented with hindsight to the larger data volume caused by the geometry. Thus not all geometry could be kept in memory, but classes with already calculated areas were released to free some memory. If the area is needed again (which happens in case of bifurcations), then the geometry is called from the already created dataset in the geodatabase.

```
public void aggregateAreas1(int nodeID) throws AutomationException,
IOException{
  // After 500 polygons save edits
  countEdits++;
  if(countEdits>500){
    editor.stopEditing(true);
    editor.startEditing(FCPoly.getWorkspace());
    while(editor.getEditState()!= esriEditState.esriStateEditing){
      //wait until editing is possible again
    }
    countEdits=0;
  }
  // If node is visited mark it true
  visited.put(nodeID,true);

  //Base case - the node is an upper node
  if(!upstreamAdjacentNodes.containsKey(nodeID)){
    try{
      IFeature returnFeature = pFCOut.createFeature();

  returnFeature.setShapeByRef(FCPoly.getFeature(nodeID).getShape());
      returnFeature.setValue(2,nodeID);
      returnFeature.store();
      //With every new feature the objectnumber rises by one
      ++objectnumber;
```

```
        System.out.println(objectnumber);
        //To avoid later on a query for the new objectnumber it is put
        //in a map
        objectIDMap.put(nodeID,objectnumber);
      }catch (Exception e){
        e.getStackTrace();
        System.out.println("Could not store upper feature " + nodeID);
      }
    //other cases: the node is not an upper node
    }else{
      try{
        //get all upper nodes
        ArrayList<Integer>upstreamNodes =
upstreamAdjacentNodes.get(nodeID);
        // Create an IGeometry as GeometryBag to hold all Polygons
        IGeometry geomBag = new GeometryBag();
        // Set its spatial Reference
        geomBag.setSpatialReferenceByRef(spatialReference);
        // QI to a IGeometryCollection for its methods
        IGeometryCollection geometryColl =
(IGeometryCollection)geomBag;
        // addPoly will be every Polygon that
        IPolygon addPoly;
        for(int upstreamNodeID : upstreamNodes){
          addPoly=null;
          if(!visited.containsKey(upstreamNodeID)){
            //upstreamFeature = FCPoly.getFeature(upstreamNodeID);
            //upperFeature = aggregateAreas1(upstreamFeature);
            //IGeometry geom = upperFeature.getShape();
            //addPoly = (IPolygon)geom;
            aggregateAreas1(upstreamNodeID);
            IGeometry geom = pFCOut.getFeature(objectIDMap.get
                               (upstreamNodeID)).getShape();
            addPoly=(IPolygon)geom;
          }else{
            IGeometry geom = pFCOut.getFeature(objectIDMap.get
                          (upstreamNodeID)).getShape();
            addPoly=(IPolygon)geom;
          }

          if (addPoly!=null){
            geometryColl.addGeometry(addPoly, null, null) ;
          }else{
            System.out.println("Could not add polygon to collection");
          }
        }//end for

        IGeometry geom = FCPoly.getFeature(nodeID).getShape();
        if(geom instanceof Polygon){
          geometryColl.addGeometry(geom, null, null) ;
        }

        //get final Polygon
        IPolygon finalPolygon = getFinalPolygon(geometryColl);
        if(finalPolygon!=null){
          if(!finalPolygon.isEmpty()){
            IFeature returnFeature = pFCOut.createFeature();
            returnFeature.setShapeByRef(finalPolygon);
            returnFeature.setValue(2,nodeID);
            returnFeature.store();
            objectnumber++;
            objectIDMap.put(nodeID,objectnumber);
          }
        }else{
          System.out.println("Could not create new Polygon from
                                      inputs");
          Thread.sleep(10000);
        }
      }catch (Exception e){
        e.printStackTrace();
        System.out.println(e.getMessage());

      }

    }
  }

  private IPolygon getFinalPolygon(IGeometryCollection geomCollection)
  throws AutomationException, IOException{
    try{

      ITopologicalOperator2 finalPoly=new Polygon();

      finalPoly.setIsKnownSimple(false);
      finalPoly.simplify();
    finalPoly.constructUnion((IEnumGeometry)geomCollection);

    return (IPolygon)finalPoly;


  }catch(Exception e){
    e.printStackTrace();
    System.out.println("getFinalPolygon: " + e.getLocalizedMessage());
    return null;
  }

}
```

Time Complexity	Steps
constant	Initialize document and create the necessary fields and indices
c*n	Create hashMaps (visited, upstreamAdjacentNodes) and fill them with data
c_1*n	Calculate new polygons and save them in the output feature class in O(n) because every node is only visited once. There is a large constant c_1 to get the already calculated features from the output, which means that, even though it is still linear order, for every additional polygon a fixed number of steps has to be calculated that slow down the calculation.
O(n)	
Space Complexity	
Extra space for hashmaps (hashmaps have to be designed large enough to avoid too much rehashing, thus for every hashmap a small extra amount of space has to be planned). For the output FeatureClass there is more space needed than for the input FeatureClass. The amount of space depends on the depth of the catchment hierarchy (the deeper the hierarchy the more often the same outer catchment boundary is copied). $O(n) <$ order of space complexity $< O(n^2)$	

5 Examples of use for the algorithms

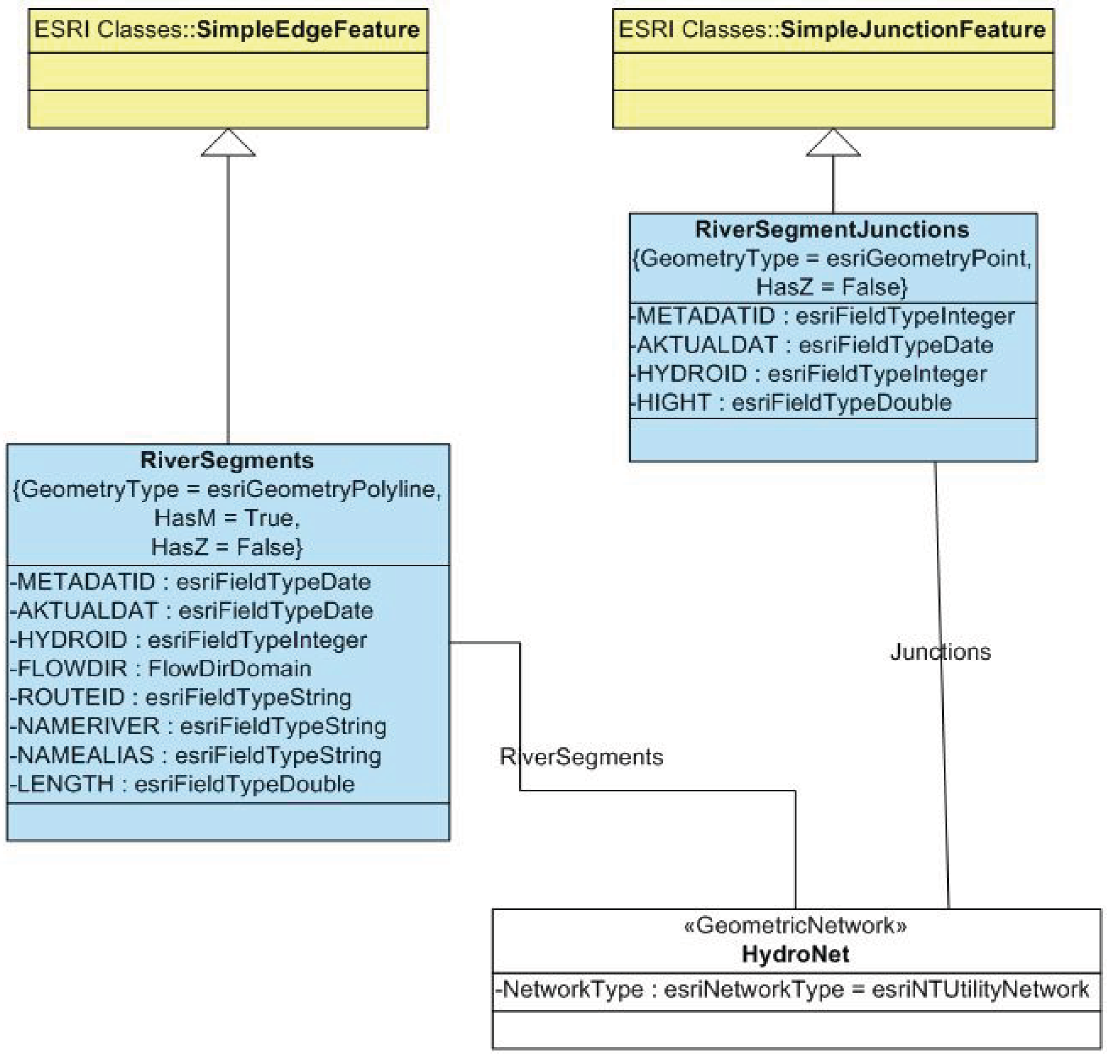
The Environment Agency Austria (Umweltbundesamt) was founded in 1985 and established and since 2003 maintains the Austrian surface water dataset for reporting for the Federal Ministry of Agriculture, Forestry, Environment and Water Management. The dataset is a centrally available combination of the nine federal states' river datasets (including the lakes and catchments) and contains the main rivers of the neighboring states, which affect the Austrian rivers or are affected by the Austrian rivers (i.e. the main incoming and outflowing rivers and the rivers along the country's frontier). In combination with a fast amount of hydrological, biological, ecological, geological, and administrative information the river dataset for reporting is a potent "tool" to model the relationship of surface waters to other areal, linear or point data.

The object oriented data model was specially developed to comply with the needs of reporting for the EU WRRL and contains all the rivers with a catchment size of more than 10 km^2, including their branching out reaches. Lakes are included when larger than 0.5 km^2, catchments must have at least 1 km^2. The newest versions (from version 7) additionally include smaller rivers as needed for operations done for the EU Floods Directive 2007/60/EC[22].

To get an impression of the numbers of items in the dataset, version 8 contains:
>35,000 river segments
6,225 routes (incl. calibration points)
> 88,000 route segments (defined on routes)
> 9,000 river water bodies (defined on routes)
> 40,000 catchments

The original data model already holds a geometric network for the rivers. The structure was slightly adapted for this thesis and can be depicted from Figure 5–1

[22] The directive requires member states to first carry out a preliminary assessment by 2011 to identify the river basins and associated coastal areas at risk of flooding. For such zones flood risk maps have to be created by 2013 and flood risk management plans have to be prepared by 2015. c.f. http://ec.europa.eu/environment/water/flood_risk/index.htm



ESRI Classes::SimpleEdgeFeature
ESRI Classes::SimpleJunctionFeature
RiverSegmentJunctions
{GeometryType = esriGeometryPoint, HasZ = False}
-METADATID : esriFieldTypeInteger
-AKTUALDAT : esriFieldTypeDate
-HYDROID : esriFieldTypeInteger
-HIGHT : esriFieldTypeDouble
RiverSegments
{GeometryType = esriGeometryPolyline, HasM = True, HasZ = False}
-METADATID : esriFieldTypeDate
-AKTUALDAT : esriFieldTypeDate
-HYDROID : esriFieldTypeInteger
-FLOWDIR : FlowDirDomain
-ROUTEID : esriFieldTypeString
-NAMERIVER : esriFieldTypeString
-NAMEALIAS : esriFieldTypeString
-LENGTH : esriFieldTypeDouble
Junctions
RiverSegments
«GeometricNetwork»
HydroNet
-NetworkType : esriNetworkType = esriNTUtilityNetwork

Figure 5–1 Object model river network

The data was used for testing purposes and provided a valuable means for this work. In the following sections some outputs of the calculations are shown.

With the CreateWatershedNetworkGP.java (8.2.5) a similar object structure was created from the watersheds (see Figure 5–2 Object model of the watershed network). During the calculation also a one-to-one relationship between centroids and watersheds is created.

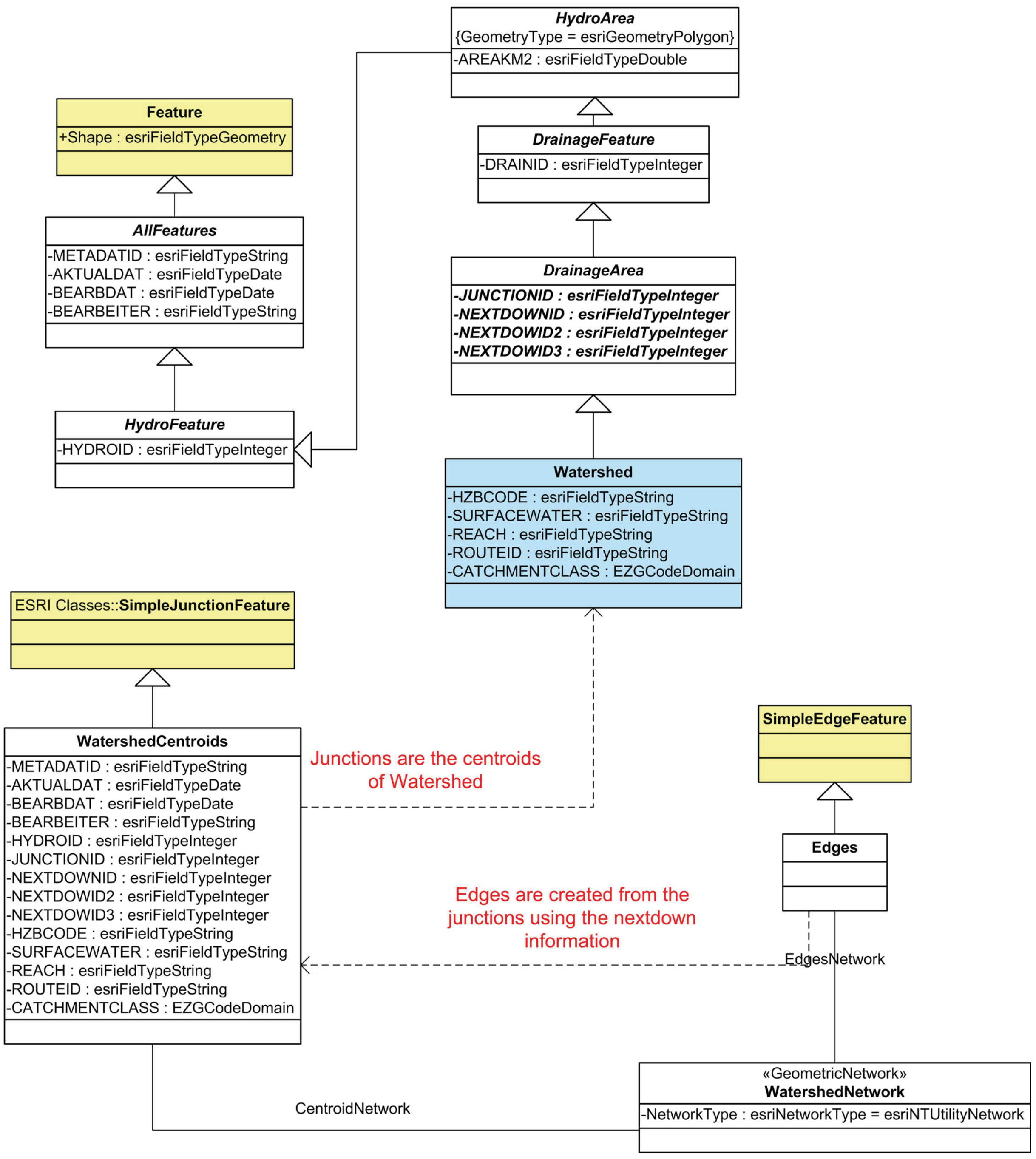

HydroArea
{GeometryType = esriGeometryPolygon}
-AREAKM2 : esriFieldTypeDouble
Feature
+Shape : esriFieldTypeGeometry
DrainageFeature
-DRAINID : esriFieldTypeInteger
AllFeatures
-METADATID : esriFieldTypeString
-AKTUALDAT : esriFieldTypeDate
-BEARBDAT : esriFieldTypeDate
-BEARBEITER : esriFieldTypeString
DrainageArea
-JUNCTIONID : esriFieldTypeInteger
-NEXTDOWNID : esriFieldTypeInteger
-NEXTDOWID2 : esriFieldTypeInteger
-NEXTDOWID3 : esriFieldTypeInteger
HydroFeature
-HYDROID : esriFieldTypeInteger
Watershed
-HZBCODE : esriFieldTypeString
-SURFACEWATER : esriFieldTypeString
-REACH : esriFieldTypeString
-ROUTEID : esriFieldTypeString
-CATCHMENTCLASS : EZGCodeDomain
ESRI Classes::SimpleJunctionFeature
SimpleEdgeFeature
WatershedCentroids
-METADATID : esriFieldTypeString
-AKTUALDAT : esriFieldTypeDate
-BEARBDAT : esriFieldTypeDate
-BEARBEITER : esriFieldTypeString
-HYDROID : esriFieldTypeInteger
-JUNCTIONID : esriFieldTypeInteger
-NEXTDOWNID : esriFieldTypeInteger
-NEXTDOWID2 : esriFieldTypeInteger
-NEXTDOWID3 : esriFieldTypeInteger
-HZBCODE : esriFieldTypeString
-SURFACEWATER : esriFieldTypeString
-REACH : esriFieldTypeString
-ROUTEID : esriFieldTypeString
-CATCHMENTCLASS : EZGCodeDomain
Junctions are the centroids of Watershed
Edges
Edges are created from the junctions using the nextdown information
EdgesNetwork
«GeometricNetwork»
WatershedNetwork
-NetworkType : esriNetworkType = esriNTUtilityNetwork
CentroidNetwork

Figure 5–2 Object model of the watershed network

The water network gives the possibility to trace watersheds with efficient speed and is also necessary to calculate aggregations. At the moment the ideal relationship of one river per catchment is not established correctly, in most parts of the network there are more catchments than rivers in the reporting system, it is not possible to use the river data to trace the watersheds. This can be seen in Figure 5–3. The network structure of the watersheds is established using existing next down relationships (this means that there exists a field or some fields in the table that hold the unique watershed number(s) of the downstream watershed(s)). Many of the next down values were manually entered into the watershed tables; only where the rivers correspond with the watersheds existing routines from Arc Hydro Tools could be used to calculate those next down values.

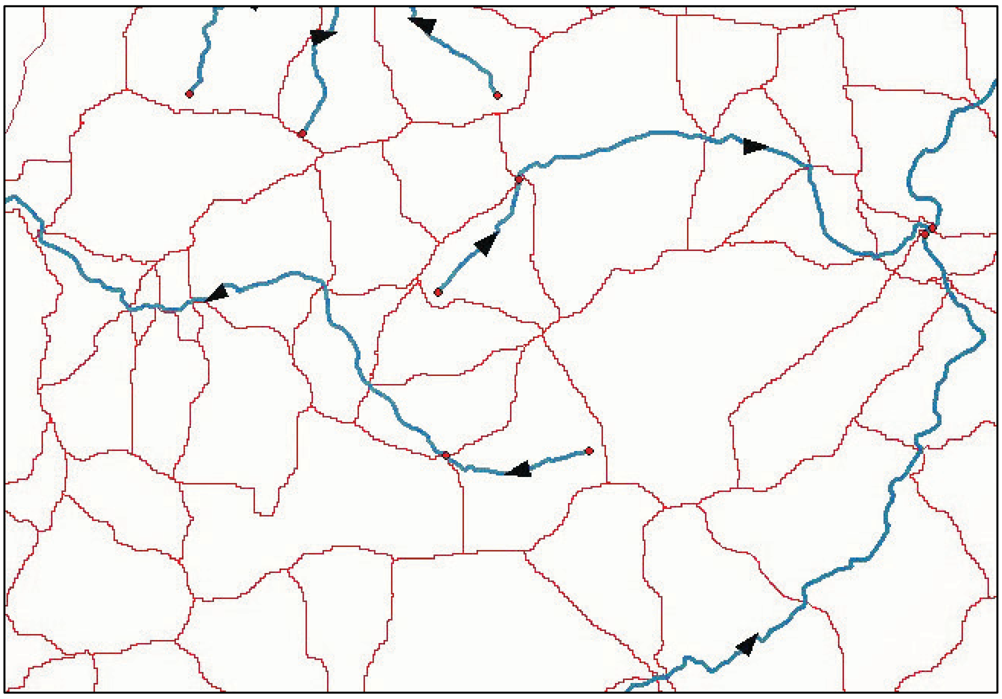


Figure 5–3 Example of the spatial relationship between rivers and watersheds

The following figure shows part of the established watershed network for the Austrian catchments.

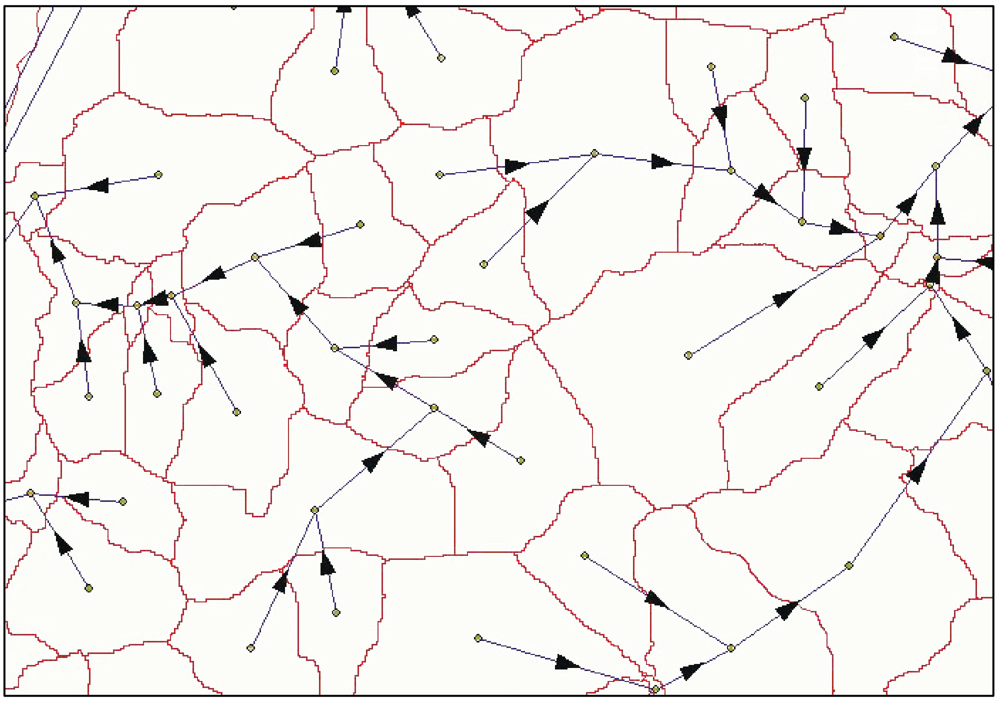


Figure 5–4 Example of an established catchment network

Strahler (Strahler.java), Shreve, Scheidegger and Rzhanitsyn (Shreve.java) numbers can be calculated for the entire Austrian network for reporting in reasonable time (on my old laptop neither calculation took longer than 10 min).

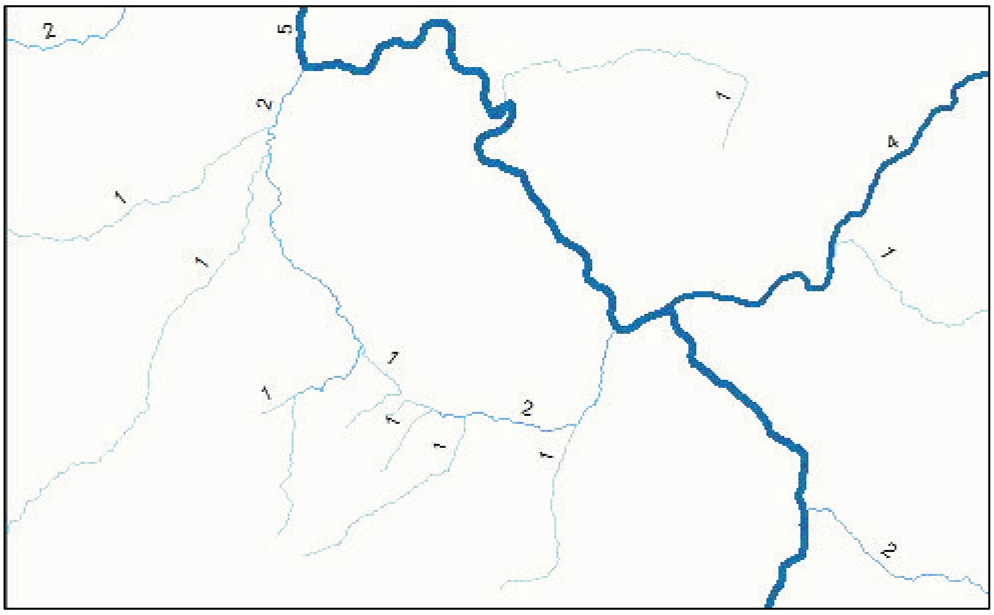


Figure 5–5 Strahler numbers

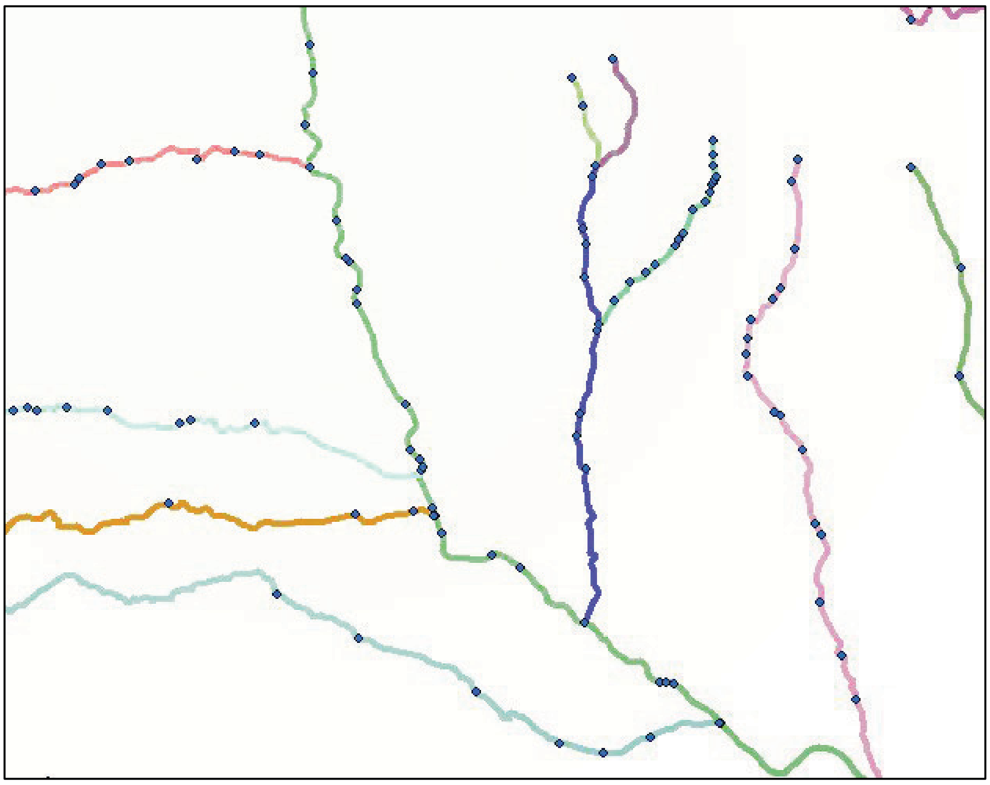


Figure 5–6 Strahler segments

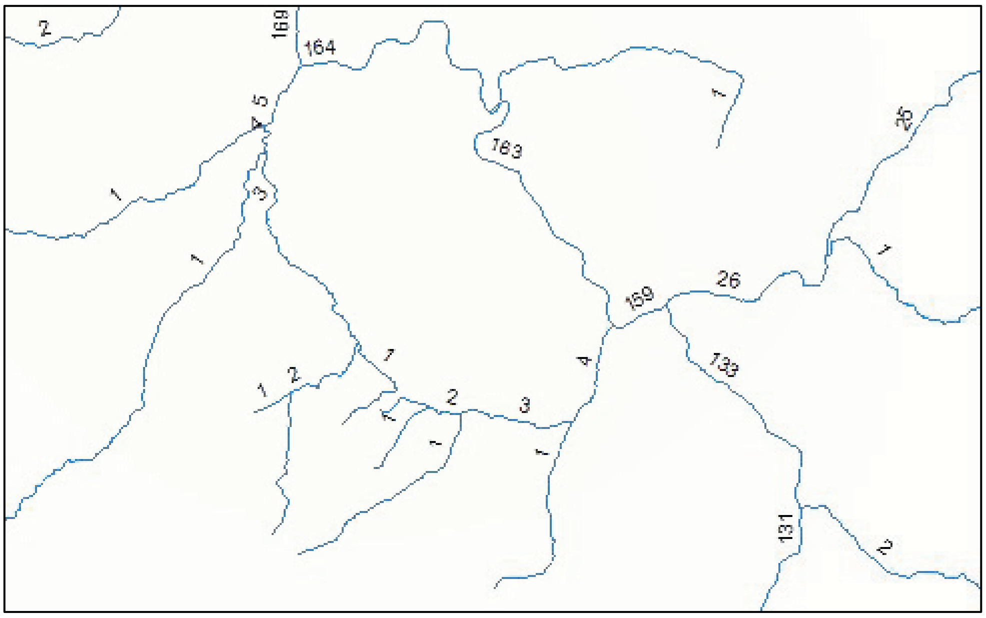


Figure 5–7 Shreve numbers

All the accumulation and aggregation algorithms were tested as well. The accumulation works for both rivers and catchments and was tested against the output of manual samples at many locations of the network - 120 samples at upper, middle and lower parts of the network were consistent with the calculated values. The running time of the algorithm was compared with those of the Accumulate Attributes-tool of the Arc Hydro Tools for all rivers and all of the catchments in the Austrian surface water dataset for reporting (for the differences between AccumulateRivers.java and Accumulate Attributes from Arc Hydro Tools see also 4.2.5.1)

River accumulation (35371 rivers)	
AccumulateRivers.java	Accumulate Attributes (Arc Hydro Tools)
12 min (4,2% or more than 23 times faster)	284 min (100%)

Table 11 Time comparison for river data accumulation

Catchment accumulation (40245 catchments)	
AccumulateCatchments.java	Accumulate Attributes (ArcHydroTools)
15 min (4,8% or more than 20 times faster)	314 min (100%)

Table 12 Time comparison for catchment data accumulation

Both times the runtime of the new tools compared favorably against the runtime of the ArcHydro Accumulate Attributes-tool as can be seen in Table 11 and Table 12. It has to be stated, that for the calculation of the catchment data there has to be created a catchment network once. The calculation took another seven minutes on my laptop.

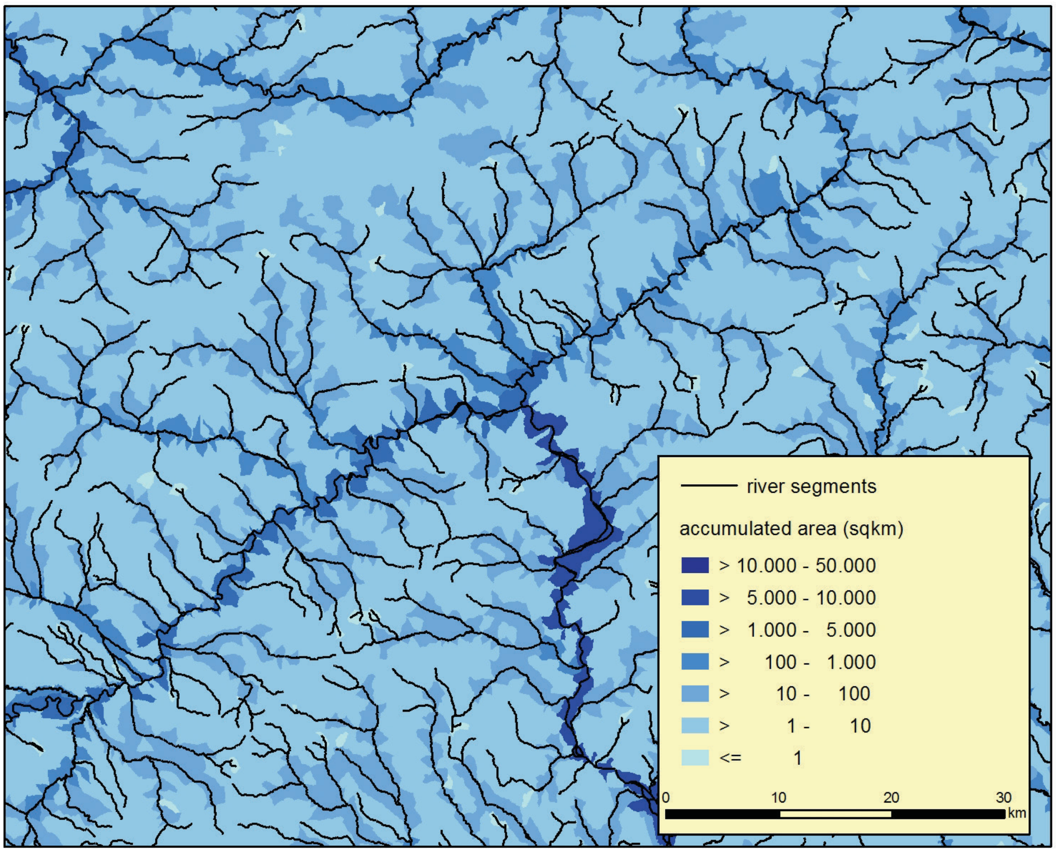


Figure 5–8 Example of accumulated area at the confluence of the rivers Mur and Mürz.

The aggregation of geometry is memory consuming and works best with a high performance computer, with my old laptop I could easily calculate up to 15,000 watersheds.

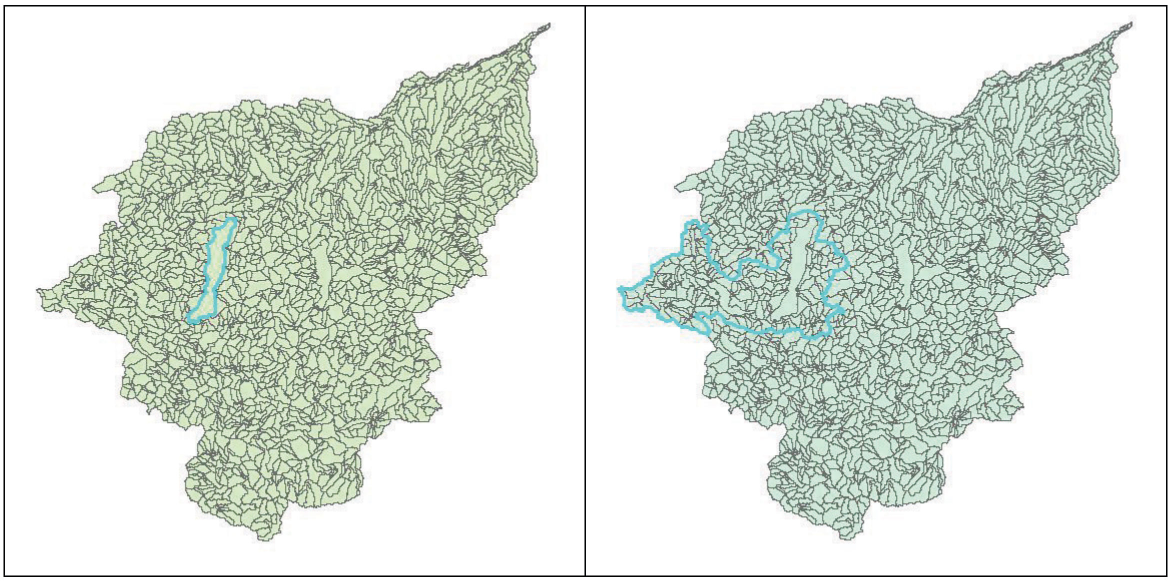


Figure 5–9 Example of watershed aggregation in the river Traun catchment.

Figure 5–9 shows on the left side the original watersheds on the right side the aggregated version. After aggregation for every watershed area a complete watershed (containing all incoming surface waters) is available. In the example the lake Traunsee is highlighted on the left side. On the right side the complete catchment of lake Traunsee can be seen.

The different discussion groups for ArcGIS mention a problem that concerns some of the used classes. Out of some inexplicable process the program may not be able to address the output FeatureClass any longer, even if there is still enough memory available. This results in an Automation error, which stops the calculation. On my laptop this only happened for catchment sizes larger than about 15,000 catchments but not every time at the same catchment, sometimes the dataset could be nearly calculated as a whole (best output was 37,000 catchments).

A workaround to this at the moment not solvable problem (because I have no possibility to change the basic ArcObjects classes) is to split the dataset in manageable parts of not larger than 15,000 and merge them together at the end.

6 Summary and future prospects

Algorithms for river and catchment networks embrace a wide field. In chapter 3 a list for algorithms for these kinds of data was compiled. The following chapters were used to create implementations of a selection of questions of this compiled list. I tried to cover three areas of interest and therefore created/implemented algorithms for establishing the necessary basics (like individual flow direction, finding circles and so on), for ordering (like the implementation of Strahler algorithms, creation and implementation of Shreve algorithm) and for accumulation (creation and implementation of accumulation algorithms for river and catchment data) and aggregation (creation and implementation of aggregation algorithm for catchment geometries).

Some conclusions I could draw from this work are:

- There are algorithms available to cover parts of the compiled list, like the algorithms by Lanfear [LANFEAR 1990] and by Gleyzer [GLEYZER 2004]. These had to be implemented for use in GIS.

- Some other numbering algorithms were based on the ideas of Lanfear's [LANFEAR 1990] and Gleyzer's [GLEYZER 2004] algorithms and have runtimes that are linear for not braided networks.

- For the aggregation of river data and catchment data no algorithms could be found in literature, but tools exist to solve these questions (e.g. Arc Hydro Tools). Testing these tools gave an idea on how their algorithms work. Taking the time for several runs and averaging the output showed that the tools need a long time to calculate aggregation values.
  As many of the problems concerning networks are in a way comparable, I tried to find a way to make aggregation about as quickly as numbering of rivers. Depth first search (already used in the numbering algorithms) proved to be a really good starting point for the creation of the network aggregation algorithms and guarantees a linear running time (at least for those parts of the diverse

network structures that are not braided). In chapter 5, where I present some outcomes, Table 11 and Table 12 show that the runtimes of my newly created tools, although still long, are much shorter than those of the tool used for comparison (river accumulation 23 times faster, catchment accumulation 20 times faster).

- Java with its collection framework provides many data structures that are necessary (esp. map, list, set, hashmap, array). There was no need to create more specialised data structures for my own purposes - this only happened when an existing algorithm with a specified data structure was implemented (e.g. the Kosaraju Sharir algorithm for finding circles in the dataset needed its own implementation of an adjacency list).

A big proportion of the runtime of the algorithm is taken up by the preparation of the data for the use in the central algorithms that do the numbering or aggregating. The data has to be read out from the GIS system to hashmaps and arrays after a tool is started. Those hashmaps and arrays are similar for the different algorithms, therefore it may be wise for future work to establish them permanently in the GIS system instead of calculating the data anew at every run.

- During the implementation of the algorithms a lot of unexpected things occurred at the "bridge" between ArcObjects and Java that were not always explainable (not even by Java or ArcGIS experts). Thus work with ArcObjects did not prove easy and some of the underlying structures may need some revision that is not in my hands.

There are still a number of questions I could not pack into an algorithm in the frame of this thesis but which I plan to work on in the near future:

- An implementation of the Pfafstetter-coding algorithm.
- Some work for the aggregation of values on Routes, and definitely
- Some work in the geometry section, esp. to define and calculate the length of catchments (polygon triangulation algorithms offer a linear approach on which such a calculation could be based) w

- Finally the algorithms should be made more modular and be transferred to the newest GIS version.

7 Literature

7.1 Data

[AUSTRIAN RIVER NETWORK 2012] Austrian River Network for Reporting, version 8, Vienna: Austrian Environment Agency for the Austrian Federal Ministry for Agriculture, Forestry, Environment and Water Economy.
(running waters > 10 km² catchment size required under the WFD as well as on running waters with a catchment size < 10 km² which are relevant for the implementation of the EFD)

7.2 Software

[ARCHYDRO 2013] Arc Hydro Tools (Version 10.1 Beta). 2013.

http://forums.arcgis.com/forums/88-ArcHydro

[ARCMAP 2012] ArcGIS ArcMap (Version10.1 SP1). ESRI, 2012.

[ECLIPSE 2012] Eclipse IDE for Java Developers (Version 1.4.20120213-0813) (Indigo Service Release 2), Eclipse Foundation, 2012.
including: ArcGIS Engine Plug-in (Version 10.1.1.3143), ESRI, 2006.

[JAVA SE 6] Java™ Platform, Standard Edition 6. Oracle.

7.3 Online Literature

[BDS 2013] The British Dam Society http://www.britishdams.org/about_dams/embankment.htm

[BRITTON 2002] Britton, P. (2002): Review of Existing River Basin Coding Systems. WFD GIS Working Group, October 2002. https://circabc.europa.eu/sd/d/3ee91b56-b3ba-4a14-8cf8-844f1c9b648f/Coding%20REVIEW-V1.8.doc

[COMMITTEE ON LARGE DAMS 2012] Austrian committee on Large Dams in the Austria Federal Ministry for Agriculture, Forestry, Environment and Water Economy.
http://www.naturgefahren.at/article/archive/25202

[*DUMKE 2001*] Dumke, R.: Einführung Algorithmen und Datenstruktur. WS 2000/2001. Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, AG Softwaretechnik. http://ivs.cs.uni-magdeburg.de/sw-eng/agruppe/lehre/ead.shtml

[EC WATER] http://ec.europa.eu/environment/water/index_en.htm

[IGH 2013] International Glossary of Hydrology: http://www.hydrologie.org/glu/HINDENT.HTM

[INTERNATIONAL RIVERS 2013] http://www.internationalrivers.org

[OTTMANN 2009] Ottmann, T.: Algorithmentheorie. Podcast.

[EPPSTEIN 1996] Eppstein, David: ICS 161: Design and Analysis of Algorithms, Lecture Notes. University of California, Irvine. http://www.ics.uci.edu/~eppstein/161/960201.html

[NETLEXIKON] netlexikon http://www.lexikon-definition.de

[NIST] Dictionary of Algorithms and Data Structures. http://xlinux.nist.gov/dads/

[PREISS 1998] Preiss, Bruno R.: Data Structures and Algorithms with Object-Oriented Design Patterns in Java. http://www.brpreiss.com/books/opus5/
c.f. [PREISS 1999].

[TRACER] Austria Federal Ministry for Agriculture, Forestry, Environment and Water Economy, forestry, environment: Markierungsversuche in der Hydrologie und Hydrogeologie.
www.bmlfuw.gv.at/lmat/wasser/wasseroesterreich/wasserkreislauf/hydrographische_daten/Markierversuche_GW.html

[VORNBERGER 2009] Vornberger, O.: Algorithmen WS 2008/2009. Podcast.

[WEISSTEIN] Weisstein, Eric: MathWorld--A Wolfram Web Resource: http://mathworld.wolfram.com/

[ZHAN 1998]_F. Benjamin Zhan. Representing Networks, *NCGIA Core Curriculum in GIScience. 1998.*
http://www.ncgia.ucsb.edu/giscc/units/u064/u064.html, created November 5, 1998.

7.4 Analogue Literature, e-books and e-journals (pdf)

[AHUJA 1993] Ahuja, R. K., T. L. Magnanti, J. B. Orlin: Network flows. Theory, algorithms, and applications. Prentice Hall: 1993.

[ARNBERGER 1975] Arnberger, Erik, Ingrid Kretschmer: Wesen und Aufgaben der Kartographie. Topographische Karten. Wien: Franz Deuticke: 1975. (= Die Kartographie und ihre Randgebiete, Enzyklopädie, Bd.1).

[BANG-JENSEN 2007] Bang-Jensen, Jørgen, Gregory Gutin: Digraphs: Theory, Algorithms and Applications. Corrected Version of the 1st. ed. from 2002. London: Springer, 2007. http://www.cs.rhul.ac.uk/books/dbook/main.pdf

[BARDOSSY 2002] Bárdossy, András, Fridjof Schmidt: FRIDJOF SCHMIDT (2002): GIS approach to scale issues of perimeter-based shape indices for drainage basins. In: Hydrological Sciences Journal. Vol. 47, No. 6, 2002, pp. 931-942.

[BEHR 1989] Behr, O.: Digitales Modell des Oberflächenentwässerungssystems von Österreich. Wien: Technische Universität Wien, Inst. für Hydraulik, Gewässerkunde und Wasserwirtschaft und Inst. für Photogrammetrie und Fernerkundung. Forschungsbericht 11, 1989.

[BENTLEY 1993] Bentley, Jon L., M. Douglas McIlroy: Engineering a Sort Function. In: Software - Practice and Experience, Vol. 23, No.11, 1993, pp. 1249-1265.

[BONDY 1976] Bondy, John Adrian, U.S.R. Murty: Graph Theory with Applications. London, et al.: Macmillan, 1976.

[BREYER 1992] Breyer, S.P., R.S. Snow: Drainage basin perimeters: a fractal significance. In: Geomorphology. Special Issue: Fractals in Geomorphology. Vol. 5, Nos. 1/2, 1992, pp. 143-157.

[CHARTRAND 1977] Chartrand, Gary: Introductory Graph Theory. New York: Dover: 1985. (Corrected republication of Chartrand, Gary: Graphs as Mathematical Models. Boston: Prindle, Weber & Schmidt, 1977.)

[CHORLEY 1959] Chorley, Richard J.: The shape of drumlins. In: Journal of Geology, Vol.3, No. 25, 1959, pp. 339-344.

[CHORLEY 1984] Chorley Richard J., Stanley Alfred Schumm, David E. Sudgen: Geomorphology. London: Menthuen & Co, 1984

[CIS GD22 2009] Common Implementation Strategy for the Water Framework Directive (2000/60/EC): Guidance Document No. 22. Updated Guidance. Implementing the Geographical Information System (GIS)

Elements of the EU Water policy. Luxembourg: Office for Official Publications of the European Communities, 2009

[COMMITTEE ON LARGE DAMS 1985] Österreichische Staubeckenkommission: Hydro power schemes and large dams in Austria / publ. by Österreich. Staubeckenkommission. - Wien, et al.: Springer, 1985. (Die Talsperren Österreichs ; 29).

[CORMEN 2009] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms. 3rd edition. Cambridge, London: MIT Press, 2009.

[CURTIN 2007] Curtin, Kevin M.: Network Analysis in Geographical Information Science: Review, Assessment, and Projections. In: Cartography and Geographic Information Science, Vol. 34, No. 2, 2007, pp. 103-111.

[DASGUPTA 2008] Dasgupta, Sanjoy, Christos H. Papadimitriou, Umesh Vazirani: Algorithms. Boston, Mass. [u.a.]: McGrawHill Higher Education, 2008.

[DEMMER 1991] Demmer, Wolfgang: International Commission on Large Dams: Dams in Austria: prepared in commemoration of the seventeenth congress of the International Commission on Large Dams by the Austrian National Committee on Large Dams / main contrib.: W. Demmer. [Eds.: F. Huber (ed.-in-chief) ...] . - Vienna: Österr. Nationalkomitee für Talsperren ; Vienna : Österr. Staubeckenkommission, Österr. Wasserwirtschaftsverb. , 1991. (Die Talsperren Österreichs; 32).

[DIESTEL 2006] Diestel, Reinhard: Graphentheorie. 3. Auflage. Heidelberg: Springer, 2006.

[EUROPEAN COMISSION 2000] European Commission: Directive 2000/60/EC of the European Parliament and of the Council of 23rd October, 2000: Establishing a framework for Community action in the field of water policy. In: Official Journal of the European Communities, 43, L327, 2000.

[GARDINER 1978] Gardiner, V., C.C. Park: Drainage basin morphometry. In: Progress in Physical Geography. March 1978 No. 2, pp. 1-35.

[GLEYZER 2004] Gleyzer, Alexander, Michael Denisyuk, Alon Rimmer, Yigal Salingar: A fast recursive GIS algorithm for computing Strahler stream order in braided and nonbraided networks. In: Journal of the American Water Resources Association (JAWRA), Vol. 40, No. 4, 2004, pp. 937 - 946.

[GOODCHILD 1987] Goodchild, Michael F., David M. Mark: The Fractal Nature of Geographic Phenomena. In: Annals of the Association of American Geographers, Vol. 77, No. 2, Jun., 1987, pp. 265 - 278.

[GOODCHILD 2001] Goodchild, Michael F.: Metrics of scale in remote sensing and GIS. In: Journal of applied Earth Observation and Geoinformation, Vol. 3, No. 2, 2001, pp. 114 - 120.

[GOODCHILD 2011] Goodchild, Michale F.: Scale in GIS: An overview. In: Geomorphology, 130, 2011, pp. 5 - 9.

[GORBUNOV 1980] Gorbunov, Y.V.: Calculation of channel storage based on Horton-Strahler-Rzhanitsyn laws of the river system structure and flood forecasting. Hydrological forecasting. Proceedings of the Oxford Symposium April 1980. (= IAHS Publ. no. 129).

[GOUDIE 1990] Goudie, Andrew S. et al. (eds): Geomorphological techniques. London: Unwin Hyman, 1990

[GRAF 1975] Graf, William L.: A Cumulative Stream-ordering System. In: Geographical Analysis, Vol. 7, No. 3, 1975, pp. 336 - 340.

[GRAVELIUS 1914] Gravelius, H.: Flußkunde. Göschen: Berlin, Leipzig: 1914, pp. 1 – 179. (= Grundriß der gesamten Gewässerkunde in 4 Bänden, Bd. 1)

[GREGORY 1973] Gregory, K. J., D. E. Walling: Drainage Basin Form and Process. A geomorphological approach. London: Edward Arnold, 1973.

[HENRIKSEN 2003] Henriksen, Jørgen, Lars Troldborg, Per Nyegaard, Torben Obel Sonnenborg, Jens Christian Refsgaard, Bjarne Madsen: Methodology for construction, calibration and validation of a national hydrological model for Denmark. In: Journal of Hydrology, 289, 2003, pp. 52 - 71.

[HOPCROFT 1974] Hopcroft, John, Robert Tarjan: Efficient Planarity Testing. In: Journal of the Association for Computing Machinery (ACM). Vol. 21, No. 4, 1974, pp. 549 - 568.

[HORTON 1945] Horton, Robert Elmer: Erosional development of streams and their drainage basin. In: Bulletin of the American Geological Society, No. 56, 1945, pp. 275-370.

[JARVIS 1977] Jarvis, Richard S.: Drainage network analysis. In: Progress in Physical Geography. Vol. 1, No. 2, 1977, pp. 271 - 295.

[JAVA 2012] Bloch, Josh: Java Tutorials, Trail: Collections. E-book. Oracle, 2012.

[KLEINBERG 2006] Kleinberg, Jon, Éva Tardos: Algorithm Design. Boston, San Francisco, New York, et al.: Pearson, 2006.

[KLINKENBERG 1994] Klinkenberg, Brian: A Review of Methods Used to Determine the Fractal Dimension of Linear Features. In: Mathematical Geology, Vol. 26, No. 1, 1994, pp. 23.

[KNUTH 1976] Knuth, Donald Ervin: Big Omicron and big Omega and big Theta. In: SICACT News, Vol. 8, No. 2, 1976, pp. 18 - 24.

[KNUTH 1997] Knuth, Donald Ervin: The Art of Computer Programming. Vol. 1, Fundamental Algorithms. 3rd Ed. Boston, et al.: Addison-Wesley 1997.

[KÖNIG 1950] König, Dénes: Theorie der endlichen und unendlichen Graphen. Providence, Rhode Island: AMS (American Mathematical Society) Celsea Publishing, 1950.

[KUNZINGER 2002] Kunzinger, Michael, Andreas Ulovec: Algorithmen, Datenstrukturen und Programmieren I. Programmieren in Java. Skriptum zur Vorlesung. 4. Auflage. Wien: 2002.

[LANFEAR 1990] Lanfear, Kenneth J.: A fast algorithm for automatically computing Strahler stream order. In: JAWRA Journal of the American Water Resources Association (also: Water Resources Bulletin), Vol. 26, No. 6, 1990, pp. 977–981.

[LUCE 1952] Luce, Duncan R.: Two Decomposition Theorems for a Class of Finite Oriented Graphs. In: American Journal of Mathematics, Vol. 74, No. 3, 1952, pp.701-722.

[MAIDMENT 2002] Maidment, D.R., (2002), Arc Hydro: GIS for Water Resources, ESRI Press, Redlands CA, 2002.

[MANDELBROT 1967] Mandelbrot, Benoit: How long is the coastline of Britain? In: Science 156(3775), 1967, pp. 636 - 638.

[MARITAN 1996] Maritan, Amos, Andrea Rinaldo, Riccardo Rigon, Achille Giacometti, Ignacio Rodriguez-Iturbe: Scaling laws for river networks. In: Physical Review E, Vol. 53, No. 2, 1996, pp. 1510

[MULLER 1991] Muller, Jean-Claude: Generalization of Spatial Databases. In: Maguire, David J., Michael F. Goodchild, David W. Rhind (eds.): Geographical Informations Systems. Principles and Applications. Vol. 1. Harlow (Essex): Longman Scientific and Technical: 1991, pp. 457 - 475.

[OTTMANN 1998] Ottmann, Thomas (Hrsg.): Prinzipien des Algorithmenentwurfs. Heidelberg, Berlin: Spektrum Akademischer Verlag, 1998.

[OTTMANN 2002] Ottmann, Thomas, Peter Widmayer : Algorithmen und Datenstrukturen. 4. Auflage. Heidelberg, et al.: Spektrum Akademischer Verlag , 2002 .

[PFAFSTETTER 1989] Pfafstetter, Otto: Classification of hydrographic basins: coding methodology. Rio de Janeiro: Unpublished manuskript, DNOS (Departamento Nacional de Obras de Saneamento), 1989.

[PREISS 1999] Preiss, Bruno R.: Data Structures and Algorithms with Object-Oriented Design Patterns in Java. New York, et al.: Wiley, 2000.

[RAPER 1995] Raper, J., D. Livingston: Development of a geomorphological spatial model using object oriented design. In: International Journal of Geographical Information Systems, 9, 1995, pp. 359 - 384.

[ROULEAU 1984] Rouleau, B.: Theory of Cartographic Expression and Design. In: International Cartographic Association: Basic Cartography for students and technicians. Vol. 1. 1984, pp. 81 - 111.

[RZHANITSYN 1960] Rzhanitsyn, Nickolai Alexandrovich: Morphological and Hydrological Regularities of the Structure of the River Net. Translated by D. Krimgold (original in Russian). Washington: US Government Printing Office, 1960.

[SAAKE 2004] Saake, Gunter, Sattler, Kai-Uwe: Algorithmen und Datenstrukturen. Eine Einführung mit Java. 2. Auflage. Heidelberg: dpunkt.verlag, 2004.

[SCHEIDEGGER 1965] Scheidegger, A.E.: The algebra of stream-order numbers. US Geological Survey, Professional Paper 525B, 1965.

[SCHEIDEGGER 1966] Scheidegger, A.E.: Effect of Map Scale on Stream Orders. International Association of Scientific Hydrology. Bulletin, 11:3, 1966, pp. 56-61.

[SCHEIDEGGER 1970] Scheidegger, A.E.: On the theory of evolution of river nets. International Asscociation of Scientific Hydrology. Bulletin, 15:1, 1970, pp. 109-114.

[SCHMIDT 1984] Schmidt, Karl Heinz: Der Fluß und sein Einzugsgebiet: hydrogeographische Forschungspraxis. Wiesbaden: Franz Steiner Verlag, 1984. (=Wissenschaftliche Paperbacks: Geographie)

[SEDGEWICK 2011] Sedgewick, Robert, Kevin Wayne: Algorithms. 4th edition - Kindle edition. Upper Saddle River, NJ, Boston, a.o.: Addison-Wesley, 2011.

[SHREVE 1966] Shreve, R.L.: Statistical law of stream numbers. Journal of Geology 74, 1966, pp. 17-37.

[STRAHLER 1957] Strahler, Arthur N.: Quantitative analysis of watershed geomorphology. In: American Geophysical Union Transactions, Vol. 38, 1957, pp. 913-920.

[STRAHLER 2006] Strahler, Alan N., Arthur N. Strahler: Introducing Physical Geography. Fourth Edition. John Wiley&Sons, 2006.

[TARBOTON 1988] Tarboton, David G., Rafael L. Bras, Ignacio Rodriguez-Iturbe: The Fractal Nature of River Networks. In: Water Resources Research, Vol. 24, No. 8, 1988, pp. 1317.

[TARJAN 1972] Tarjan, Robert: Depth-First search and linear graph algorithms. In: SIAM Journal of Computing. Vol. 1, No. 2, 1972, pp. 146-160.

[TEWARSON 1973] Tewarson, Reginald P. (Ed.): Sparce Matrices. Elsevier, 1973. (= Mathematics in Science and Engineering, Vol. 99).

[TIMPF 2000] Timpf, Sabine, Frank, Andre U.: Using Hierarchical Spatial Data Structures for Hierarchical Spatial Reasoning. In: Hirtle, S.C., Frank, A.U. (Hrsg.): Spatial Information Theory – A Theoretical Basis for GIS (International Conference COSIT 97). 2000. (= Lecture Notes in Computer Science, Berlin, Vol. 1329), pp. 69 – 83.

[TURAU 1996] Turau, Volker: Algorithmische Graphentherorie. Bonn, Paris, Reading (Mass.) et al.: Addison-Wesely, 1996.

[VERDIN 1999] Verdin, Kris L., Verdin, James P: A topological system for delineation and codification of the Earth's river basins. Elsevier: Journal of Hydrology 218(1999), pp. 1-12.

[VOGEL 2011] Vogel, Richard M.: Hydromorphology. In: Journal of Water Resources Planning and Managment. Vol. 137, No. 2, pp. 147-149.

[VOSSEN 2000] Vossen, Gottfried: Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme. 4. Auflage. München, Wien: Oldenbourg, 2000.

[WARD 1986] Ward, J. V.: Altitudinal zonation in a Rocky Mountain stream.- Archiv für Hydrobiologie, Suppl. 74. Stuttgart: Schweizerbart, 1986, pp. 133-199.

[WHEATCRAFT 1986] Wheatcraft, S.W., S.W. Tyler. An expalanation of scale-dependent dispersity in heterogeneous aquifers using concepts of fractal geometry. In: Water Resources Research, Vol. 24, No. 4, 1988, pp. 566-578.

[WIMMER 1994] Wimmer, Reinhard, Moog, Otto: Flussordnungszahlen österreichischer Fließgewässer. Wien: Umweltbundesamt, Dezember 1994. (=Monographien Bd. 51).

[WFD 2000] Directive 2000/60/EC of the European Parliament and of the Council of 23 October 2000 establishing a framework for Community action in the field of water policy. OJL 327, 22 December 2000, pp. 1–73

[ZEPP 2002] Zepp, Harald: Grundriß Allgemeine Geographie: Geomorphologie. Paderborn, a.o.: Ferdinand Schöningh, 2002. (=UTB für Wissenschaft; Bd. 2164).

8 Appendices

8.1 Stream order systems

8.1.1 Classic system of stream orders

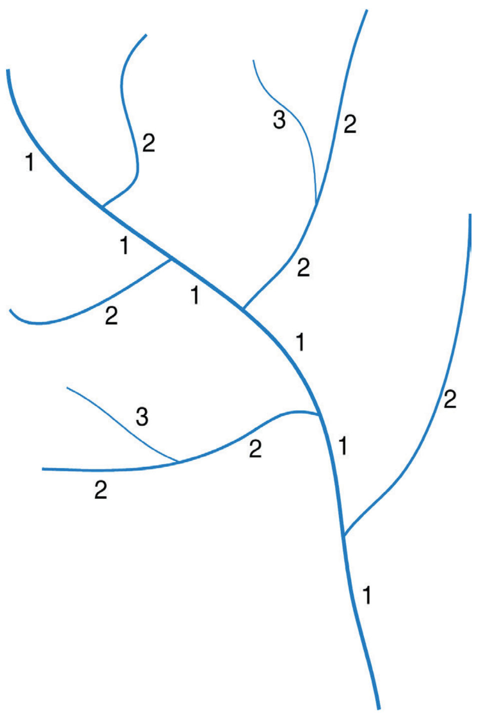


Figure 8–1 Classic system of stream orders

The classic system of stream orders, which is based on Gravelius [GRAVELIUS 1914], assigns the first order to the main river - that is the river emptying into the sea. All those rivers flowing into the first-order river are assigned second order those flowing into the second-order rivers become third order, etc.

"*Indem wir so dem wasserreichsten der beiden obersten Nebenflüsse die Eigenschaft des obersten Stücks des Hauptflusses oder eigentlichen Quellflusses beilegen, schließen wir uns den gegenwärtigen physikalischen Verhältnissen des Flussnetzes so genau als möglich an.*"

([GRAVELIUS 1914], p. 6)[23]

The main river is thus chosen according to the flow conditions.

A version of this system is used in Austria by the Austrian Lebensministerium department of "hydrologic accounting". The rivers Danube and Rhine are thereby first-order rivers.

The system's advantage lies in its clarity; its disadvantage is its neglect of the catchment size. A small adventitious river flowing into the first-order stream has order 2 as well as a big one.

[23]"In assigning the characteristics of the upper segment of the main river to the contributing headwater river with the strongest flow we follow the recent physical conditions of the river network as close as possible."

8.1.2 Stream orders by Horton

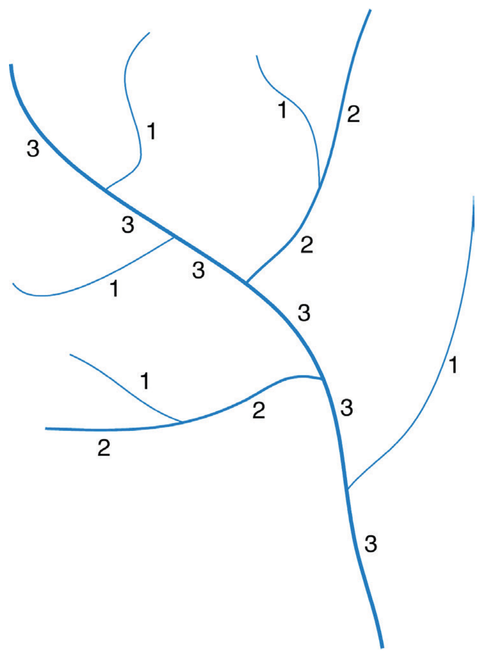


Figure 8–2 Stream orders by Horton

Robert Elmer Horton, the "father" of some important terms in hydrology like "infiltration capacity" and "maximum possible rainfall", developed a theory for stream numbers that also incorporates quantitative aspects. He detailed his theory in a paper published in 1945, only a month before his death [HORTON 1945]. The stream order system by Horton follows the hierarchical layout of river basins, but in contrary to the classic system the numbering starts at the river source. The headwaters (or finger-tip channels as Arthur N. Strahler later named them) are assigned order one. Second-order rivers only have first-order rivers as tributaries, third-order rivers only first and second-order rivers, etc. The strongest tributary in a (sub)system gets its stream order assigned from river mouth to source.

8.1.3 Stream orders by Strahler

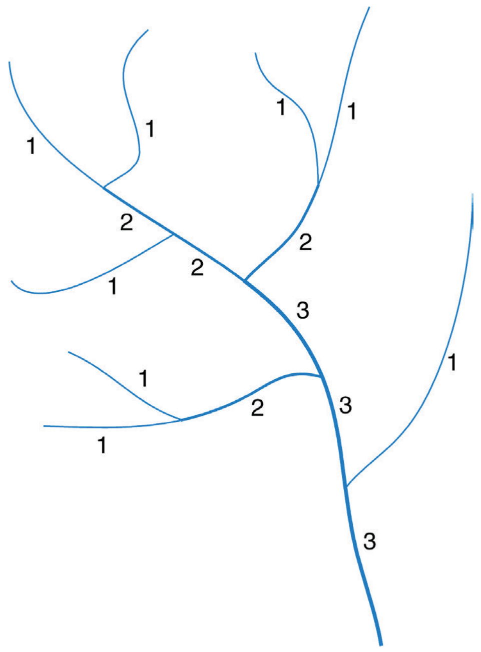


Figure 8–3 Stream orders by Strahler

The concept of stream orders by Strahler [STRAHLER 1957] is based on Horton's method (c.f. 8.1.2.). Actually it's the first step of the Horton method, but only through Strahler it was further distributed. In Strahler's method all headwaters get the order one. From a junction where two rivers of the same order join the order of the downstream river (segment) will be raised by one. Otherwise the higher of the two river orders that meet at a junction will be the new order.

$$O_k = \begin{cases} O_i + 1 & \forall i,j \quad O_i = O_j \\ \max(O_i, O_j) & \forall i,j \quad O_i \neq O_j \end{cases}$$

O_k ... Order of river k

$O_{i,j}$ ...Order of rivers being confluences to river k

8.1.4 Stream orders by Rzhanitsyn

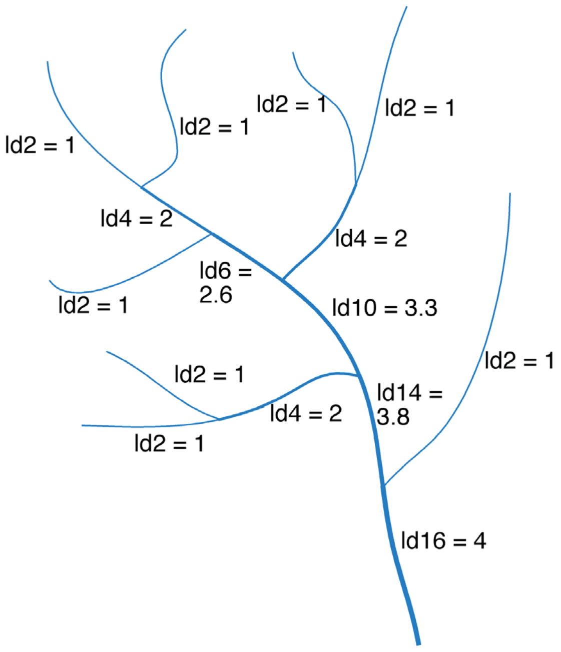


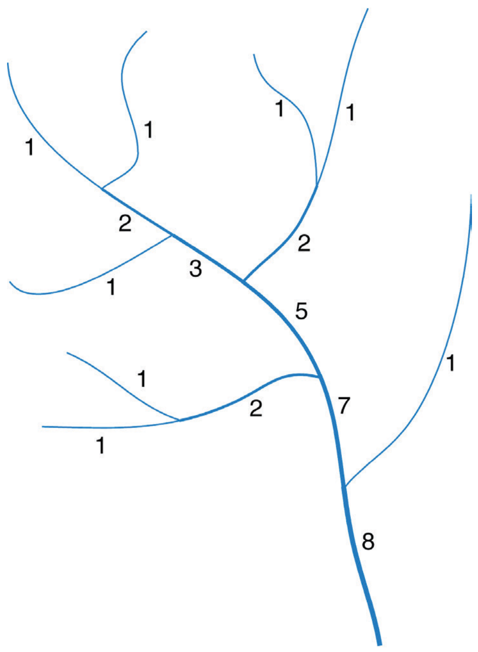
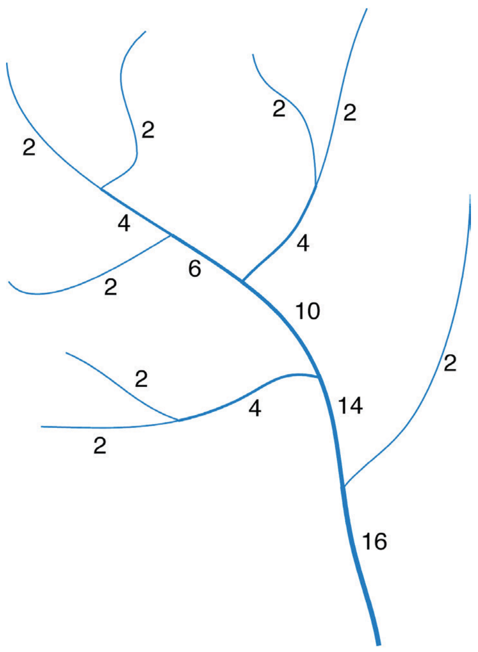
Figure 8–4 Stream orders by Rzhanitsyn

Only shortly after Horton's and Strahler's publications Nickolai Alexandrovich developed an analogical concept. Rzhanitsyn's [RZHANITSYN 1960] approach also considers - compared to Strahler - tributaries of lower orders. Which means, that also rivers of a lower order that flow into a river of a higher order generate a rise in the order numbers. Rzhanitsyn's concept of stream order is logarithmic (based on the logarithm of 2).

$$O_k = O_i + O_j \quad \forall i,j,x \in \mathrm{N} \rightarrow O_i, O_j \in x * \lg 2$$

8.1.5 Stream order by Scheidegger and by Shreve

The approaches by Scheidegger [SCHEIDEGGER 1965] and Shreve [SHREVE 1966] resemble each other. They are additive concepts. William L. Graf proposed a similar stream ordering system [GRAF 1975]. Like Rzhanitsyn the two methods consider tributaries with lower orders. At the rivers confluences the orders are simply summed up. The main difference between the two methods is the initial value - for the headwaters Shreve uses the order 1, Scheidegger uses order 2. Shreve's orders are therefore always identical to the count of headwaters, Scheidegger's orders are the count of headwaters multiplied by 2. The relation of Shreve and Scheidegger orders is always 2:1. The term magnitude is often used in context with Shreve ordering (esp. when using STRAHLER ordering in the same text. e.g. [JARVIS 1977]).

 1 1 1 1 2 3 2 1 5 1 1 2 7 1 8 **Figure 8–5 Stream orders by Shreve**	 2 2 2 2 4 6 4 2 10 2 2 4 14 2 16 **Figure 8–6 Stream orders by Scheidegger**

$O_k = O_i + O_j \quad \forall O_i, O_j \quad \rightarrow \quad O_i, O_j \geq 1$ $\qquad$ $O_k = O_i + O_j \quad \forall O_i, O_j \quad \rightarrow \quad O_i, O_j \geq 2$

Rzhanitsyn orders are to be generated by logarithmizing Scheidegger orders on the logarithmic basis of 2.

8.1.6 Stream order by Ward

Ward [WARD 1986], a limnologist, combined Strahler's widely used concept with those of Shreve.

> *In the site description that follows, a modification of the stream order system (STRAHLER 1957) has been adopted to account for tributaries of a lower stream order (which do not increase the stream order designation).*

For every tributary of lesser order, a superscript is added. If, for example, a first-order tributary enters the third-order stream, the designation is "31"; if a second-order stream (or two first-order streams) then enters, the designation becomes "33". This system essentially combines the widely used concept of stream order with the advantages conferred by link magnitude, which is the sum of all first-order streams (Shreve 1966). First order streams are those shown as permanent streams without tributaries on 7.5 minute topographic maps (scale 1:24000). [WARD 1986, p.137]

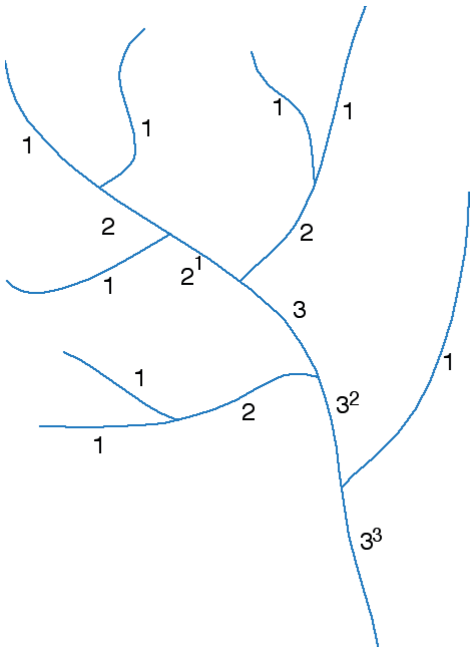


Figure 8–7 Stream orders by Ward

8.1.7 Stream order system by Pfafstetter

The Pfafstetter system [PFAFSTETTER 1989] is a system where, similar to the classic system of stream orders (see above 8.1.1), the first-order stream empties into the sea.

At the heart of a basin's identity are the size and shape of the catchment area and channel configuration that produce flow at the outlet. All channel reaches have unique direction, and therefore order, and they are arranged in a bifurcated network. ([VERDIN 1999], p. 3)

This system is based upon the topology of the drainage network and the size of the surface area drained. Its numbering scheme is self-replicating, making it possible to provide identification numbers to the level of the smallest sub basins. For a given location it is possible to automatically identify all upstream sub basins, all upstream river reaches, or all downstream reaches. Additionally to the classic system the size of the *tributary (or better the size of its catchment) will be involved in the calculation of the order.* ([BRITTON 2002], p. 2)

Pfafstetter coding steps

1. Determine main river
2. Determine the 4 tributaries with the largest catchment areas (basins). Those get the even numbers 2 to 8
3. The remaining tributaries become interbasins and are coded with the uneven numbers 1 to 9, whereas the upper end of the main river (from the number 8 tributary to the source) is always coded with 9.
4. Repeat steps 1 to 3 for all the basins and interbasins and add the number as a new digit to the right of the existing code.

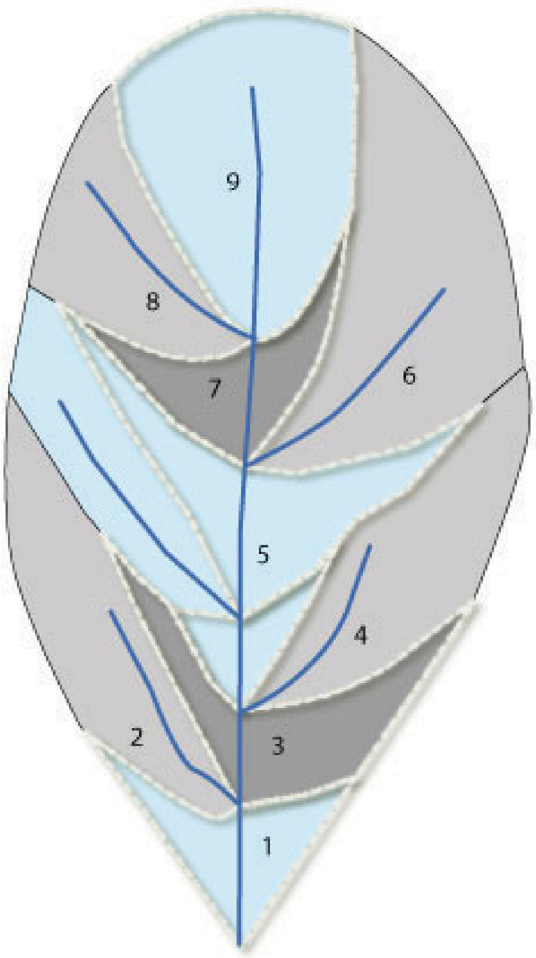


Figure 8–8 Pfafstetter Coding

The white surrounded areas are the catchments of the main stem of this Pfafstetter level. Notice: Interbasin 5 is composed of main stem catchments and a catchment that was not in the list of the four largest sub-catchments.

The lowest catchment of a Pfafstetter level is always catchment 1, the highest catchment is always coded catchment 9. It may happen that there are not enough catchments between the lowest and the highest, thus that there can be free numbers.

The advantage of the Pfafstetter system lies in it optimized use of numbers and its easy interpretability concerning the location of the rivers in the river system. The disadvantage lies in the big number of polygons that have to be calculated to get the catchment size.

> *A modified version of the Pfafstetter system is proposed as the European coding system for hydrological features. The hydrological code is composed of different segments, which together uniquely identify a hydrological feature. The hydrological code consists of 6 hierarchical related items. The first item is a character defining the Ocean or Endorheic system. It is followed by one digit numbering of the seas into which the Ocean can be subdivided. In the case of islands subsequently a sequence number of the island order along the coast is defined. The landmasses thus defined can be subdivided at sea outlet level using the 5 digit length commencement code. Finally the river system can be coded up to the river reach level using the Pfafstetter methodology. ([CIS GD22 2009], p. 99)*

Thus after a 5 digit preceding code that defines the outlet of a stream system Pfafstetter coding will be used for all rivers that are reported to the EC. To provide the necessary means to create a Pfafstetter coding catchments must be available for all the rivers. These catchments form the basis on which can be decided what river has the bigger aggregation area.

8.2 Supportive Java Add-ins

The following section lists the ArcGIS Java Add-ins that help preparing the network for the use of the algorithms from chapter 4.2 .

8.2.1 CheckHydroID.java

Description:

Calculates for a dataset a field called HydroID. HydroID is a positive and unique number > 0. If for example segments were split and have the same HydroID or new features have no HydroID this routine calculates it. The original HydroID is written to a field HydroIDold. If the original HydroID is null then HydroIDold will be -9999. New numbers are larger than all existing ones. CheckHydroID uses methods from the ArcObjects class IDataStatistics, namely getUniqueValue, getUniqueValueCount, and getMaxiumum. To find the values that are not unique the algorithm uses a HashSet (which is described as the best performing Set implementation in Java) and writes their OBJECTID into another HashSet, which is used to replace the not unique numbers with new ones.

Data preconditions:

- Data must allow edits.
- FeatureClass (river segments, junctions, ...)
- HydroIDs must be positive integers

Running time:

From the ArcObjects API the Time complexity of IDataStatistics methods getUniqueValue, getUniqueValueCount, getMaximum is not clear, but most probably it is not less efficient than n*log n. This is based on the assumption that the algorithms are implemented using modern sorting algorithms (which are O(n * log n)).

Time Complexity	Steps
O(n*log n)	Find out if there are more features than unique values.
O(n*log n)	Build a HashSet and use it to get positive multiple values.
O(n)+(n*log n)	If there are multiple values calculate the new ones.
O(n)+(n*log n)	Once go through the data set and set all negative values to a new positive unique number.
O(n*log n)	

8.2.2 SetFlowDir.java

Description:

Sets the flow direction in a geometric network according to the field FLOWDIR.

Flowdir = 1 – same as digitized order

Flowdir = 2 - against digitized order

Flowdir = 3 - not determined

All other Flowdir's - uninitialized.

There are no fields calculated, but the flow direction in the geometric network is established. If an arc flows in the wrong direction, either the value in the field FLOWDIR has to be adapted or the arc has to be flipped before using SetFlowDir.java again.

Data preconditions:

- Data must allow edits
- FeatureClass of river segments with FLOWDIR field.
- River segments have an existing network topology

Running time:

O(n). To avoid unnecessary time in very big datasets for already set flow directions, the program can also be used on a selection of edges.

Time Complexity	Steps
O(n)	Once run through the dataset, query and apply the FLOWDIR field in a constant number of steps for every edge.

8.2.3 FromToNodes.java

Description:

Writes for every river segment the from- and to-junction-number into the FeatureClass' attribute table. If the junction does not have a valid id, a default value (-9999) will be calculated.

Data preconditions:

- Data must allow edits.
- FeatureClass of river segments with a unique id number (in the Add-in this is called HydroID)

- FeatureClass of river junctions with a unique id number (in the Add-in this is called HydroID)
- River segments have an existing network topology (geometric network)
- Field FlowDir that indicates whether the river segment is digitized in or against flow direction.

Running time:

The Add-in once runs through the river dataset and uses the object oriented data structure to retrieve values of the from- and to-junction of every river segment.

Time Complexity	Steps
O(n)	Once run through the dataset and query the junctions in a constant number of steps

8.2.4 JunctionValences.java

Description:

Gets for every junction the overall number of rivers attached to the junction (Valence), and calculates the number of outflowing rivers (ValenceOut) and the number of incoming rivers (ValencIn).

Data preconditions:

- Data must allow edits.
- Network topology (geometric network)
- Field FlowDir that indicates whether the river segment is digitized in or against flow direction.

Running time:

The Add-in once runs through the junction dataset and uses the object oriented data structure to retrieve and calculate the valence values. The constant factor per run is large, because the endpoints of the adjacent edges have to be queried and checked against the FlowDir value and the junction's id.

Time Complexity	Steps
O(n)	Once run through the dataset and query the junctions in a constant number of steps

8.2.5 CreateWatershedNetworkGP.java

Description:

CreateWatershedNetworkGP.java is a program that converts the watershed polygons into centroids using ArcGIS geoprocessing tools (FeatureToPoint, Copy), creates straight lines between two centroids if they should be connected (see NextDownIDs). From centroids and lines a geometric network is created using the INetworkLoader2 interface. The necessary relationships between the data sets in the database are established, to help the user address the watersheds.

Data preconditions:

- Watershed polygons with NextDownID, NextDowID2 (can be empty), NextDowID3 (can be empty) and a corresponding HydroID field.

Running time:

Running time is not calculable for this Add-In, because there is no information available about the running time of the algorithms of the used ArcTools (but the used ones are most probably of O(n)) and the INetworkLoader2 interface. Compared with other measured times of algorithms with known time complexity it could still be possible that the running time is of O(n) but with a large number of constant steps.

8.2.6 Multipart.java

Description:

Finds multipart features in a FeatureClass and selects them. Can be useful as well for watershed cleaning routines or to control the geometry of rivers and routes.

Data preconditions:

- FeatureClass

Running time:

O(n), because once every feature is visited.

Time Complexity	Steps
O(n)	Once run through the feature table of the selected dataset and query the GeometryCount (this is a field in the ArcObjects' IGeometryCollection interface that shows for every feature out of how many pieces it is constructed

9 CD/ROM

9.1 License Information and Disclaimer

It is therefore strongly recommended to backup the databases before starting to use the software.

9.2 Contents of the CD/ROM

The CD/ROM contains an ArcGIS Add-In for ArcMap (DRWaterNetworkAddin). In the Folder Source the following files are stored for viewing purposes:

StreamOrdering Classes	
Lanfear.java	Calculates Strahler numbers with Lanfear algorithm.
Gleyzer.java	Calculates Strahler numbers with Gleyzer algorithm (for braided rivers), also calculates river segments.
Shreve1.java	Calculates Shreve numbers (for braided rivers), also calculates Rzhanitsyn numbers and Scheidegger numbers.
Accumulation Classes	
Shreve1.java	see above - Shreve numbers are the "magnitude" (count) of headwater segments.
AccumulateRivers.java	Calculates the accumulative value of an arbitrary numerical field for rivers and junctions.
AccumulateCatchments.java	Calculates the accumulative value of an arbitrary numerical field for ordered catchments.
AggregateCatchments.java, AggregateCatchments1.java, AggregateCatchments2.java	Creates aggregated areas flowing into selected catchments (there are 3 versions that use different approaches to aggregate the data: array version, in memory recursion, feature class version).
Supportive Add-in Classes	
SetFlowDir.java	Sets the flow direction based on an

	integer field "flowdir".
FromToNode.java	Adds a field with the from-nodes and to-nodes to a geometric network edge feature class.
JunctionValences.java	Counts the overall cardinality of each node as well as its from and to-valence.
CreateWatershedNetworkGP.java	Creates a network from ordered catchments based on nextdownID(s).
FindBackflows.java	Returns a list of reachable backflowing arcs.
FindCircles.java	Returns a list of circles.
Helper Classes	
EditorStarter.java	Helps to start editing in the right workspace and prevents losing data.
FCFieldChooser.java	A dialog to choose a field from a feature class.

9.3 Information for the use of the CD/ROM

The DRWaterNetworkAddin is written for ArcGIS 10.1. Further research is planed which may lead to add-ins for newer versions of ArcGIS Software. The information will be posted on http://homepage.univie.ac.at/doris.riedl/.

ESRI provides on its ArcGIS Resource Centre a description how to deploy add-ins.
http://resources.arcgis.com/en/help/arcobjects-java/concepts/engine/index.html -/How_to_deploy_your_add_in/0001000006sp000000/
In short you may run through the following steps:

- Copy DRWaterNetworkAddin from the CD/ROM to a location on your computer.
- Open ESRI ArcMap.
- Open Customize/Add-In Manager ...
- Choose Options tab
- With Add Folder ... add the location of your copy of DRWaterNetworkAddin

- Choose the option: "Load all Add-Ins without restriction"
- You may have to restart ArcMap to see DRWaterNetworkAddin in the Add-In Manager Add-Ins tab.
- Open Customize/Customize Mode ...
- Choose Toolbars tab
- Check GN Toolbar

Load your data.
Select the dataset you want to apply a command to and run the command (the commands are assigned to different topics and can be found in the drop-down menus). If you should choose by mistake a dataset that is not appropriate for the command, the program will check the input, before running the calculation part of the command.

Declaration

I hereby confirm on my honor that I personally prepared the present academic work and carried out myself the activities directly involved with it. I also confirm that I have used no resources other than those declared. All formulations and concepts adopted literally or in their essential content from printed, unprinted or Internet sources have been cited according to the rules for academic work and identified by means of footnotes or other precise indications of source.

Mag. Doris Riedl

May 2014, Vienna

Curriculum Vitae

Title and Name	Mag. Doris Riedl
Date of Birth	11.02.1971
Place of Birth	Wels, Austria

Education

1991-1997	**University of Vienna** Magister in Geography (with excellent success) and German Philology
1985-1990	**HGBLA für Mode und Bekleidungstechnik in Linz/Austria** Matura with excellent success
1977-1985	**Volksschule (primary school) and Hauptschule (secondary school) Gunskirchen/Austria**

Work Experience

1998-2008 and 2014	Lecturer for Geomorphology and Geoinformation at University of Vienna
1998-2008	Federal Environmental Agency Austria water department - Desktop officer for geoinformation and cartography
2008 - 2012	University Assistant - University of Vienna
2013 to present	Teacher at Bundesgymnasium Maroltingergasser/Vienna