INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

A Research Framework for Graph Theory in Routing Applications

Edgar-Philipp Stoffel

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach

Betreuer: Dipl.-Inf. Bernhard Lorenz

Abgabetermin: 15. November 2005

Mittelpunkt dieser Diplomarbeit ist die Konzeption und Implementierung eines Systems, welches Wegeplanung in realen Verkehrsnetzen durch eine ausreichend tiefgehende Modellierung des Anwendundsgebietes unterstützt.

Das Hauptaugenmerk des gewählten Ansatzes gilt in erster Line gerichteten, gewichteten Graphen sowie Techniken der Wissensrepräsentation. Gemeinsam mit Ontologien eignet sich dieser mathematische Formalismus nämlich bestens, um Verkehrsnetze auf eine eindeutige Weise zu beschreiben.

Insbesondere findet die Modellierung von hierarchischen Strukturen wie z.B. Cluster in diesen Netzen besondere Geltung. Darüber hinaus werden geeignete Algorithmen für die Wegeplanung in Betracht gezogen, von denen die meisten von Dijkstras Algorithmus zur Berechnung des kürzesten Weges in Graphen ableiten und durch zusätzliche Attribute als Gewichtsfunktion flexibel gestaltet sind.

Da heutige Informationssysteme überwiegend auf einige Verkehrsmittel spezialisiert sind, ist eine einheitliche semantische Darstellung für den übergangslosen Wechsel von verschiedenen Verkehrsmitteln, auch über Grenzen von Städten oder Gebäuden nach außen hinweg erstrebenswert. Komplexe Strukturen wie Länder, Regionen und Städte enthalten alle wertvolle Informationen für Heuristiken zur Wegeplanung.

Als Ergebnis dieser Arbeit ist ein Rahmenwerk entstanden, welches durch den Ausbau von weiteren Komponenten zur Verwendung im Kontext von räumlichem Schließen (*Geo-Spatial Reasoning*) bestimmt ist.

Dedicated to the conceptual design of a framework which fulfills the functions of routing within real life transport networks, this thesis is endeavouring to tender a sufficiently accurate modelling of this specific domain, including many of its eminent properties.

The approach pursued by this work is primarily characterised by being focussed on the use of directed, weighted and attributed graphs along with knowledge-based methods. Indeed, this abstract mathematical concept supplemented by the use of ontologies is appropriate to describe transport networks in an unambiguous manner. Special emphasis, beyond, is being put upon cluster-like hierarchic structures which essentially compose these networks. Furthermore, suited routing algorithms are taken into consideration, most of which derive from Dijkstra's algorithm for finding shortest paths in graphs, accessorily extended by the notion of using an adjustable cost function as argument.

However, unlike in the predominant majority of current systems for this purpose, the one proposed here is not confined to some selective modes of transportation (automotive), but rather is capable of representing diverse networks uniformly so that seamless transitions can take place - not only from one transport system to another, yet also from within the boundaries of complex buildings and cities to the outside. Since entities like countries, regions, cities, buildings et cetera are distinguished by a nested inner structure, especially with regard to heuristics, it is their inherent hierarchic nature which constitutes valuable information for routing algorithms.

By the contribution of this work, namely a solid foundation for further enhancement, the appliance of reasoning techniques pertaining to geo-spatial entities and the task of routing shall be facilitated.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. November 2005

Unterschrift des Kandidaten

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Always has there been a fascination to travelling for humans.

In our modern society, for the purpose of overcoming greater distances, a sheer variety of different transportation networks exist, comprehending road networks inside and between major metropoles, complex public transit systems (underground, buses, tram), maritime networks (ferries) and networks of flight connections (airplanes), and many more. A traveller therefore has a great deal of choice concerning the mode of transportation he can select for getting to his desired location along this multitude of networks, each of which by itself is a highly complex structure.

'How do I get from Munich to Paris the cheapest way ?'

'At which gate in Charles de Gaulle is the flight connection ?'

It is these and similar everyday questions asked by travellers, who are willing to reach a destination, before making a journey, what routing is all about.

Due to information systems (map servers and navigation devices), the task of acquiring information about transport networks in order to make a choice which transport mode to employ can be performed automatically. Nowadays, whenever a person wants to get from one location to another, he can rely on a the guidance offered by electronic systems: In-vehicle navigation systems as well as travel information systems on the World Wide Web, which are offering routing services, have experienced a substantial boom in recent years. Automated, intelligent routing by systems is the desirable goal to be striven towards.

However, little effort has been made to overcome the lack of integration between these different systems. Semantics are a key concept for the solution of this problem.

1.2 Goals and Scope

An object-oriented framework for routing applications - *TransRoute* - which is capable of not only of representing various real-life transport networks (highways, public transport etc.), but also of answering shortest path and nearest neighbour queries applicable to those, has been developed in *Java 5.0* [22] as a contribution to the *REWERSE* project [16] for geo-spatial reasoning.

In order to make the World Wide Web, as it is known today, machine processable, *REWERSE* is aiming at enhancing it with semantic concepts. Above all, this principle is of utmost importance and hence to be respected.

Basically, *TransRoute* distinguishes three different levels of abstraction by separating between **ontology concepts** describing real-world entities of the domain (**transport networks**) together with their attributes and relations, and a **graph structure** which can be instantiated with the respective concepts from the ontology.

The core functionalities of the framework consequently consist of following aspects:

Ontology Since this framework will be employed in the context of *REWERSE*, ontologies, which have proven to be well-suited to describe complex entities and features of the real world in a machine-processable way, are naturally a substantial part of it.

Within the package *de.lmu.ifi.pms.transroute.domainmodel*, which uses the *OWL API* framework for Java [13], there are tools for loading an *OWL* ontology [27] and checking the domain types of graph elements against it to be valid. Furthermore, *de.lmu.ifi.pms.transroute.io* offers the *OWLDataReader* class which implements the *IReader* interface for loading graphs from physical files. Consequently, the framework is well-suited for use within web-context.

Graph Structure An object-oriented in-memory representation of a graph structure, mainly from the *JUNG* framework [21] pertaining to transport networks, can be found in the package *de.lmu.ifi.pms.transroute.graph.generic*. Graphs, which are assumed to be sparse, can contain vertices and both directed and undirected edges. The generic graph structure can be seen as a hull for the domain concepts defined in the ontology, which can be filled into the according graph element. Being of abstract nature, the graph structure is highly reusable for different concepts.

For example, a vertex on a high abstraction level represents an entire transport network, and consequently, its incident edges stand for connections to other transport networks. In contrast, another vertex may represent a concrete bus station, whose adjacent edges are physical connections. Yet another application to a generic vertex is the modelling of a country, its incoming and outgoing edges representing adjacency, containment and part-of relations between other countries, continents and cities.

Not only are all graph elements attributed, i.e. attributes can be attached to them arbitrarily, but also they comprehend a built-in hierarchic structure which can be navigated and entails interesting semantic concepts ensuring its integrity (see chapter 4). Especially for edges, numeric attributes can be considered to be edge weights. Some predefined attributes exist concerning location, geometry and the ontology class represented. For each graph element, an appropriate factory class exists abstracting over the purpose of creating the former. Programming against this API, of course, one can dynamically enrich the graph structure of a transport network.

Persistence In contrast to the transient in-memory structure from above, the persistent representation of graphs is responsible for saving those onto physical files, whose format notably is a specialised XML derivate for specifying graphs known as *Graph eXchange Language* [8]. By using the *GXL API* for Java [19], *de.lmu.ifi.pms.transroute.io* offers tools for both reading from and writing to *GXL* files, which implement the *IReader* and *IWriter* interfaces respectively. Founded on a well-defined document type definition, *GXL* semantics are clear for other applications handling this file format. It is possible to store attributes of common basic types (*integer*, *float*, *boolean*, *String*), as well as composite attributes representing vectors for any graph element. One of the primary benefits of *GXL* is that by using it, it is also possible to represent nested graphs within edges and vertices, which obviously makes it suitable for saving the hierarchic graph structure.

As any file reader or writer fulfilling the contract imposed by the interfaces *IReader* and *IWriter* can handle data exchange, the framework can be extended by further plug-ins for different data formats (e.g. *GML* [11] or *GDF* [57]).

So the attention of the framework obviously is upon graph representation of transport networks (e.g. bus, rail, streets, air connections, ferry lines..). Embedded by the ontology modelling are entities of the infrastructure along with their attributes, which may either be fixed or dynamic, the latter

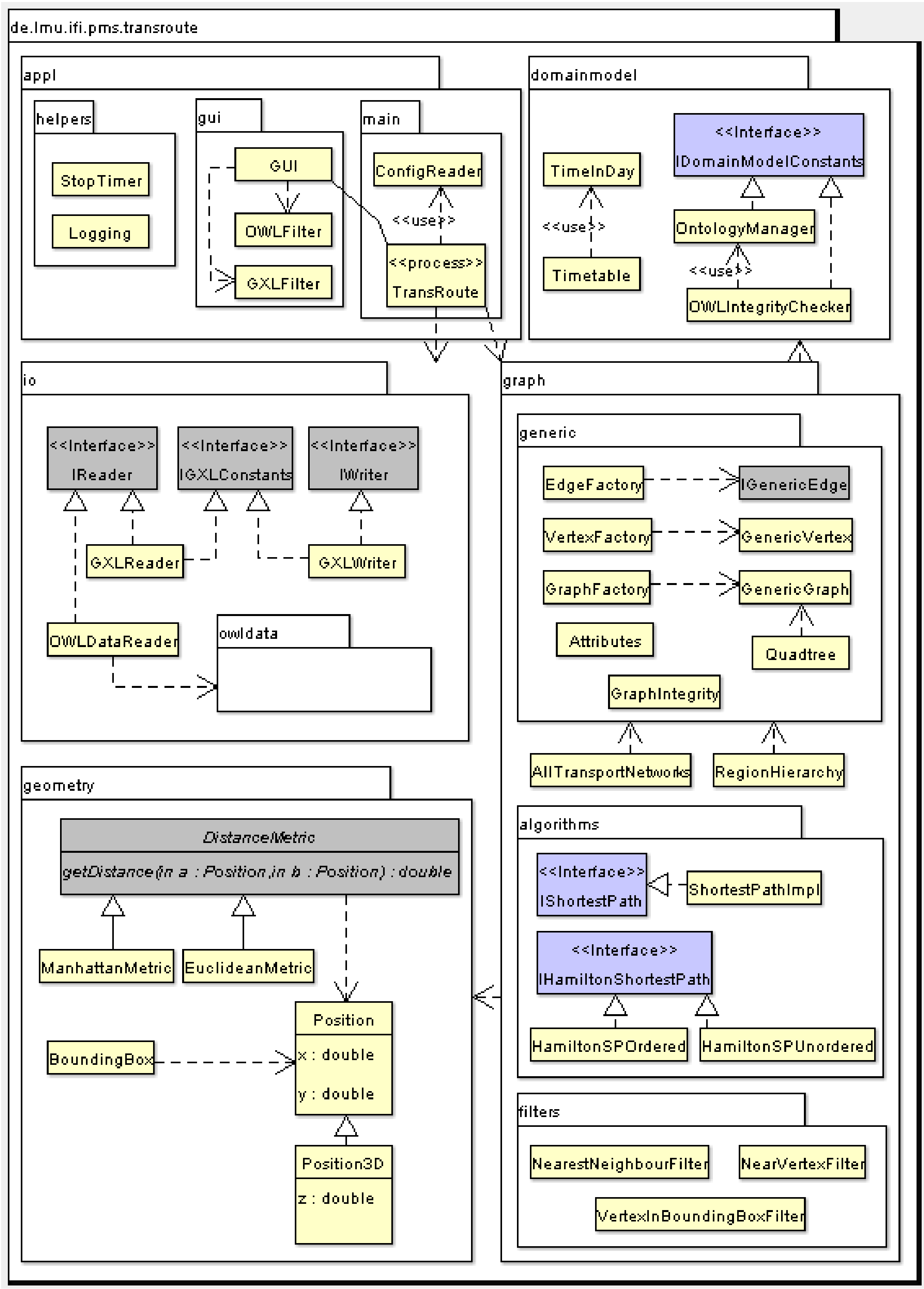


Figure 1.1: Overview of Framework Packages

being vulnerable to umpteen updates, due to incoming traffic messages or periodical measurements concerning the flow of traffic. The corresponding graph elements take over all these properties.

Having decided to use directed and weighted graphs as mathematical formalism, routing is equivalent to searching shortest paths fulfilling special constraints in graphs by standard algorithms or some of their derivatives. Employing a shortest path algorithm, one can profit from an important principle of dynamic programming, stating that such an algorithm being too complex can be subdivided into smaller problems (proof in [52]): A sub-path within a shortest path is also optimal (sub-optimality), hence a shortest path can be decomposed into smaller ones, which are built more easily.

Mostly, these efficient derivates of standard algorithms in form of heuristics comprising techniques of reasoning are of practical value for enormous computer models since they yield faster results at the trade-off of less accuracy. Some of them showing proximity to human cognition, they can be considered to be a natural way of finding a solution [48].

A further improvement manifests itself in the hierarchic graph structure [45] making the framework applicable beyond real life transport networks: Buildings as well as cities including their interiors among other hierarchic entities can be represented by the generic graph structure. Integrity is maintained by strict restrictions for adding vertices and edges into the right place of the hierarchy (see chapter 4).

Furthermore, multi-modal transportation including transfers can be regarded more formally [36] in terms of combinations of transport modes forming certain patterns. Transport networks can be seen as generic vertices in an abstract graph, for which routing essentially decides the combination of transfers. All these aspects influence reasoning techniques for a more complex shortest path algorithm in transport networks. Altogether, the results achieved by this thesis found a solid basis for further research in this field.

1.3 Synopsis

The structure of this diploma thesis consists of two major parts: Dedicated to the illustration of the domain and methods of graph theory, the first one can be considered as general part, whereas the second part is concerned with special applications and extensions of graph theory for traffic in transportation networks after having explored the domain and graph theory. Finally, the framework *TransRoute* with respect to its design principles and major components is explained more deeply.

Chapter 2 introduces the domain of application, namely all different sorts of real world transport networks prevailing, in which routing will take place, and focuses on the systems available for this purpose, illustrating further ideas how to enhance them.

Chapter 3 points out the powerful repertoire of graph theory with respect to problem solving, and shows the benefits of using graphs as a mathematical formalism for routing purposes, which evidently become apparent in terms of various efficient algorithms.

Chapter 4 goes beyond the standard concepts of graph theory into special ones apt for transport networks by applying geometrical and hierarchic extensions, showing how they may influence the behaviour of a routing algorithm taking into account involved aspects like multi-modality and hierarchy. In order to make semantics clear, a formal definition for hierarchic graphs is presented, which is inspired from [45].

Chapter 5 finally presents the framework *TransRoute* more detailed, especially its generic core component for graph representation, but also the ontology to aptly describe the domain features along with the persistence component. All underlying design principles are shown by example, as well as the existing frameworks *JUNG* [21], *GXL* [8] and *OWL API* [13] from which the system has been assembled.

2 Domain of Transportation and Routing

Obviously, developing software operating on computers first requires an examination of the real entities and relations to be captured by the model, which is a (to some extent) simplified abstraction of reality. It is crucial to understand the subtleties of the domain itself, before encoding it into an abstract model or concrete program.

2.1 Introductory Example

All different transportation networks fulfill the basic purpose of circulating a large number of persons or goods on vehicles - special entities which have the ability move on the respective net structure. Altogether, they provide an infrastructure capable of connecting remote locations. Participants can take advantage of this existence and should easily be able to use them. In real life, when the number of vehicles gets too high, traffic jams and other effects take place which lead to increased travel times. So when thinking of traffic, for the first moment, one may get a picture of crowded streets and traffic jams in large cities (see figure 2.1).

However, traffic is not confined to this narrow realm, but additionally involves plenty of other means of transportation, like busses, tram, underground, railway, planes, ferries, cars, taxi, bicycles etc. Even pedestrians are participating in traffic as they are walking on pavements from one location to another. In any case, traffic suggests the very fundamental idea of mostly coordinated, sometimes maybe chaotic movement, irrespective in which concrete way.

Imagine a businessman on the way to an international conference in an enormous city on another continent, like New York or Hong Kong. Planning such an international travel is not as easy a task as it might seem at first glance. The question to be considered is following: Which transportation modes are best suited for the businessman?

In order to give an appropriate answer, one has to reflect on several important aspects: From the broad

Figure 2.1: A Complex Highway Network (from [50])

variety of possibilities reaching from public transportation (including busses, trams and underground as the most common representatives) to taxis, cars, railways, airplanes, ferries (for metropoles like London, Paris or Venice) and many more, one can surely discard some modes like bicycles or roller-blades, which are obviously not suited for this task. Yet from the remaining reasonable options, the problem is to pick the best ones for an optimal choice.

Secondly, not only location and accessibility of transportation modes are vital aspects of travel planning, but also the total time to be elapsed certainly is. According to [25], ”distance is absolute, while time can vary” (by factors like congestions etc.). Given the fact that his conference is of utmost business importance, any delays on the arrival shall be omitted. So the choice will be made to safely arrive at the conference location, leaving some extra time for any unsuspected events that might occur in the last minutes, by determining the right plane or train from a schedule. Last but not least, costs are also a relevant factor. However, in this case, the businessman's employer is likely to pay the trip. Each mode of transportation differs in costs. In some cases, fees are charged for usage, predominantly in public transportation. As a contrast to this, travelling by car is paid at petrol stations. Fuel costs depend on distance, the vehicle used, the individual driving style but also on economic factors.

Factors influencing this optimal choice may also depend on the considered person himself. If the businessman is afraid of flying, he may discard the option of aeroplanes and stick to the other remaining possibilities, e.g. a ferry (in a city like London), taking into account some disadvantages. So individual human characteristics are in a certain way contributing to the overall choice, which might be captured in a profile of the respective person or group.

Never having been to New York before, our traveller might get lost at JFK, one of the most famous airports worldwide (Heathrow, Schiphol and may other airports may also serve as good examples for complex airports). Inside the building, many elevators and conveyor belts transport people to their terminals. For boarding issues, one will often encounter shuttle busses loading passengers on board. As a conclusion, trip planning should also include the feature of guidance through the interior of a rather complex building, showing the simplest ways perceivable for a human being through modern mazes like airports.

This brief introduction aims at inclining the reader's imagination to grasp the highly complex domain of transportation and routing in its entire extent. A rather abstract view from above is taken, looking at the underlying general principles. Whenever necessary, these concepts will be further illustrated by examples. At the end of this chapter, for instance, a concrete application scenario will be introduced, involving travel between Munich, Malta and Marseilles.

2.2 Transportation in a Nutshell

"The possibility of movement in a society has permanently been related to its cultural and technological progress." [25]

It is statements like this one which underline that transportation has always been and still is pervasive in every country on the globe as an indispensable service. This section illustrates what transportation is all about. *Transportation Geography on the Web* [25] is a good starting point, offering a lot of material for further reading. In the following, some commonplace terms of the domain are presented (see also [25], [73]).

2.2.1 General Notion

Transportation is a mean for "overcoming space" [25] in order to connect different locations by motion. It can be considered as a kind of **service** provided by government or some commercial company, offering either the infrastructures or the mobile vehicles on them, sometimes both. In most countries, for example, the street infrastructure is built by the local government and shared between different modes of transportation. Most of the streets being owned by the state, private ones with limited accessibility only form a minor part of it.

An ideal transportation provider would offer instantaneous, unlimited and cost-free access at any time. As an example, many science fiction authors write about teleporting. Nowadays, realistically, any endeavour to reach this utopian view of transportation is confined not only by political and economic interests, but in the same time by **physical constraints**, most notably spatial division and the time required to fill this gap.

In **transportation systems**, which are built to provide transportation, a movement of people and freight is taking place. Physically, this motion takes place on links between nodes of the transport network, hence this is the point where graph theory will come into play. The main focus of this thesis being on the routing aspect, persons are of special importance, albeit most of the concepts to be elaborated should be applicable for freight transportation as well. Regarding transportation in terms of a service, the intent is to meet peoples' basic need for mobility as well as the general desire of accessing points of interest. So high connectivity and accessibility is an important aspect of this provided service.

[25] names out four major types of problems and constraints affecting transport systems:

”**Capacity** [is] a basic constraint, ... both along a transport route and at a terminal. The capacity of a transport system is often restricted by its circulation bottlenecks.

Transfer. Transfer points are crucial as they permit the interface between different transport systems, a role commonly served by hubs or gateways.

Reliability ... concerns the expectation that a movement will occur within a specific time and cost range. While a route could be shorter, it may not be as reliable as a longer route. Congestion is a common factor [indicating] the lack of reliability of a transport system.

Integration. Exploiting the benefits of each transport mode so that flows become more reliable and/or less costly. This goal is sought by intermodal transportation.”

Remarkably, the basic notion behind transportation is meeting peoples' demand for movement by

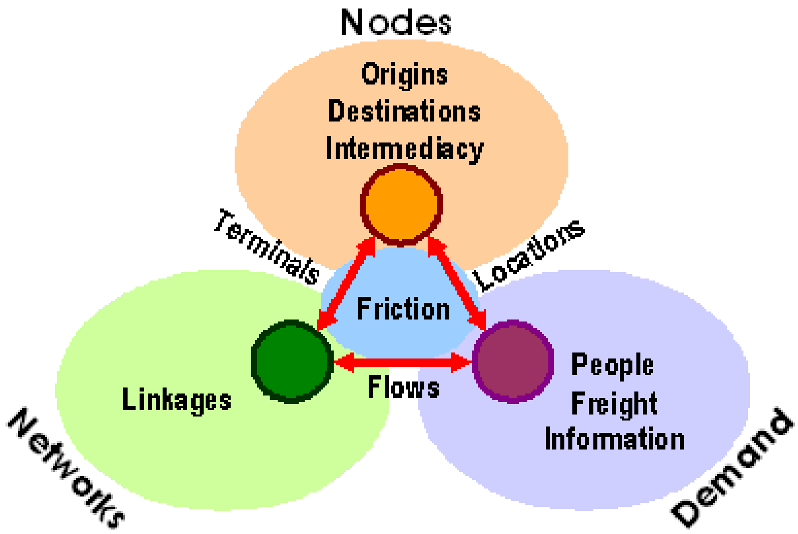


Figure 2.2: Transport Systems In Theory (from [25])

supplying adequate transportation systems making it possible to overcome greater distances, and thus connecting otherwise unreachable locations. This idea inevitably shows the influence of economy on transportation, both being interconnected and depending on each other. Evidently, transportation demand and supply are obedient to the laws of economy. A trend of falling prices (e.g. for inner country flights) for the recent years can be observed, indicating an augmenting demand. On the same instant, social aspects also play a major role, as governments are interested in traffic. Moving from one location to another should be possible for a large number of participants without major incidents, collisions, delays or any other unforeseeable obstacles.

Traffic comprises all sorts of motion on the various transport systems, whose problems are studied more deeply in the field of operations research.

2.2.2 Forms of Appearance

Transportation can be further categorised into different **transport(ation) modes**, each having its own characteristics. The first distinction is between land, sea and air, depending on where the network, especially its nodes and links are geographically positioned. By nature, in maritime and aerial networks, connections are not clearly defined. Air corridors in theory are freely available, yet restricted by international laws and regulations. Nodes, namely airports and shipyards, play an important role in that context: Dealing with mass-transit, these **terminals** indeed are a key factor for efficiency (the amount of people which can be conveyed, and the frequency and speed they can travel at).

Multi-modal transportation involves several networks of different transport modes, uniting them at common points. A mode of transportation can be changed at these important nodes, sometimes referred to as **terminals** or **transit points**. Whenever different modes meet together, competition arises, sometimes forming a cooperation in case of overlapping economic interests. As a result, and from the view of the customer being positively, he has the right to choose.

Another aspect is **scale**, which varies from local to international level. Depending on the total distance of a connection, distinct transport modes can be related to it:

- Intercontinental: long haul aircraft, high speed trains (in case of no maritime division).
- Large distances: (several thousands of kilometres) aircraft, railway, maritime
- Intermediate distances: (several hundreds to a few thousand kilometres) railway, aircraft, or car
- Regional distances: (e.g. between nearby cities, from city to airport) train, car, public transit
- Local distances: (in a closed area like a city) car, public transit, bicycle or pedestrian

This approximate mapping of according transport modes in function of the total segment length can be integrated into a hierarchical structure, by giving a distinct order of transport modes for growing distances. (see section 4.4)

Furthermore, transportation can be characterised by opposite pairs:

- **Public vs. Private**: The distinction between networks supervised by governments and those carried on by private companies. In most cases, being collectively owned by the same municipality, public modes are run together, whereas private providers are more significantly concerned with competition.
- **Fixed vs. Flexible**: Most public transportation modes are much more restrictive than their privately owned counterparts. On the one hand, the routes being offered are in most cases of a rather simple nature (mostly even linear, e.g. underground routes) compared to a complex, meshed street network. Not only is there a spatial restriction, but one also encounters a temporal confinement, which implies the movement of a vehicle along a link only at certain times being

organised in schedules. Vehicles spend a certain amount of time (so-called layover) at a terminal before heading towards the next one. Airplanes and trains also follow a strict timetable. However, divergence of these prearranged schedules appears in some cases (emergency or other important issues) demanding an approach with more adaptivity keeping information up-to-date. On the other hand, circulation of automobiles (on a street network inside a city or on a highway network connecting cities) offers a much greater freedom and thus flexibility of routing. From parking lots or garages inside buildings, where cars spend most of their lifetime being parked and ready for use, the voyage goes to other parking lots (or garages) on different sites.

- **Autonomous vs Non-autonomous**: Flexible and autonomous transportation is mostly convergent (i.e. for bicycles, cars and pedestrians). However, taxis are an important counterexample for showing that both terms are not necessarily synonymous. When the motion is actively performed by the person himself, one speaks of autonomous transportation, while being driven around by other persons means passively participating in the transportation system, hence being non-autonomous.

2.2.3 Dynamic Phenomena

Traffic as a dynamic process occurs on transport networks with a limited capacity. Commonly, the number of vehicles per hour on a certain segment is the unit of measure used. Within the boundaries of this maximum capacity, flow as a movement being triggered by a demand-supply interaction [47] takes place.

Among other causes, urbanisation and the development of agglomerated areas has lead to a variety of notable phenomena. Especially, complex traffic systems have evolved in clusters with a high density, for example in Japan. In order to cope with dynamic effects and successfully manage traffic even in difficult situations, sophisticated traffic control and assignment systems have been introduced there. Reacting to unforeseen situations in the right manner requires a real-time system.

Intelligent transportation systems (ITS, not to be confounded with intelligent traffic simulations) have been developed for this purpose, which are highly complex and therefore expensive. Some are even capable of regulating traffic dynamically when appropriate. Commuting as an everyday phenomenon of modern life invariably leads to a multitude of undesired effects, especially in areas which are already highly crowded and thus even more vulnerable.

Several important components for a dynamic ITS model are explained by [75]:

- Traffic prediction within-day based on the time periods morning, home-to-work, midday, non-commuting, work-to-home, evening and night. It can be further extended by day-to-day dynamics [47], e.g. when holidays are starting in a country. Prediction is often based on large amounts of data from surveys.
- Real-time route guidance reacting to events instantaneously.
- Integrated traffic control system. In contrast to static control measures (e.g. traffic signs, pricing and tolls), dynamic ones (predominant in Japanese metropoles) need to consider current flow and react case-sensitive. Therefore, being far more sophisticated, they are called intelligent. Notably, the entire problem can be regarded from a standpoint of artificial intelligence: multi-agent systems, as proposed by [72], are a viable mean for modelling this problem.

The complex dynamic model consists of information on link length, free flow speed, speed limit on segments, length of weaving areas, types of ramp control, number of bus lanes, one-way lanes, intersection control data (turns, fixed vs actuated signal, major minor priority).

Probably, the most typical of those dynamic traffic patterns one can think of is congestion. It most often takes place in agglomerated cities and around other accumulated areas, occurring only over a distinct period of time. Rush hour is an apt term to describe this aspect of temporal limitation. The repeated motion of masses from home to work in the early hours and back home in the beginning evening can hardly be handled satisfactorily. This causes an increasing stress level, which may lead to accidents, still worsening the situation and finally resulting in a vicious circle.

By periodical and regular measurements of traffic density on a segment, an appropriate capacity of this particular street part can be observed, and thus the current traffic flow can be compared to the value of capacity. The ratio of current traffic flow and maximum capacity may serve as a congestion indicator of factor.

Not only does congestion lead to longer travel times by drastically reducing average speed and hence reducing the overall transportation efficiency, but it also has an ecological impact (at least for cars, but certainly not for bicycles, trains or other 'clean' transportation modes [73]). An agglomeration of cars with running motors pollute the surrounding air by emissions of carbon dioxide and other harmful substances. Hopefully, the trend towards 'cleaner' cars introduced by exhaust emission control and followed by filtering of soot particles will prevail (see [73]).

Among other impediments negatively affecting travel time, works due to repair (i.e. for freeways) are to mention. Over a certain period, a street section might even be closed for traffic. Sometimes, a restriction in the number of lanes takes place at certain times of a day, or in order to guarantee safe transit, a tunnel may only be passed at certain times. So capacity may also vary as a function of time in these cases. All these constrictions may lead to a slowed down traffic through these narrowings.

Notably, in its nature, congestion is dynamic and therefore highly has the tendency to spread by moving back or forth over time. The effect of spill-back [75, 80, 72] is enhanced as the duration of congestion is getting larger, whereas the traffic flow increases or even remains constant over time. As queues are getting longer and longer in the tail, the first ones can accelerate again. Over time, congestions can grow or shrink, shift in their local extent (move forward or backward, local to regional extent). One has to be aware of human behaviour: in stressful situations like congestions, accidents are likely to happen.

2.3 Routing

2.3.1 Finding Paths and Chains

Irrespective of the current application area, routing is always concerned with finding an optimal way to connect an **origin-destination pair** (also known as **source-target**), which is an ordered tuple of two nodes or points in the considered network system.

From a computer scientist's point of view, the term 'routing' is primarily associated with the rather physical task of directing and controlling packets of information along a computer network from an origin to a destination (in this case both computers). Routing in this category is done by network protocols on a higher level, supported by the network structure below [33].

Now considering transportation networks, routing is a sub-problem in the field of traffic assignment. Like in computer networks, the activity of routing can be described in terms of a search: Given the exact knowledge of the network in question (hence its topology), routing is done by looking for a suited shortest path (or as a synonym: route) from a given location to another in a network which can be further constrained (the mandatory fulfillment of certain criteria like cost, distance, time, .. or personal preferences). Not only does routing appeal to transportation networks: in any sort of network, routing is a crucial activity substantially improving its functionality from the user's perspective by hiding the structure 'under the hood'. The structural information is employed for finding a distinct path as a sequence of related connections forming a probably very small part of the network. In other words, routing helps users to cope with entire and probably highly complex structure by abstraction.

Another parallel can be drawn with a sub-domain of artificial intelligence: Planning a robot's movement [84, 51, 66] likewise employs methods from graph search (see chapter 3). From all possible connections, a robot controlling component has to discard several options which contain obstacles in order to get as quickly as possible to the desired target (i.e. to move along the shortest path from source to target). Despite all sorts of possible obstacles along the way, the robot should finally find its way through the world (perceived by sensors) to the desired target by successfully avoiding all of the former. Yet another striking example is a search engine on the World Wide Web, where the local machine is equivalent to the source node, and the result page with the highest ranking is the target node (thus shortest path criterion is highest ranking). By filtering and ranking a great amount of information for each user individually (personalisation [71], often in terms of portals), these shortest path algorithms can be parameterised.

2.3.2 A Users' Perspective

Due to the fact that in the discussed domain of transportation, humans and not machines are of utmost importance, adopting a users' perspective is essential. Instead of coordinating the execution of movements (for robots), one has to adequately recognise the subtleties of human behaviour, which may be even more complex a matter than controlling a robot. This point becomes evident when thinking of human cognition and interpretation [78, 71, 68] in contrast to robot sensors. On the one hand, robots movement is complex because of the interaction with the environment, which is not known in advance, but has to be perceived by the sensory system of the robot. A "cognitive map" [66] (which every human has by individual perception) has to be built in order to cope in the mainly unknown environment.

But then, a machine cannot be afraid of walking in a park late at night, neither can it avoid flying 'merely' due to fear. All sorts of feelings and non-verbal preferences, likes and dislikes are difficult to measure or capture as they are soft factors. Classic predicate logic is not capable of handling such concepts, however fuzzy techniques can model inclinations. Accordingly, an efficient search algorithm alone is not sufficient for realistic routing applications, but rather yields fruit when being combined with ”knowledge-based problem solving [which] emphasises the use of human problem solving strategies on a computer” [68].

Individual abilities and disabilities are to be considered as well, but also other factors like reductions and discounts (people working for railway companies etc.) and everything users do already have, be it a vehicle (car or a bicycle) or a season ticket (e.g. for public transport). For the point of view of transportation as a service, users aspire both high reliability and safety (in terms of scarce accidents), virtually permanent availability, low prices and much more. Personal likes and dislikes, predominantly expressed in modal preferences, also have to be taken into account. Altogether, everything that could actually influence the reasoning task has to be contemplated. So offering as much as possible to satisfy the users' needs is preferable: One can distinguish different kinds of users, so-called roles taken over, extending from leisure activities, travelling and tourism to business purposes.

2.3.3 Limitations of Existing Systems

On the Internet, on can find plenty of **map servers** offering routing services and travel assistance, based on informations encoded as electronic maps, such as [5, 10, 9, 3, 87, 26] among many others. These computer-aided trip planning systems have two basic functionalities:

First, by offering maps of countries and cities with transportation networks (mostly roads) on them, travellers get an orientation. Different scales are offered, enabling zooming for a closer look. On a larger scale, because of the total size of the map on the screen, not all information is displayed, but only the most essential one (e.g. main streets as arterials in city). When looking closer by clicking, the smaller ones will be displayed, too.

Second, more important for this work, a routing service is offered. Users can insert information like origin and destination, and sometimes also criteria for the path to be found (a shortest or cheapest path). For persons looking at unknown maps, a simplest path may well be of practical value. Essentially, every suchlike system asks for an **origin-destination pair**:

Mappy **[10]** can find these locations not only by address, but also by district or underground station or other locations. It offers routing for both car and pedestrians.

Map24 **[9]** enables users to seek for parking lots, restaurants, metro station and other points of interest along a route. Ferries transporting cars are also integrated in this solution.

Google maps **[5]** offers a switch from satellite picture to map. Locations and restrictions upon those can be freely inserted as text, form which some important keywords are extracted to find the matching geo-location (e.g. JFK for the famous airport of New York).

Falk and Via Michelin. In both *Falk* [3] and *Via Michelin* [26], intermediate stations can be inserted in a fixed order (sequence).

All these systems are mostly useful for persons having cars, however there are some deficiencies when considering different transport modes. So truly multi-modal travel planning is not realised to date in the form desired, which would mean integrating heterogeneous informations of different providers into a uniform model so that seamless transitions between these different modes of transportation (e.g. public transport with park-and-ride facilities) can take place. Nowadays, integration for multi-modal transportation is not yet offered in a satisfactory manner, but often, a cumbersome linking of information is required by hand (for instance when looking up a train schedule, and matching it with the timetable of an airport). The gist of this all is that semantic data is still lacking today, making it harder to process various separate data sources. In conclusion, the need for knowledge-based methods and semantic data processing is present.

Thus, from the vision sketched above, the idea is to offer a system capable of comprehending multiple transport modes, and in the same time integrating data / information from different heterogeneous sources. Being open to new technologies like magnetic levitation trains and those which will be invented in the future is desirable. The aim of this work is to contribute to the construction of a such system, which will be able to handle both multi-modal and hierarchical networks (see 5.1.1).

As stated above, travel planning is an issue matching in a natural way with routing. Navigation, on the one hand, implies guidance with instructions (for directions) when sitting inside a vehicle already on-route for the destination. On the other hand, routing can be seen as assistance in the early planning stages of a voyage, where its proposal for a path in the underlying transportation system from an origin to a destination location can more easily be dismissed. So it not necessarily has to be adopted, as one is not bound to any mode of transportation yet.

One can perceive the movement of vehicles from an origin to a destination node as a flow through the transport network (traffic). Likewise, sources and sinks are describing this movement as a flow, where a source offers a supply which is missing in the sink. Following this physical metaphor, it inevitably invokes the notion of potential differences between nodes being the fundamental cause of flows. Many more metaphors and models provide interesting ideas, such as gravity models and propagation of waves along potential fields [84].

Since selection of a route from a possibly complex network essentially uses some kind of computation, the goal is to find an optimum with respect to a weight function reflected by some attributes of importance, which in turn are extracted from the user's profile. "In human nature, the process of selection is imprinted by effort minimisation" [25], which emerges from the surface in many ways: most importantly saving money and time, but also finding various short-cuts like crossing a lawn instead of using a given path in order not to be late can be observed generally where larger areas with more buildings are attended by numbers of people. Minimum cost and in the same time maximum efficiency perfectly describe the optimal path to be looked for. Depending on each individual with his characteristics, the weight function varies. A generic concept for defining weight functions and associating them with attributes depending on preferences from a user's profile is adorable. By this generic approach, adaptivity to individual demands imposed by different future applications should be made possible.

2.4 Scenario for Applications

As the European community grows more closely together, the cooperation of its universities is increasing. When planning business trips between different countries in Europe, in order to visit foreign universities or participate in workshops and conferences, routing is an essential part of helping persons to arrive at the desired location.

So for the department of programming and modelling languages of the University of Munich together with its partner universities in Malta and Marseilles, facilitating the planning of business trips is a first goal to strive for.

The interesting question concerning what possible routing applications there are could be further examined. For the future, **location-based services** are not negligible, especially when thinking of mobile devices. Semantic attributes concerning possible points of interest could be read from a user's profile and matched with the current position tracked by global positioning systems in order to give location-sensitive routing information to the end user.

3 Principles of Graph Theory

Since its beginnings, graph theory has always been a branch of mathematics of particular practical relevance. In this chapter, general applications and important concepts of graph theory are introduced. It primarily emphasises its various mathematical definitions and notions, ranging from adjacency to trees. By far, the chapter is not comprehensive, not including all aspects and definitions of graph theory. Since further readings are abundant [55, 50, 59, 38, 49, 35, 52, 82, 61], this treatise is confined to the small part which will be required for the understanding of chapter 4.

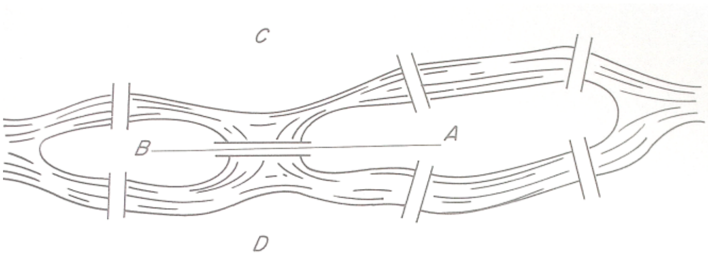
3.1 Practical Applications

Graph theory as a mathematical discipline deals with various problems which can be encountered in everyday life. The basic structure of a graph consists of a set of nodes / vertices on one hand and a set of edges as a binary relation specifying connections between the former on the other hand. A graph being a conceptual structure, nodes and connections are not necessarily of physical nature, but may well be abstract. This point clearly is an advantage because of the various possibilities to represent graphs: for visualisation, a different one can be chosen than for internal representation on a computer. (see section 3.6). Consequently, graphs are suited both for conceptualising problems of diverse fields and for modelling networks, most prominently the Internet. "Bring[ing] many .. ideas out of the [abstract] realm of mathematics .. into the world of practice [, graphs are pervasive]." [49]

3.1.1 Modelling Structures

Graphs are well-suited to model static structures, especially (however, not necessarily) of physical nature:

Chemistry: modelling complex molecules with graphs. Saturated hydrocarbons (methane CH_4, ethane C_2H_6, propane C_3H_8 etc.) can be seen as graphs (more precisely as trees) with central



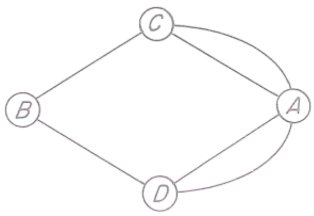


Figure 3.1: The Koenigsberg Bridge Problem (from [49])

chains of carbon atoms surrounded by hydrogen atoms. [49]

Electricity: For analysis and design of electric circuits and chips (VLSI) in production, graph theory can be employed. [49]

Facilities design: A block plan layout can be seen as a planar graph. From this special subdiscipline of graph theory, results can be used for an intelligent arrangement of rooms inside a building. [49]

Robotics: For motion planning of robots, graphs are modelling planning space including obstacles and metric information. This way, the incrementally enriched cognitive map of the vehicle is represented. [84, 51]

Internet: The World Wide Web can be naturally modelled by a huge graph where documents or web sites are nodes and hyperlinks are edges. [61]

Social Networks: Particularly sociologists are interested in the structure of human societies. In their graphs, often known as ”sociograms” [6], persons are modelled as nodes and friendship or acquaintance as edges. [61]

Transport Networks: Most importantly, transportation can also be modelled by means of graphs.

Physical connections are represented by edges, whereas locations, intersections and stations are by nodes. [25]

3.1.2 Optimisation

Not only can graphs model structures, but problems adhering to these can be formalised. Facets of graph theory are useful to solve problems in manifold disciplines:

Assigning n different jobs to n persons with different capabilities can be modelled as a bipartite graph (two disjoint node sets) as a special case of a tournament [50]. This problem shows that graph theory indeed has a lot in common with combinatorics. Especially when complete graphs K_n are considered, these similarities become apparent. A typical practical application would be allotting students to classrooms when writing an examination.

Network of telephone lines between cities: How many redundant lines have to be added in order to reduce interruptions and cut offs by 80% ? [50]

Suppling houses with electricity, gas and water can be modelled with graphs. [50] More generally, these and similar problems of connectivity can benefit from techniques like determining minimum spanning trees.

Interferences between radio stations can be measured such that senders can be set up in appropriate distance. [50]

Determining cheapest flights between pairs of cities by a price matrix. [50]

Travelling Salesmen are preoccupied to return to their home city after having visited each city of their tour once. In order to reduce both travel costs and time, they are seeking to find an optimal tour in terms of distance. The travelling salesman problem is prominent for graph theory. [50, 34]

Transporting freight from n supply depots to m retail stores in a street network with edge capacities most economically. This issue is known as **transportation problem**. [49]

Planning the construction of **highway routes**: A highway engineer wants to construct a path from A to M with minimal cost (several alternative routes are available). A shortest path, in this case a so-called ”shortest policy” [49], has to be determined for this purpose. By looking at sub-paths or partial routes, a solution can be incrementally constructed.

Foreseeing Congestions by determining maximum safe number of vehicles per minute (flow) in a traffic network with capacities. [49]

Planning One-Way Streets in order to reduce traffic flow, yet maintaining distinct districts connected. By adding orientation, the essential task is to transform undirected edges into directed ones. [50]

Determining the Critical Path of a project for management when parallel processing of activities is considered. Within the critical path, any delay of activities leads to subsequent delays of the entire project. So the progress of the project as a whole is crucially interrelated with its critical path. [49]

In scheduling, where order is relevant, determining the right sequence of tasks is vital and can be solved with topological sorting of directed graphs. [52]

One perceives the pervasive nature of graphs as a conceptual notion, which reflects reality on many different areas and aspects.

> ”There is hardly a field of human interest that in some way cannot use the concepts of graph theory to solve its problems, or at least to provide a better understanding of them [by offering a formal, non-ambiguous representation].” [49]

3.2 History

Looking back to the early beginnings of the 18th century, the very fundamentals of this branch can be found: In 1736, *Leonhard Euler* (1707-1783) was the first to write a paper on graph theory, inspired from the contemporary Koenigsberg Bridge problem (see figure 3.1[1]): The river Pregel and two islands on it (A and B), which are connected to the main land (C and D) by seven bridges. Is there a way, from an arbitrary starting location, which crosses all bridges once and returns to the start? Euler proved that it was impossible to do so. For his proof, he used a natural modelling of the problem as a

[1] An excellent example for the two step approach taken by Euler: 1. modelling with graphs by abstraction. 2. solving the formalised problem.

graph by abstracting to nodes (the islands and river banks) and connections (the bridges), being both intuitive and mathematically precise in the same instant. [35]

By publishing this article in which he demonstrated that the problem had no solution, the famous swiss mathematician made a major contribution to this emerging new discipline, which was enriched by *Gustav Kirchhoff* in 1847 with the concept of trees and flow of currents (metaphors inspired from chemistry and physics). In 1859, *Sir William Hamilton* introduced the problems related with Hamiltonian graphs and travelling salesman by creating a game in which famous cities of the world had to be travelled on a tour. The concept of planarity can be attributed to *Kazimierz Kuratowski*. Graph theory has split into many more fields of research, namely coloring, *Ramsey* theory etc. [49]

It was rejuvenated in the late fifties by a considerable number of researchers in computer science, among whom ranked *Dijkstra*.[2] Slowly, the main focus shifted towards problem solving with foraging for new efficient algorithms. As many interesting problems can be formulated using graphs, practical applications (construction, transportation, routing, planning, scheduling etc.) were fostered by the algorithmic approach. Considering the theory's generality and high degree of abstraction, the sheer variety of uses is intelligible. Notably the special section concerning networks and flows received a major boost. An entire book could be written, only containing the names of the many researchers who have contributed to graph theory, all of whom are not explicitly mentioned in this thesis.

3.3 Basic Definitions

3.3.1 Graphs, Vertices and Edges

The subsequent definitions serve two purposes: Firstly, they hopefully will provide a basic understanding of the most important types of graphs and their properties. Secondly, by setting up a clear terminology, they ease the readings of continuative literature [55, 35, 38] on the vast realm of graph theory. As *Diestel* [55] and *Cormen et al.* [52] cover the topic of graph theory profoundly, the following definitions are inspired from their work, however mostly in short.

An undirected **graph** G is defined as a tuple

$G = (V, E)$

[2] His algorithm for determining shortest paths is shown in section 3.7.3.

consisting of two disjoint sets V and $E \subseteq V \times V$.

At this, the non-empty **set of vertices** (or nodes) is denominated V ($V \neq \emptyset$), and the **set of edges** realising the binary **edge relation** on V is denominated E. Within this thesis, $|V|$ will be used for the number of vertices of a graph G, and $|E|$ for the corresponding number of edges.

More precisely, the edge set

$E = \{e = \{v_1, v_2\} | v_1 \neq v_2 \wedge v_1, v_2 \in V\}$

consists of edges e, each one defined as a **set** of two vertices, for the order is not important.

By convenience, the notation $e = (v_1, v_2)$ [3] is often used instead of the more accurate $e = \{v_1, v_2\}$ which rather acknowledges the idea of an edge e being an **unordered** pair of vertices. In order to avoid confusions (so that one can easily distinguish between directed and undirected graphs) and make definitions unambiguous, the latter, exact notation will be used throughout this thesis.

According to the definition, v_1 and v_2 are distinct vertices, which means that self-loops (any edges from one vertex to itself) are forbidden. For directed graphs (see section 3.4), however, this restriction does not hold any more.

In general, both sets V and E may be infinite. Yet for computational reasons, when speaking of graphs in this thesis, V and E are always implicitly limited to **finite** cardinality. By definition, the set V cannot be empty. Yet if E **is empty**, i.e. the graph only consists of a set of vertices, G is called **degenerate**. [49]

Incidence is a relation between vertices and edges. Both vertices v_1 and v_2 from the previous definition being **end-points**, they are **incident on** the edge e connecting them. A relation (in the sense of a formal logic system) *incident* indicating that the vertex v is incident to the edge e can be defined:

$\forall v \in V, e \in E : incident(v, e) \Leftrightarrow e = \{v, \star\}$[4]

Adjacency. Basically indicating direct neighbourhood, adjacency is another important relation, defined for two different vertices. From the basic definition of a graph, since both v_1 and v_2 are on the common edge e, they are **adjacent to** each other.

[3] Irritatingly, the same notation will be employed for directed graphs. However, in that case, the order is of importance.

[4] $\star$ is used as a wildcard symbol for any arbitrary vertex $w \neq v \in V$.

$\forall v, w \in V : adjacent(v, w) \Leftrightarrow \exists e \in E : e = \{v, w\}$

Analogously, two edges $e1$ and $e2$ can be defined as being adjacent to one another:

$\forall e1, e2 \in E : adjacent(e1, e2) \Leftrightarrow \exists v \in V : v \in e1 \wedge v \in e2$

This kind of formalism has several advantages. First of all, the formal mathematical definition is completely independent of any visual representation, thus abstracting from it. It is this strict separation that allows various ways of graphical representation of the very same graph formally defined as $G = (V, E)$. That is to say, a formal graph being entirely decoupled from its actual representation, a fitting one can be chosen satisfying the special needs. In section 3.6.2, the notion of visualisation is taken on further.

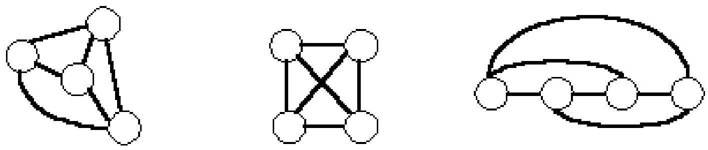


Figure 3.2: Three Different Representations of the Same Complete Graph K_4

Beyond, when a visualisation is given, the underlying graph can be derived from it. So very different graphical representations of the same graph, which cannot easily be identified as being equal by humans, can be represented in an unambiguous manner (see figure 3.2). So it possible that from different application domains, which all use graphs as modelling structures, commonalities (which may not be obvious at first glance) can be concluded, consequently fostering interdisciplinary exchange of knowledge.

For example, a methane molecule (see example of chemistry in section 3.1.1, figure 3.3 [5]) can be formally represented by a graph $G_M = (V_M, E_M)$ with $V_M = \{c, h_1, h_2, h_3, h_4\}$ and $E_M = \{\{c, h_4\}, \{c, h_2\}, \{c, h_1\}, \{c, h_3\}\}$. Here, G_M has five vertices and four edges: $|V_M| = 5$, $|E_M| = 4$.

Regarding **density** of edges, compared to the number of vertices (ratio $\frac{|E|}{|V|}$), a graph G is called..

- .. **degenerate** if $E = \emptyset$ (only isolated vertices).
- .. **sparse** if $|E| \ll \frac{|V|(|V|-1)}{2} = O(|V|^2)$. Being sparse, G is best way represented by adjacency lists (see section 3.6.1).

[5]The hydrogen atoms are distinct vertices and have to be numbered (e.g. from h_1 to h_4).

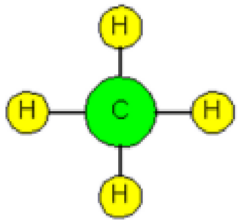


Figure 3.3: A Methane Molecule Modelled by a Graph

- .. **dense** if $|E| < \frac{|V|(|V|-1)}{2} = O(|V|^2)$. In this case, the adjacency matrix is the best choice for representation (see section 3.6.1).
- .. K_m or **m-complete** with $m = |V|$

  if $|E| = \frac{|V|(|V|-1)}{2}$ (formula originating from combinatorics), i.e. and each pair of distinct vertices is joined by an edge. For example, K_5 consists of 10 edges, and K_6 of 15 edges. Interestingly, K_1 to K_4 are planar, yet K_5 can't be drawn on a plane any more [38] (see section 3.6.2).

By subsequently populating a degenerate graph G by adding edges, it first becomes sparse, then dense until finally becoming complete. Since the total sum of vertex degrees (number of incident edges for each vertex) in the graph is depending on $|E|$ (see equation 3.1), the density of a graph can also be extracted from the average degree of its vertices (see section 3.3.2), for instance as the arithmetic average $\frac{\sum_{v \in V} deg(v)}{|V|}$. [55]

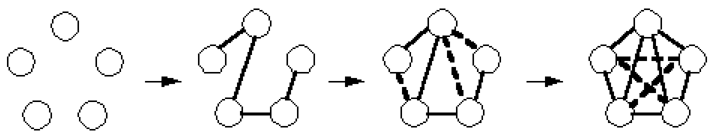


Figure 3.4: From a Degenerate Graph to K_5 by Successively Adding Edges

Taking up the example again, the methane molecule G_M has a ratio of $\frac{|E_M|}{|V_M|} = \frac{4}{5} = 0.8$ ($|E_M| \ll 10$), so it can be classified as a sparse graph. In G_M, the vertex c has a degree of four, and all other vertices h_1 to h_4 have a degree of one. Thus, the average degree of G_M is $\frac{4+1 \cdot 4}{5} = 1.6$. This result is not surprising at all, since equation 3.1 expresses the equality of the total sum of degrees of a graph to $2|E|$ (see section 3.3.2). In order to transform it into the complete graph K_5, six more edges have to be added in total. The complementary graph G_{complM} of G_M consists of these six missing edges and all five vertices of G_M.

3.3.2 Degree and Connectivity

The **degree** or valency $deg(v)$ **of a vertex** v is the total number of edges being incident to v, formally defined as

$deg(v) = |\{e|e = \{v, \star\}\}|$ [6]

For each undirected graph $G = (V, E)$ not comprising any self-loops by definition, the following important equation holds [50]:

$$(3.1) \quad \sum_{v \in V} deg(v) = 2|E|$$

As noticed in the methane example, the **average degree of a graph** is a global metric for edge density: $deg(G) = \frac{\sum_{v \in V} deg(v)}{|V|} = \frac{2|E|}{|V|}$

The overall sum of all degrees in the graph is twice the number of edges. As by definition, self-loops are prohibited, each edge is counted into the degrees of two different vertices.

When dealing with directed graphs, since self-loops are allowed, one has to distinguish between in- and out- degree of a vertex depending on the directions of the edges (for the first, positive incidence is counted, whereas for the latter negative incidence as defined in 3.4.1).

Another important aspect is the **connectivity** of a graph, which can be determined by basic graph search algorithms, namely depth- and breadth-first search (see section 3.7.1): It is the edge relation basically defines connectivity of a graph. By calculating the transitive hull of this connectivity relation and applying it to every vertex, one gets the connected components of a graph as separate equivalence classes containing the respective sets of vertices in this extended relation of reachability.

3.4 Variations

3.4.1 Directed and Weighted Graphs

Until now, when speaking of graphs, edges were defined as being undirected. However, there are cases in which a distinction of directions constitutes valuable additional information. Considering directed graphs, the edge relation in general is not symmetrical, as by definition edges have an orientation. In

[6] $\star$ is used as a wildcard symbol for any arbitrary vertex $w \neq v \in V$.

contrast to undirected graphs, self-loops[7] are possible, i.e. $v_1 = v_2$. Apart from this, the definition is very similar to the previous one for undirected graphs, except for one further point: arcs, which are **ordered** pairs of vertices, are replacing edges. For display, as arcs are distinguished by having an orientation, they are being drawn by arrows.

A **directed graph** (digraph) $G_D = (V, A)$ consists of a set of vertices $V \neq \emptyset$ and a **set of arcs** $A \subseteq V \times V$ (ordered) defined as

$A = \{a = (v_1, v_2) | v_1, v_2 \in V\}.$

Every element a of the arc relation A is an **ordered** pair of vertices (v_1, v_2) that connects the **start vertex / tail** v_1 to the **end vertex / head** v_2 [50]. The arc a entering its head, v_2 is **positively incident** on a.

$\forall v \in V, a \in A : incident_{+}(v, a) \Leftrightarrow a = (\star, v)$

Analogously, the tail v_1 is **negatively incident** on a.

$\forall v \in V, a \in A : incident_{-}(v, a) \Leftrightarrow a = (v, \star)$

Adjacency is defined in a similar way as for undirected graphs:

$\forall v, w \in V : adjacent(v, w) \Leftrightarrow \exists a \in A : (a = (v, w) \vee a = (w, v))$

$\forall a1, a2 \in A : adjacent(a1, a2) \Leftrightarrow \exists v \in V : incident(v, a1) \wedge incident(v, a2)$

So an arc $a_{2,4} = (v2, v4)$ is by no means equal to an arc $a_{4,2} = (v4, v2)$, but rather they are distinct.

Obtaining an undirected graph G from G_D can be done by removing orientation of arcs, which are thus becoming ordinary edges. As a consequence, the edge relation becomes symmetric, and G is called **underlying graph** of G_D.

Connectivity also slightly differs from the definition for undirected graphs, since directed edges have to be taken into account.

A directed graph G_D is called **weakly connected** if its underlying (undirected) graph G is connected. Beyond, a directed graph G_D is **strongly connected** if additionally, for every pair of vertices there exists a (directed) path to each other (see section 3.5.2), i.e. every pair of vertices is mutually reachable.

[7] Edges whose start and end point are the same vertex.

Whereas a **weak component** $C_w(G_D)$ of a directed graph G_D is an equivalence class within the transitive hull of the edge relation pertaining to its underlying graph, strong components are characterised by equivalence classes of the transitive reachability relation in the directed graph itself. Basically, a **strong component** $C_s(G_D)$ of a digraph G_D is a strongly connected subgraph of G_D, without being a subgraph of any other strongly connected subgraph of G_D, hence is atomic.

It is useful for directed graphs to distinguish between **in-degree** and **out-degree** of a vertex. The in-degree $deg_{in}(v)$ of a vertex v is defined as the number of arcs where v is head:

$$deg_{in}(v) = |\{a|a = (\star, v)\}|$$

Accordingly, the out-degree $deg_{out}(v)$ of a vertex v is the number of arcs in which v is tail:

$$deg_{out}(v) = |\{a|a = (v, \star)\}|$$

The (overall) degree or valence of a vertex v is obtained by $deg(v) = deg_{in}(v) + deg_{out}(v)$.

(3.2) $$\sum_{v \in V}^{n} deg_{in}(v) = \sum_{v \in V}^{n} deg_{out}(v) = |A|$$

Like for undirected graphs, combining the two parts of the equation, $\sum v \in V deg(v) = 2|A|$ holds.

A **weighted graph** is a graph $G_w = (V, E)$ together with a **weight function** w mapping each edge $e \in E$ to a certain weight represented by a real number.

The weight function

(3.3) $$w : E \mapsto \mathbf{R}$$

is formally defined, so that

(3.4) $$\forall e \in E : \exists w(e) \in \mathbf{R}$$

holds, giving each edge a particular weight.

With respect to practical issues, it is often useful to combine weighted graphs together with directed graphs, for edge weights are a natural mean for describing distances or costs attributed to an edge. Shortest-path computations, for instance, essentially rely on these kinds of graphs.

In a more general notion, it is possible to define graphs with several meta-attributes of different data types, e.g. $String$, $integer$, $boolean$, which comes close to an object-oriented view of graphs (see chapter 5.2.1). Hence, any numeric attribute could be applied as edge weight in this general approach offering more flexibility.

3.4.2 Multigraphs and Hypergraphs

A **hypergraph** additionally consists of hyperedges from a vertex to any subset of vertices (the edge relation is not binary anymore: $E \subseteq V \times V^n$ with $n \geq 1$).

In a **multi-graph**, more than one edge from a vertex v_i to another vertex v_j can be defined. The Koenigsberg Bridge problem, for example, can be modelled by a multi-graph. However, by adding new vertices wherever more than one parallel edge exists, one can define this problem in terms of usual graphs, too.

3.5 Further Definitions

3.5.1 Subgraphs and Other Relations

Beyond considering a single graph on its own, there are essential relations between two graphs which can be investigated more deeply. For two or more graphs, these relations are principally intersection and union, obtaining smaller or larger ones.

The **intersection** of two graphs G and G' is defined as

(3.5) $G \cap G' := (V \cap V', E \cap E')$

In addition, if the two graphs G and G' are such that

(3.6) $G \cap G' = \emptyset$

they are called **disjoint**, while on the other hand, if

(3.7) $V' \subseteq V \wedge E' \subseteq E$

holds, G' is a **subgraph** of G, or respectively, G contains G' as a subgraph.

The **union** of two graphs leading to a larger graph is defined as:

(3.8) $G \cup G' := (V \cup V', E \cup E')$

3.5.2 Paths and Cycles

A **chain** from a vertex v_1 to a vertex v_n is the non-empty undirected graph

$chain_{1,n} = (V, E)|V = \{v_1, v_2, ..., v_n\}, E = \{\{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_{n-1}, v_n\}\}$

in which all vertices v_i and v_{i+1} are pairwise distinct.

A chain can be obtained by a subsequent traversal of a graph from one node to its adjacent neighbour and so forth, never returning to the same node twice and covering some or all nodes. Conveniently, it can also be denoted as a sequence of incident vertices:

$chain_{1,n} = (v_1, v_2, ..., v_{n-1}, v_n)$

Beyond, it can also be denoted as an **edge sequence**, all neighbouring pairs e_i and e_{i+1} of its constituent edges being incident:

$chain_{1,n} = (\{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_{n-1}, v_n\})$

A **circuit** $circuit_{1,n}$ for $n > 3$ is a chain $chain_{1,n-1}$ extended by the edge $\{v_{n-1}, v_0\}$ (a closed chain whose start and end vertex are the same).

A **path** $path_{1,n}$ is equivalent to a chain, only defined for a directed graph. Thus, a directed sequence of arcs is called path (in contrast to chain for undirected graphs).

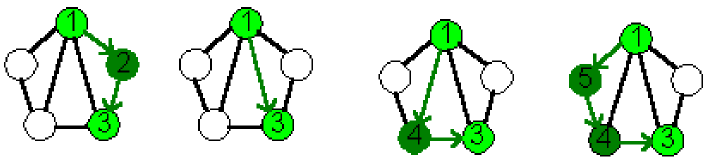


Figure 3.5: Paths From Vertex 1 to Vertex 3

A **cycle** $cycle_{1,n}$ is the counterpart of a circuit defined for a directed graph.

In an **Eulerian** path or cycle, no edge appears twice, whereas in a **Hamiltonian** path or cycle, no vertex appears twice.

Taking up the notion of weighted edges from section 3.4.1, a weight function can be extended from an arc $w : A \mapsto \mathbf{R}$ to an entire path (P being the set of all possible paths):

$w : A \cup P \mapsto \mathbf{R}$ with $\forall a \in A : \exists w(a) \in \mathbf{R}$:

$w(p_{1,k})$ with $p_{1,k} = (v_1, v_2, ..., v_k)$ being one path from v_1 to v_k and $a_i = (v_i, v_{i+1})$ is defined as

(3.9) $w(p_{1,k}) = \sum_{i=1}^{k-1} w(a_i)$

Determining a shortest path from v_1 to v_k consequently can be stated as finding

(3.10)$min\{P_{1,k}\}$

with $P_{1,k}$ being the set of all possible paths from v_1 to v_k.

From a formal point of view, a shortest path reveals interesting aspects, most notably **sub-optimality** at **decomposition** [52, 49, 50]:

Let $p_{1,k} = (v_1, v_2, ..., v_k)$ be a shortest path from vertex v_1 to v_k. A sub-path of the shortest path $p_{1,k}$ is a path $p_{i,j} = (v_i, v_{i+1}, ..., v_j)$ with $1 \leq i < j \leq k$.

Every sub-path of a shortest path has the property of being a shortest-path itself (lemma 24.1 in [52]).

The proof is straightforward: By decomposing the shortest path $p_{1,k}$ ($w(p_{1,k}) = w(p_{1,i}) + w(p_{i,j}) + w(p_{j,k})$), one can easily see that if there were a shorter sub-path from v_i to v_j then the path from v_1 to v_k would not be the shortest path, leading to contradiction.

This property inheres **Bellman's principle of optimality** from dynamic programming (divide and conquer) [49]. So regardless of the previous decisions for a shortest path, one can achieve a shortest path if the coming path combined with the previous is optimal. This allows **decomposition** of shortest-path problems to smaller ones concerning only one stage or sub-path ("sub-policy" in the example of highway route planning from [49]).

An idea for routing is to partition a complex route into simpler routes (smaller sub-routes), for which an optimum route is relatively easy to find.

3.5.3 Trees

A **tree** is a connected, acyclic graph. A **directed tree** is a connected, directed acyclic graph (DAG).

For undirected trees, the following proposition holds:

$T = (V, E)$ is a tree $\Leftrightarrow |E| = |V| - 1$

However, this proposition is valid only in the direction from left to right when multigraphs are considered. For multigraphs, the inverse proposition is not generally valid, but only for some special cases.

A **spanning tree** of a graph G is a connected, acyclic subgraph containing all vertices of G.

With respect to weighted graphs, determining a minimum spanning tree (a spanning tree with minimum overall edge weight) is an interesting problem for which efficient algorithms (Kruskal and Prim [52]) have been developed.

3.5.4 Flow Networks

Until now, when speaking of graphs, merely static aspects, namely the topological constitution of the represented infrastructure were taken into account. Beyond, there are special kinds of graphs which are also apt to model dynamic facets by including means to describe properties that can change over time:

Flow Networks are directed, weighted and connected graphs in the first instant containing a **capacity function**

(3.11)$c : E \mapsto \mathbf{R}$

with

(3.12)$\forall e \in E : \exists c(e) \in \mathbf{R}$

which can change their current flow with progressing time.

Secondly, the **flow function** is defined as

(3.13)$f : E \mapsto \mathbf{R}$

describing the dynamic flow with

(3.14)$\forall e \in E : \exists f(e) \in \mathbf{R}$

There are certain defined constraints every flow in a network must fulfill [35, 75]. First, two distinguished vertices s and t called **source** and **sink**, are selected as points determining all flows in-between.

As shown in [52], the flow network with several source and sink nodes can be reduced to the more simple one stated above. In detail, super nodes as artificial source and sink nodes are being introduced, which are connected to all actual source or target vertices, each edge weighing one unit so that a simple flow network is formed up again.

Second, different **integrity constraints** are imposed to flows at each instant concerning:

Symmetry $\forall u, v \in V : f(u, v) = -f(v, u)$

Conservation $\forall v \in V - \{s, t\} : sum_{w \in V} f((v, w)) = 0$

Capacity The current flow of an arc may never exceed its capacity: $\forall ainA : f(a) < c(a)$

Determining the maximum flow through a network without violating capacity constraints can be done efficiently by several algorithms [35, 75] (see section 3.7.6).

3.6 Representations

3.6.1 Data Structures

Adjacency is a very central concept of a graph expressing its connectivity, i.e. which particular vertices are connected to each other. From a computational point of view, in order to establish efficient algorithms tackling known problems of graph theory, compact and ingenious data structures representing graphs by implementing this basic relationship between vertices are needed. As different data structures reveal altering performance for operations on different graphs, one cannot take one structure for better than the other, except for a particular use with special types of graphs (**sparse** vs. **dense** graphs, see section 3.3.1). Each data structure has its advantages and drawbacks. By knowing the density of the encountered graphs in advance, one can choose the optimal structure:

Adjacency matrix especially for dense graphs: First to mention is the unique the two-dimensional adjacency matrix $adjMatrix : V \times V \mapsto \{0, 1\}$ of a graph $G = (V, E)$.

For any two vertices v_i and v_j, a value of 0 at position (i, j) in the matrix denotes that these two vertices have are not connected directly by any edge.

In contrast, a value of 1 at position (i, j) stands for a connection between v_i and v_j: $adjMatrix(i, j)$ = 1 if e = (v_i, v_j) $\in E$, otherwise $adjMatrix(i, j)$ = 0.

For undirected graphs, the adjacency matrix is symmetrical to the diagonal of the matrix as a connection for v_m to v_n is represented by both $adjMatrix(m, n)$ = 1 and $adjMatrix(n, m)$ = 1. However, the adjacency matrix for a directed graph is not necessarily symmetric anymore, except for when every pair of vertices is connected by two arcs in both ways. When representing weighted graphs by an adjacency matrix, its values are no longer restricted to boolean cardinality (0 or 1), but can take any real number as a value indicating the particular weight of the edge connecting both vertices.

Regardless of the number of edges in the graph, this kind of representation requires $O(V^2)$ memory, as there are redundancies (chiefly for undirected graphs, due to the symmetry mentioned before). When anticipating sparse graphs (see section 3.3.1) in applications, it would not be a sound choice to represent them by adjacency matrices, due to the waste of valuable storage. If anything, adjacency lists would probably be more appropriate, while dense graphs (see section 3.3.1) on the other hand are optimally represented by adjacency matrices, as the total need for storage is invariant to the density of the graph, i.e. its number of edges.

	c	h_1	h_2	h_3	h_4
c	0	1	1	1	1
h_1	1	0	0	0	0
h_2	1	0	0	0	0
h_3	1	0	0	0	0
h_4	1	0	0	0	0

Table 3.1: Methane Molecule Graph Represented by an Adjacency Matrix

Adjacency lists (linked lists) especially for sparse graphs: This approach is characterised by the use of $|V|$ adjacency lists: For each vertex v in V, a list is stored containing pointers to all vertices adjacent to it. In other words, $adjList(v)$ contains all vertices w for which $e = (v, w)$ is an edge in E. Linked lists are a viable candidate for implementing adjacency lists efficiently. Furthermore, adjacency lists require no special numbering of vertices.

For instance, when (v_1, v_6), (v_1, v_3) and (v_1, v_4) are all incident arcs on v_1 in G, the linked list could look like this: cell (value = 1) $\rightarrow$ cell (value = 6) $\rightarrow$ cell (value = 3) $\rightarrow$ cell (value = 4). Any other order starting with value 1 is also allowed, since order is not relevant. The starting cell contains the index of the vertex to whom all following are adjacent. Using adjacency lists, memory requirements are only $O(V + E)$ in terms of **storage** for both directed and undirected graphs, which is a highly "desirable property" [52]. However, determining whether an edge e is in the graph can only be done by searching the adjacency lists of vertices incident to e (no ordering can cause a longer search in the worst case, but maximally $|E|$ in theory).

from	to list
c	$\rightarrow h_1 \quad \rightarrow h_2 \quad \rightarrow h_3 \quad \rightarrow h_4$
h_1	$\rightarrow c$
h_2	$\rightarrow c$
h_3	$\rightarrow c$
h_4	$\rightarrow c$

Table 3.2: Methane Molecule Graph Represented by Adjacency Lists

3.6.2 Graph Drawing and Planar Graphs

In order to get behind the meaning of an otherwise merely abstract mathematical system, for human beings, visualisation is extremely important. As an ease for cognition of a complex notion, graph drawings can intuitively describe graphs in a human readable fashion. A graph on its own does not possess any information concerning how it should be represented, leaving a great deal of freedom. Planar and topological graphs serve the purpose of visualisation well, as they can be realised on a subset of Euclidean space R^n (in most cases, the Euclidean plane R^2 suffices). G can literally be shown on a two-dimensional drawing board.

In general, a **graph drawing** or **embedding** is a representation of a graph on a surface, mapping a graph's vertices to points in the 2-dimensional Euclidean space and respectively its edges to lines or a special kind of curves, so-called *Jordan arcs* [49]. Speaking more formally, a graph drawing can be defined by a **mapping function** m from a graph element (vertex and edge) of a graph G to the corresponding distinct image (point or curve) on the surface S satisfying the following conditions:

- The image of no edge (curve) contains that of a vertex (point).
- The image of an edge (curve) from v_1 to v_2 joins the images of v_1 and v_2 (the corresponding points).

A graph G is said to be planar if G can be drawn on a surface in such a manner that it is only at vertices that two different edges intersect [38]. As a consequence, a planar graph divides a surface into distinct regions. If G is embedded on the surface S, then the complement of G is a collection of open sets, called regions.

However, when modelling physical networks with real locations on the surface of the Earth, these coordinates only need to be extracted and can be used for representing graphs. In our context, the drawing is given, and a graph structure has to be extracted. This is an inverse mapping from S back to G, which is by far more simple as there are not several different possibilities but only one, which is quite obvious.

3.7 Selected Graph Algorithms for Routing

Since application fields for graph theory are abundant (see section 3.1.1), the study of efficient graph algorithms has a great impact on solving a multitude of problems from real life.

An algorithm being efficient is basically distinguished by its low computational complexity, which must be within polynomial time.

NP-completeness certainly constitutes an important issue in computational complexity of algorithms: For one major class of similar problems, researchers could neither find efficient algorithms nor prove that there are indeed none. Predominantly known as **NP-complete**, these problems are highly interesting. For example, the **travelling salesman** problem for a complete graph is known to be NP-complete.

Positively determining that a problem is classified as NP-complete automatically implies that no efficient algorithm has yet been found. Finding such for one NP-complete problem would mean solving them all in polynomial time.

Some algorithms can be considered of particular interest to the task of routing within transport networks. Relying on the data structure of directed, weighted graphs, processing routing queries can performed by appropriate graph algorithms.

3.7.1 Breath- vs. Depth-First Search

Systematically scanning a graph from a start node is an essential task. Offering this basic functionality, both search algorithms introduced in this section are archetypes for more advanced ones, e.g. Prim's algorithm for finding a minimum spanning tree, or Dijkstra's algorithm for determining a shortest path [52].

These two search algorithms **breadth-first search** (**BFS**) and **depth-first search** (**DFS**) for graphs remarkably differ in the way they traverse vertices of a graph, i.e. the order of traversal which is chiefly determined by the use of an accordant data structure.

Breadth-first search is using a **queue** to maintain a list of nodes still to visit. In contrast, depth-first search makes use of a **stack** for nodes to be visited, enabling the special technique of backtracking. DFS is particularly used for determining the strongly connected components of a directed graph [52].

3.7.2 Nearest Neighbour

From a given start vertex, finding the nearest neighbours, i.e. the nearest adjacent or reachable vertices connected to it with respect to a weight function determining the distance of edges, is an issue of special practical relevance.

Persons situated somewhere in a city who ask for finding the nearest location of interest (for instance a hotel, a museum, a theatre or petrol station) from their current location could make use of such an algorithm scanning the modelled structure of a city network enriched with thematic information using their mobile electronic devices (e.g. mobile phone). In general, the field of **location-based services** is preoccupied with such applications, including touristic matters and travelling.

Adopting a technical point of view, nearest neighbour queries can be answered using breadth- or depth-first search with additional filtering of constraints.

3.7.3 Dijkstra's Shortest Path Algorithm

Finding a shortest path (or as a variation a fastest, cheapest or simplest depending on the weight function to be used) for an origin-destination pair in a network is the essence of routing, navigation and motion planning in general [84].

In case of all arc weights being non-negative, shortest paths of a directed, weighted graph $G_D = (V, E)$ together with a weight function $w : A \mapsto \mathbf{R}$ from a single source can be computed efficiently by Dijkstra's algorithm, which is founded on breadth-first search for optimality of solution.

Starting from the source vertex s, it computes shortest-path weights following all surrounding arcs to neighbouring vertices. A set S of vertices is maintained, for which the the final value of distance to s has been calculated. [52] suggests a min-priority queue for an efficient implementation.

Relaxation is an important technique used by shortest-path algorithms: the initial distance value of a vertex is serving as an upper bound estimate for its shortest path distance to s.

Pseudo-code of Dijkstra's Shortest-Path algorithm [52]:

```
Dijkstra(GD, w, s)

//init

foreach(Vertex v: G.getVertices())
{
    v.dist = posInfinity;
    v.pre = null;
}

s.dist = 0;
S = S.empty;
Q = G.getVertices();

while (Q != Q.empty)
{
    u = Q.extractMin();
    S.add(u);

    foreach(Vertex v: u.AdjacentVertices())
    {
        //relaxation
        if (v.dist > u.dist + edge(u, v).getWeight())
                v.dist = u.dist + edge(u, v).getWeight();
        else v.pre = u;
    }
}
```

Despite the relatively efficient computation of shortest paths by Dijkstra's algorithm, which can be reduced down to $O(|V| \log |V|)$ time according to [52] when choosing Fibonacci heaps for implementing the BFS queue, there are many further enhancements, taking into account more information (e.g. geometry) [86, 60, 64]. Additional domain-specific information allows speed-up techniques to decorate the basic algorithm, hence still improving its run-time performance for these special cases.

3.7.4 A* Algorithm and Heuristic Improvements

Heuristic algorithms in general are characterised by a trade-off in favour of computational speed at the loss of an optimal solution [65, 64]:

A heuristic for routing basically is a " 'good algorithm' for seeking a route, which will not guarantee optimality, but a relatively good/acceptable solution in sufficiently short time." [65]

For instance, in the study of shortest paths for realistic transport networks by [64], search is accelerated by preferring certain paths which match criteria imposed by the heuristic in use (e.g. a geometric speed-up technique pruning vertices which are in the opposite direction of the target vertex looking from the start vertex). As a consequence, the search is narrowly focussed (no more in the entire breadth of the large network), ideally degenerating from breadth-first to a perfect depth-first search having one distinguished path as a kind of tunnel through the tree of possible paths to take.

The A* search can be considered as the classical heuristic search [84]. Being generic, it can be instantiated with the **actual distance** from the start vertex plus the **remaining estimated distance** to the target vertex. Other factors like orientation, inclusion, size may also serve a heuristic estimate function (see chapter 4).

Both quality of partial solution and the estimation of the remaining path in terms of distance are essential for the quality of a heuristic.

3.7.5 Travelling Salesman and Hamiltonian Path

The travelling salesman problem "is a prominent illustration of a class of problems in computational complexity theory which are hard to solve" [31], namely the class of NP-complete problems.

It can be stated as finding a shortest path (the weight function being cost, the path with least cost) connecting all cities from a starting city on a round trip, thus returning to the home city. In particular, as each vertex may be visited only once, a minimum weight Hamiltonian path (or cycle) is looked for.

For a restricted, weakened variant containing less than 10 nodes, computation results are acceptable [49]. Tackling NP-completeness can be done by using heuristics, of course at the loss of optimality, which may even solve this problem in a larger magnitude [34].

A similar problem to this is the Chinese Postman Problem, in which an Eulerian cycle is looked for (each edge may be visited exactly once).

3.7.6 Edmonds-Karp Maximum Flow Algorithm

The transportation problem is described by a network with distinct **source** and **sink** nodes and a flow in-between [49]. However, the restriction to one source and one sink node is sufficient, since the former problems can be reduced to such [52].

Determining a maximum flow in such networks can be done by the Edmonds-Karp algorithm, an improvement of the Ford-Fulkerson algorithm by using breadth-first search for determining shortest paths between source and sink.

Most notably, like Ford-Fulkerson, it is an iterative algorithm, stepwise augmenting flow in different paths from an initial network with no flow at all (thus admitting more) until the limits of the maximal capacities are attained. Each iteration keeps on pushing additional flow with respect to the integrity constraints for flow networks up to the point when the network is finally saturated.

Pseudo-code of the Edmonds-Karp algorithm [52][8]:

```
EdmondsKarp(N, s, t)

//init

foreach(Edge e: N.getEdges())
{
    f(u, v) = 0;
    f(v, u) = 0;
}

//iteration

while(exists a shortest path p(s, t) in residual network)
{
    cResidual = getMinCResidual(p.getEdges())

    foreach(Edge e(u, v): p)
    {
        //augmenting flow
```

[8] $cResidual$ is the amount of flow which can still be added to an edge, being the difference $c(e) - f(e)$.

```
        f(u, v) += cResidual;
        f(v, u) = -f(u, v);
    }
}
```

In [52], also a proof can be found for verifying that the overall running time of this algorithm amounts to $O(|V| \times |E|^2)$.

4 Enhanced Concepts Enabling Geo-Spatial Reasoning

This chapter is devoted to demonstrating that graphs (see chapter 3) - with some special extensions - are a viable mean for capturing the domain of transportation and routing (examined in chapter 2) into a formal model. By introducing important semantic concepts appreciating the complex subtleties of these networks, it is shown how this modelling is performed in practice.

Our group of Ph.D., master and project students is collectively contributing to this system. Its very fundamental components, namely the data model and inner representation, will be implemented in the **research framework** for routing purposes (see chapter 5, in which the **graph structure** is the main paradigm. This thesis is focused on a conceptualisation and implementation of an adequate graph structure, enabling further research in the field of reasoning, as well as in graph transformations, which are covered in the *Ph.D. thesis* of *Laibing Yang*. Using such a formal instrument, operations like graph transformations can be done more easily. Visualisation will be the main issue for the *project thesis* of *Martin Wassermann*. In this chapter, important **semantic concepts** being the pillars of the framework are considered in detail.

4.1 Geographic Information Systems and Internet

For a better understanding of the concerning problems when making up an information system with a geographical imprinting[1], one can profit from the experience gained in the field of *Ggeographic Information Systems*, which have passed the test of practical application [43] and therefore are introduced in the following. All presented systems in section 2.3.3 have one common principle upon which they are based: the representation of transportation networks which are all situated on the **surface of the Earth**.

Seemingly trivial at first glimpse, however, this statement is of crucial importance, for nowadays,

[1]Since all transport networks have a spatial extent, they can be subsumed here, too

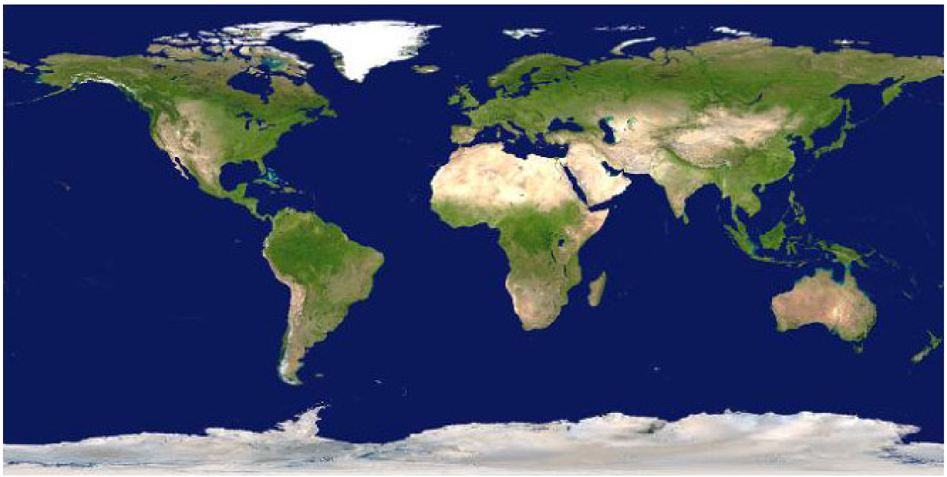


Figure 4.1: A Satellite Image of the Earth (from [23])

geographical data is relevant in many respects. In recent years, digitalisation of geo-data has been done in a great scale, still showing a growing tendency. Above, there are even some tree-dimensional simulations using techniques of virtual reality for displaying maps of cites or buildings [4].

Indeed, geographical data has a highly visible economic impact. According to Dickmann [54], an estimated 80% of information has spatial extent. For now and the near future, the evaluation of thematic attributes will be of special interest for companies. Geo-data is the emerging economic factor for the future.

4.1.1 GIS

Basically, geo-information are any pieces of information which have a spatial extent on the surface of the earth. Specialisation in terms of storage, processing and retrieval has taken place for this sort of information: Particularly in the context of database technologies, **Geographic Information Systems** (**GIS**) have evolved for a more appropriate description of real objects on our planet.

Definition:

Geographic Information Systems (**GIS**)

are "computer-assisted information systems for capture, storage, retrieval, processing, analysis and display of spatial data" which describe a "part of the surface of the Earth or any technical and administrative institutions on it" [74, 43].

The most striking distinction from ordinary database systems clearly is that data always has a spatial reference (surface of the Earth). As for distinguishing from computer-assisted cartography, GIS not only provide mechanisms for mere representation and visualisation of available data, but also for deduction of new knowledge by complex processing and analysis steps. Nevertheless, both representation and processing are naturally important aspects of GIS.

Geographic data is being displayed on two- or (nowadays often) tree-dimensional maps. Historically, the main focus shifted from mere image processing towards semantic integration of spatial data enabling a multitude of operations on it [46, 76].

As more research was done in this area, the shortcomings of traditional relational database schemata became apparent. For this domain, a special representation was needed, which could make a distinction between physical and conceptual attributes of an object (properties in terms of the *Open GIS Consortium* specifications [12, 11, 67]): Geo-objects (buildings, rivers, mountains, streets etc.) can be classified by their **spatial attributes** on one hand an their **thematic** (non-spacial) **attributes** on the other.

The former is described by the geographical space of a geo-object, which can be split into **geometrical** (e.g. coordinates, Euclidean distance, angles etc.) and **topological** aspects (which are adequately represented by graphs). In a formal manner, relations like the binary $geo \times geo \mapsto bool$ adjacency or $geo \times geo \mapsto real$ (e.g. sum of areas) or geo^n can be defined, which realise geometrical and topological operations for geo-objects (see section 4.2).

Coordinate reference systems are being employed for the appropriate description of positions (absolute spatial reference) [43] so that any objects, regardless if fixed or mobile, can be located and tracked. As the Earth is well approximated by a rotating ellipsoid, circles known as medians (360 in total) and parallels (180 in total) partition it into different longitudes (East/West) and latitudes (North/-South) respectively. Its surface can be mathematically described by a mapping of an ellipsoid's surface onto a plane, flattening this sphere surface to the Euclidean R^2. This technique is especially used in cartography. *UTM* (*Universal Transverse Mercator*) and *Gauß-Krüger*, which are the most common coordinate reference systems, both describe geo-location by projecting the geo ellipsoid onto a plane.

A keenly classification for geo-objects is required for their semantic processing. This way invariably leads to the field of ontologies for knowledge representation (see section 5.3.2). Nowadays, GIS have become an indispensable handwork for ordinary geographers, offering plenty of utility and still being

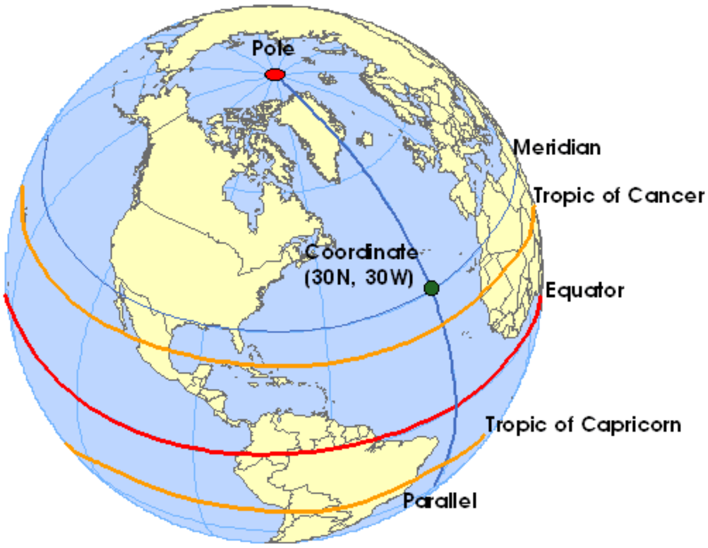


Figure 4.2: Coordinates on the Earth Represented by Latitudes and Longitudes (from [25])

under progress. But it is not only geographers who avail themselves of these potentials, as can be seen by the integration of applications into these systems, most prominently for tourism, traffic and mobile devices. Especially **location based services** are currently gaining increasing attention, most importantly in the sector of mobile radio telephone service for which knowledge of current location opens the gate for a multitude of applications making use of this geographic information. Another field of application certainly is traffic, as the presence of electronic systems in cars, so-called embedded systems for navigation and other aid, are getting indispensable.

4.1.2 Vector vs. Raster Representation

Invisible to the general user communicating by a graphical front-end with the system, the internal representation of data is an issue of interest particularly for the efficient and correct processing of queries. But also for maintainability and adaptivity, like for any proper piece of software, it takes a thorough data model 'under the hood'. In literature on GIS [74, 43], two fundamentally different approaches for representing data are pointed out which are prevailing in the system landscape of today:

1. **Vector** *or topological representation.*

In this kind of representation, **points** are the basic building units, which correspond to nodes/vertices in graphs. Decorated with coordinates, points can be used to derive geometric attributes (length, distance, surface, angle). Notably, same position is space does not mean that two geo-objects are equivalent, i.e. two different points within the same position can exist independently.

Line segments (also called vectors), which have a direction, are composed of two distinct points with different positions. Usually, $+$ and $-$ is used for denoting the orientation of vectors (in terms of positive and negative incidence, see section 3.4.1). Hence, directed edges (arcs) are a natural way of representing line segments. Not only can geometries of objects be defined as line segments, but also they can be represented by curves instead.

In turn, an arrangement of chained vectors composes a **polygon** (ideally suited for wire frames of three-dimensional objects), which can be either convex or concave, and for which the analogue in graph representation is a cyclic graph. Furthermore, adjacency and its transitive hull, represented by the topological reachability relation (all vertices of the same connected component), can be ideally defined with line segments and polygons.

According to [43], the key advantages of vector representation are low storage space, simple co-ordinate transformations but also simple realisation of distance operations (e.g. Euclidean distance metric). On the other hand, computing neighbourhood and intersection are rather difficult. Dickmann [54] offers examples of real GIS which use vector representation.

2. **Raster** *or grid representation.*

The second paradigm for representing geo-objects is relying on cellular organisation, subdividing space into partitions by a rectangular, orthogonal grid, along x and y coordinate axes, called raster. Basically composed of cells, the grid interpolates measure points of an area. To each cell, a set of thematic attributes can be assigned, yet it has no explicit geometry.

In this context, the basic data structure (originating from image processing) is a **coverage** (see also [12]). Being an area-like structure covering homogeneous areas with same thematic meaning, e.g. weather data as thematic attributes (coverage for storm clouds, isobaric areas etc.), a coverage can contain several cells of a raster. It is not surprising that satellite pictures are naturally represented by raster data.

The shortcomings of this representation are, according to [43], large need for storage, difficulties in rotating and transforming coordinates, not to mention complex object generation.

Vector and raster representations are complementary, since one's advantages are the other's disadvantages and vice versa. Notably, the conversion from raster to vector representation is very costly.

Research in GIS technologies has additionally yielded special, sophisticated indexing structures for efficient processing of spatial queries including *quad trees*, *kd-trees*, *R+-trees* or *cell trees* [74]. Hence, for indexing points in multi-dimensional space, there are several techniques:

Quad trees offer a hybrid approach integrating both paradigms. Principally, a quad tree is divided into the four cells NW (North-West), NE (North-East), SW (South-West) and SE (South-East), which may contain further recursive subdivisions themselves. For the purpose of indexing, all points of a quad tree are assigned to the respective cell according to their coordinates.

Bounding boxes (sometimes also called minimal bounding rectangles: MBR) are a good first approximation for geo-objects, when executing queries. They can be seen as a first filter, for which false hits are possible. So as a second step, the exact representation of an object needs to be considered in order to determine whether the query does indeed match or not.

4.1.3 Map Servers and Web-GIS

When Internet had its breakthrough in the nineties, it also yielded fruit in the domain of GIS and mapping. Being a medium for spreading information to a great number of people and for communicating all around the globe, it is ideally suited for making cartographic and geographic data publicly accessible. Further advantages of this net of reticulating knowledge are feasible data exchange and communication, and the timeliness of data, as 'old' web pages are afflicted with diminishing interest [54].

In recent years, electronic representations of maps on the World Wide Web appeared as so-called **map servers**. Whereas these electronic maps are concerned with visualisation in the first place, the scope of web-GIS is primarily on the data representation and querying [54], so rather being a globally accessible data source providing information for further analysis.

Still, the amount of geo-data which is available on-line is constantly growing. Contrary to the previous static view-only maps, modern *dynamic maps* are characterised by an increasingly interactive GIS functionality. These **sensitive** or **click-able maps**, which are offering features like click-sensitive points (countries, cities etc.), react to user input accordingly, by showing more detailed information on the entity selected or by zooming in. So each user can influence the information shown by actively telling the system the desired entities. Changing perspective by zooming in and out of certain regions has become a common yet indispensable way of manipulating an electronic map.

In addition, **maps on demand** have ascended, where information layers can added and removed dynamically (on-the-fly), like many contemporary systems offer [54]. They provide personalisation in terms of selectable information layers (e.g. weather data, population of cities etc.) which every individual user can either select or not depending on his individual interests.

Consequently, the division line between web mapping and web-GIS is becoming more obscure. Both zooming into different levels of representation (scale) and overlaying new strata of information (hence this technique is called map overlay) are map based on one hand, yet take up a correct data modelling on the other hand. Web-GIS do have an extended search functionality compared to pure map servers, since they can benefit from their more complex data model. Thus, web-maps are merely serving the purpose of being the presentation layer of geo-data in a complete web-GIS system architecture, i.e. *view* and *controller* in the classical *model-view-controller* (*MVC*) architecture, whereas the core of a GIS is certainly the *model*).

It is not only geographers who can profit by on-line geo-information services, but increasingly also other special applications, for instance dealing with tourism or traffic. Enriched with thematic aspects of the respective domain, systems can offer **touristic information**, e.g. **points of interest** for sightseeing in cities or regions, restaurants, petrol stations etc., but also weather forecast, warning for earthquakes and storm, and information about the climate. Choosing the right accommodations and finally **reservations of hotels** are also made possible.

Since transportation networks as an infrastructure are a substantial part of the application domain considered in this work, the specialised subfield of geography introduced by *Transport Geography on the Web* [25] is of special relevance:

"Transport geography is a sub-discipline of geography concerned about movements of freight, people and information. It seeks to link spatial constraints and attributes with the origin, the destination, the extent, the nature and the purpose of movements."

Last but not least, **prearrangements for travel**, involving routing with **current traffic flow status** and announcements of stoppages and traffic jams, but also on-board navigation (e.g. navigation systems for cars) when travelling are worth being mentioned. One clearly perceives the large possibilities of geo-information, of whom only a small part is put into reality today. For the future, these special applications will possibly gain even more significance if economic interest persists. As the value attached to geo information is appreciated, it will continue to find its way into enterprises, as so-called *geo-business*.

As the emerging data representation technology, developed by the *World Wide Web Consortium* (W3C) for standardisation of data, the generic markup language *XML* [29] can be tailored by many applications, among whom web-based GIS (denominated web-GIS) can be counted.

The *Open GIS Consortium* has also set off an initiative for the free exchange of geographical data preliminarily encoded in different proprietary formats. [67, 12] Accordingly, its main effort is to integrate heterogeneous source data into a common format: the *Geography Markup Language* (GML) [11], an XML derivate (see also 5.2.2 for practical application). By additionally offering web services for geographical data, the *Open GIS Consortium* is fostering distributed geo databases.

4.1.4 Legal Conditions

When acquiring geo-spatial data, one is not able to do so indiscriminately, yet has to respect the legal conditions regulating the rights of the parties affected.

Especially in terms of **intellectual property rights**, the acquisition and survey of geographical data may be limited or even prohibited by law. Otherwise, when not obeying these legal restrictions, one may get into problems by infringing copyright:

"[Every piece of information, which] by virtue of selection or arrangement of data .. is a personal intellectual creation [, stands under governmental protection.]" [54] [2] [3]

[2] author's own translation, from German act for regulation of general conditions for information and communications services, article 7 (Gesetz zur Regelung der Rahmenbedingungen für Informations- und Kommunikationsdienste - IuKDG), which is changing the German Copyright Act

[3] translation of original text from the German Copyright Act - UrhG:

§ 4 Sammelwerke und Datenbankwerke

(1) Sammlungen von Werken, Daten oder anderen unabhängigen Elementen, die *aufgrund der Auswahl oder Anordnung der Elemente eine persönliche geistige Schöpfung sind* (Sammelwerke), werden, unbeschadet eines an den einzelnen Elementen gegebenenfalls bestehenden Urheberrechts oder verwandten Schutzrechts, wie selbständige Werke geschützt.

For utilisation of official topographic maps in Germany, in addition, one has to respect laws for land survey. Recapitulating, further use of official geo-information is rigidly regulated and has to be respected.

4.2 Spatial Reasoning with Topology and Geometry

Originating from the Greek words 'topos' and 'logos', topology literally denotes the study of places.

Substantially, topology is a "general notion of space" [43], thus specifying a spatial structure abstractly (see also [77]).

Topological spaces in general are point sets which fulfill special conditions (for the mathematical definitions see [77]). Commonplace examples of topological spaces include the Euclidean space R^2 or R^3, especially for maps.

As a mathematical sub-discipline, topology regards space in an **abstract**, **relative** manner, unlike geometry, where exact (absolute) coordinates, metrics and angles are the essence.

The two dimensions of maps can be reduced to some important topological relations, which are sufficient to describe the relationship of one spatial object relative to another one [77, 43] on a higher level of abstraction. Beyond, topology is invariant to homeomorphisms (rotation and translation), accordingly describing the relative position of two geo-objects to each other, which is an advantage gained from its generality. Notably, space can be subdivided into geometry and topology, depending on the level of generalisation. Abstracting from precise positions (coordinates) and metric (distance), topological relations deliver a "relative spatial reference" [43].

It is topological abstraction that comes next to human understanding of relative positions of objects in space (e.g. whether an object is above another, or next to it).

Indeed, graph theory and topology have a lot in common, since most relevant relationships, which can be described by means of graphs (adjacency / neighbourhood and connection) are specific **topological relations**.

Neighbourhood This relation in graph theory basically determines all adjacent nodes from a given node being connected to it. Again, its transitive hull can be used to define a $reachable$ relation grouping all connected components of a graph into equivalence classes.

Interior / Hull / Exterior Until now, this notion can merely be applied to checking whether a vertex or edge belongs to a graph (interior then) or not. Beyond, regions which may include elements will be of interest. Their link to graph theory will be set up by applying a hierarchy to graphs (see section 4.4): nodes themselves on a higher level may recursively contain graphs on a lower hierarchy level.

Above / Under Usually, within transportation networks, two dimensions are sufficient for locating entities. Yet for modelling underpasses and overpasses, a third z-coordinate has to be introduced.

Nodes and edges are the basic units not only in graphs, but also in topological spaces (for a formal definition see [77]) characterising spatial constitution with 0- ore 1-dimensional cells or "simplical complexes" respectively. [43] For example, the three different yet isomorphic representations of K_4 depicted in section 3.3.1 are topologically equivalent in terms of connectivity since the underlying graph is the same.

Further topological relations constitute intersection and overlapping of two objects. Being on an abstract level, topological relations are good for the description of routing, especially for humans, since advices like taking a *left turn* at the *next* junction or taking the bridge *above* the river by nature convey the ideas of directions, but also topology.

[74] points out the language $GEOQL$ as an extension of SQL being predominant in the field of databases for querying spatial data in particular, which is performed in the data model by several spatial operations, among whom there are to mention $intersects$, $adjacent$, $joins$, $ends\ at$, $contains$, $situated\ at$, $within$, $closest$ and $furthest$. On an implementation level, touching points with computational geometry become apparent, since algorithms from this area can be applied for efficiently performing these basic geo-operations.

In contrast to these topological spatial relations, **geometrical** ones comprising orientation and distance are all depending on a certain metric, thus can be measured not only qualitatively, but also quantitatively in an appropriate unit of measure:

Orientation The orientation of a shape or network is describing its position with respect to fixed directions like North, East, South and West (the corresponding binary directional relations are $north\text{-}west\ of$, $south\text{-}east\ of$ etc.). Since by rotating an entity, its orientation is changed, this spatial concept is not topological but absolute. Furthermore, angles between parts of a shape can be determined.

Distance Clearly, as distance can be measured, it is metric and thus absolute. Different approaches for this purpose exist, most prominently the Euclidean metric, but also other ones like the Manhattan metric.

In mathematics, a **metrical space** d (for distance) is defined as a function $d : P \times P \mapsto \mathbf{R}_0^+$ over a point set P which fulfills the following axioms:

Identity Axiom $d(P,Q) = 0 \Leftrightarrow P = Q$. This axiom is not necessarily valid for all transport networks, because two vertices may coincidently have the same coordinates, but need not be equivalent by any means.

Symmetry Axiom $d(P,Q) = d(Q,P)$. Neither is this axiom is generally valid for transportation networks, since there exist anomalies for two lanes, for instance as it is the case for one-way streets, or street segments which describe a curve (one lane is on the inner side of the curve whereas the other is on the outer side).

Triangular Inequation $d(P,R) \leq d(P,Q) + d(Q,R)$.

The Euclidean space is a special metric space defined as $d(P,Q) = |P - Q| = \sqrt{\sum_{i=1}^{2}(P_i - Q_i)^2}$. As another example, the Manhattan space originating from the street system of New York named after the quarter of 'Big Apple' is defined by $d(P,Q) = P + Q = \sum_{i=1}^{2} |P_i - Q_i|$. In chapter 5.3, the implementation of these two metrics are highlighted among others.

Both [86] and [60] have shown that geometrical concepts do have benefits in speeding up routing computations by heuristics.

Altogether, topology provides abstract means to describe space and accordingly, and thus can be well used for **spatial reasoning** [77], a subfield of artificial intelligence. A multitude of different formal models pertaining to spatial reasoning exist:

The Region Connection Calculus RCC-8 [51, 58] defined on eight different topological relations is able to express most importantly inclusions for regions as well as neighbourhood.

A Modal Logic with spatial interpretation of the modal sentential connectives is introduced by [39], being a computationally good compromise (like the propositional temporal logic) between the expressive yet semi-decidable first order logic and the decidable yet less expressive propositional logic.

Regular Path Expressions , which build up a formal language defined by [36], express combinations of changeovers. An approach can be adopted in which reasoning on these aspects significantly influences the process of determining a shortest path. These modal chains and their most common patterns will be more deeply covered in section 4.5.

Topological relations go along well with graphs and are well-suited for spatial reasoning since yielding only yes/no digital answers, they can easily be expressed in a formal logic. Over and above, these means can also be employed for special heuristics to improve the average case performance of routing algorithms.

For graphs, topology can be described by some general patterns (depicted in figure 4.3):

Linear . The simplest of all forms, e.g. a bus route.

Mesh . For instance a metropolitan street network.

Tree . For instance a strictly hierarchic structure.

Hub-and-Spoke : airports often serve as hubs.

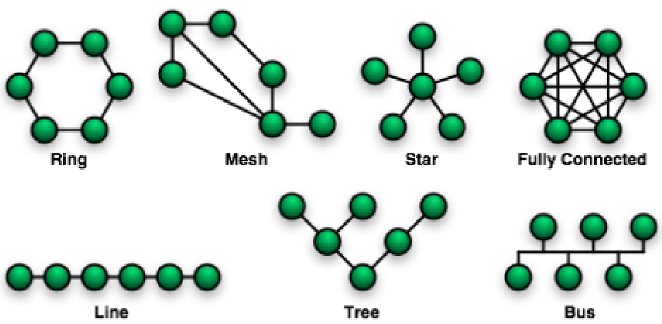


Figure 4.3: Different Kinds of Network Topologies (from [32])

Manhattan and radial graphs are also mentioned as special types of topologies in literature [48]. Often, graphs representing street networks are rather irregular due to parks, airports and other large areas.

4.3 Graphs for Modelling Networks - Practical Considerations

In this section, the focus is upon the benefits obtained by using the formal characterisation of the specific real-world entities within transport networks by graph theory.

As remarked before in chapter 2, transportation networks are an infrastructure upon which vehicles can move. They can also be regarded from a high level of abstraction as networks in terms of graph theory. That is to say, one can account streets or any other types of connections as edges between different nodes, regardless of which concrete meaning is attributed to them. Graph theory with its great expressiveness can capture all kinds of traffic networks, being the commonality of all.

The general benefits of modelling with graph theory consist of ..

- being a universal format for describing many different domains.
- capturing domain entities in a formal manner, being precise and unambiguous.
- being both **intuitive** (ease of visualisation) and **formal** (mathematical foundation) in the same instant.
- having applications for many different domains, all contributing to its further progress, graph theory certainly is **interdisciplinary**. Various applications can profit from each other on similar problems, enriching their own solutions with ideas of other researchers.
- being largely independent of other branches of mathematics.
- resenting a major boost during the last years according to [50, 49]. The research which was done has yielded into a great arsenal of algorithms to pick from, which can be tailored for a concrete application individually.

What are the particular benefits and drawbacks of describing aspects pertaining to the domain of transportation by means of graphs?

Reflecting the physical structure of real transportation networks, which can substantially be reduced to be consisting of points and connections between them, graphs are an ideal mean for modelling these sorts of infrastructure in general. [50] Also, as [49] aptly points out, graphs are well-suited to ”conceptualise the problems faced in the everyday world”, and in particular, its application areas include flows, transportation and traffic (airlines and railroads for example).

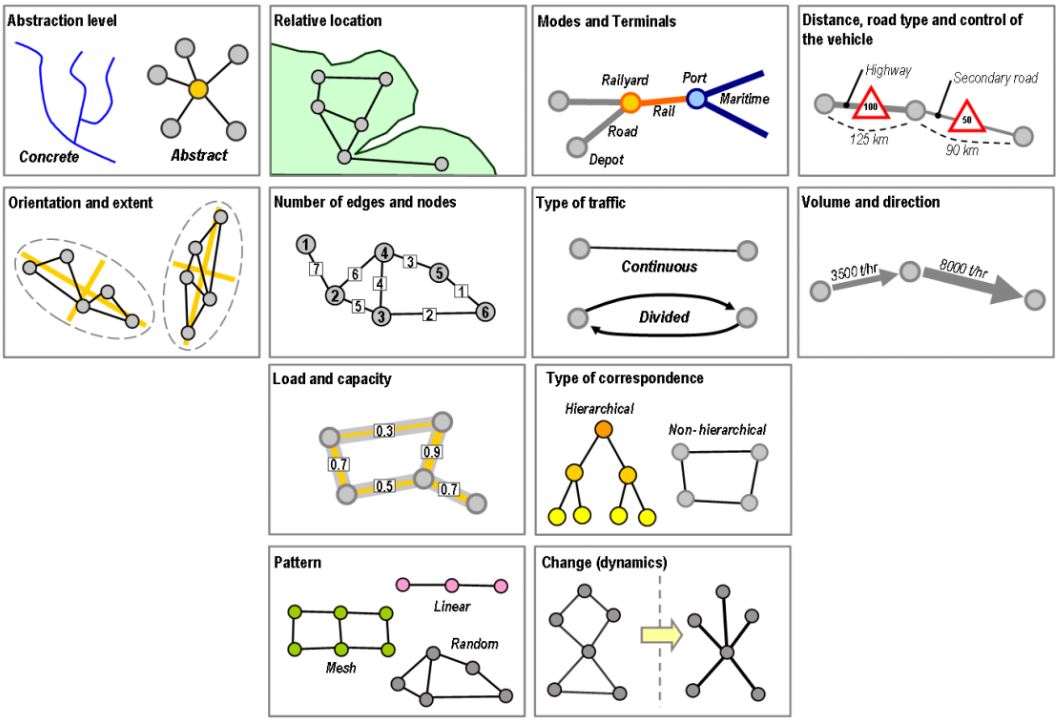


Figure 4.4: A Multitude of Aspects to be Covered by the Modelling (from [25])

The dynamic aspects within these networks, namely traffic flow from a source to a target location, can be captured, in terms of graph theory, by the **traversal of a path** between these two nodes, or by a **flow through a network** having edge **capacities**. Especially, **directed and weighted graphs** best meet these requirements:

As there are many one-way streets in cities, directed graphs are needed for modelling. But also edge weights indicating travel times or distances can subsequently be used, when asking for a shortest (fastest, cheapest,..) connection between two nodes in the network. This way, one can avail oneself of manifold graph algorithms like the one contributed by **Dijkstra** for shortest path computations, or the **Edmonds-Karp algorithm** for solving the maximum flow problem (see section 3.7.6).

Additionally, graph search in general can be exploited for many queries, especially in applications concerning **location-based services**. All these algorithmic tasks can be abstracted to user queries, namely the searching of graphs with paths, edges, or nodes as result types. Giving valuable information to users on demand (triggered by the users themselves) is the gist of this query execution implemented as graph search.

In contrast, some functions should be sparked off by the system itself, contributing to its dynamic behaviour. Important notifications like formation and breaking up of traffic jams, caused by accidents or due to construction may well be treated this way. Fitting into this second category are also all other sorts of delays, e.g. late take-offs or departures deflecting from the usual schedule. In the recent winter term 2004-05, the research unit for programming and modelling languages at the Ludwig-Maximilian's University of Munich has developed a system capable of picking up and decoding *Traffic Message Channel* (TMC) [24] messages transmitted by radio waves, enveloping them into XML data for further exchange [14]. TMC is a standard commonly employed by modern navigation systems. Medium-term, the idea of tying in with the TMC system sounds appealing, for research can be enhanced in a sensible way.

However, graphs are not the sole possibility for describing this part of reality. [75] introduce optimal control theory as an alternative to graph theory, defining constraints as a collection of equations to be solved. Flow constraints expressing conservation and propagation are considered. Moreover, interesting dynamic aspects like traffic jams, first-in first-out ($FIFO$) queues and stop-and-go waves with overlays are presented in detail. In graphs, especially flow networks, dynamics can be modelled by a macroscopic simulation [75, 73, 47] using flows along edges respecting capacity constraints which may change over time.

Beyond, considering some special contexts, one has to distinguish between a physical infrastructure (e.g. rail tracks) and a logical infrastructure (e.g. the labelling of a certain underground line as 'U1'). Notably, a physical structure can encompass several logical ones operating on it, for instance the underground lines 'U3' and 'U6' both sharing the very same rail track at different times. Corresponding to these logical lines, timetables play an important role in the field of public transport, since they regulate the sharing of physical infrastructure within time. In general, as they are more often subject to changes than the physical structure itself, a separation between physical networks (e.g. rail tracks) and all logical lines operating on them (e.g. underground $U3$ and $U6$) is highly desirable for modelling in order to be flexible. Furthermore, all aspects described in chapter 2 have to be respected appropriately.

In chapter 5, the practical implementation of all this knowledge into an ontology, namely the *Ontology for Transport Networks* [63], is presented more detailed. Keeping the ontology separate from the graph structure is the principal idea for facing the problems mentioned above. The graph model should offer possibilities for inserting new elements (vertices and edges) representing physical entities into the

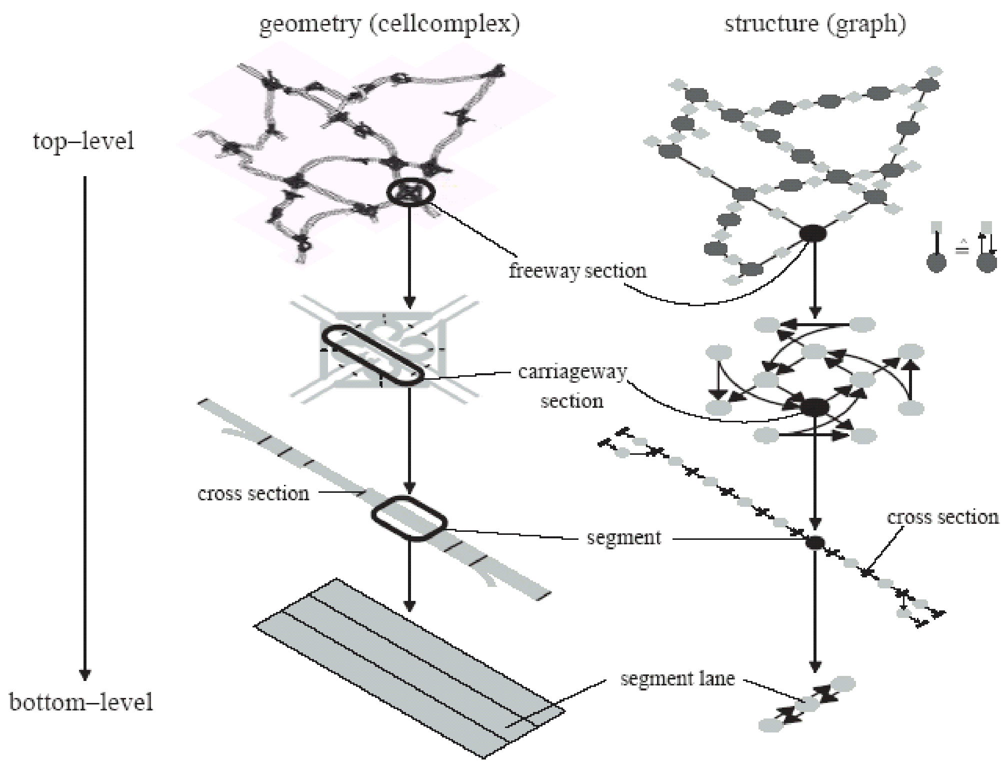


Figure 4.5: A Hierarchical Graph Modelling a Freeway Network (from [80])

network, and also for removal, of course.

Even Dijkstra's algorithms is not useful in practice for these applications, as a broader, more flexible approach must be taken. Depending on one fixed weight only is too restrictive a constraint. A solution to this may be using **attributed graphs** decorated with more attributes, which offer additional flexibility as an adjustable true cost function depending on more than one parameter can be passed over to the shortest path algorithm as input argument.

It are the semantic concepts introduced here which further enrich the simple graphs in expressiveness, making them of use for real applications in the field of transportation. Whereas simple aspects can be well covered by attributes, the more complex ones, however still require further concepts for thorough semantics, for instance the uniform modelling of **hierarchic** networks:

Within complex entities like cities or buildings, an inner structure reveals itself when entering them. From the more general viewpoint of connections between cities, for instance, their interior road networks are irrelevant. There are cases in which the concepts of generalisation (e.g. a street network belonging to a city) are essential for keeping large structures intuitive. Being easier to understand for humans primarily underlines the utility of this hierarchic concept, thus making it a useful mean for reasoning (advanced reasoning techniques could be made possible, consequently facilitating the efforts of applying a logical calculus on space), as additional information is given:

Different scales of objects exist in complex structures, depending on the point of view. In section 4.4, a hierarchical concept will be presented which is capable of describing such nested structures correctly.

4.4 Hierarchic Graphs

4.4.1 Motivation

Throughout literature [48, 56, 80, 44, 81, 45], spatial reasoning avails itself of concepts of a hierarchy with several **levels of abstraction**. Different rules to use for routing in form of heuristics are presented by [48], significantly speeding up computations.

Often denoted as hierarchical graphs [79, 44, 81], these types of graphs which are best able to represent the nested structure of a complex network are first to mention when extending concepts for modelling transportation with means of graphs. Due to the geographical nature of data involved, this hierarchy manifests itself in different levels of abstraction. Indeed, this important concept of levels and the action of scaling between them is employed in GIS (see section 4.1).

Busatto [45], most notably, has led some important research in the field of hierarchic graphs. Some of these important ideas and results are concentrated in this section.

Graphs of a particular level being condensed to one node of the level above, this notion can also be seen in terms of **clustering**, i.e. compressing a graph to one node is a natural grouping into a logical (or sometimes physical) cluster. Since whenever a great amount of nodes is encountered, humans typically form groups of nodes with same characteristics for a better understanding, this approach is a natural one coming close to human cognition [48]. This also facilitates visualisation [56], which otherwise would become indistinct for humans due to its inhering complexity.

Examples of such clusters are:

Cities , which can be either seen as a node or a complete network containing junctions and connections.

Buildings with an inner structure represented as a graph (shopping malls - depicted in figure 4.8, airports or train stations)

Intersections and their interior, since one can consider each lane on its own, as well as each possibility of turning

Vehicles like trains, which can be described by a graph structure for their passengers traversing the interior (e.g. each compartment as a node). However, the focus of this thesis being upon a macroscopic view primarily, this particular notion of also considering vehicles as (hierarchic) graphs could be integrated in a future work.

Parallels to other areas of computer science can be drawn, particularly to software engineering, where the notion of **packages** and subsystems stands for hierarchies of **containment** or physical nested file systems [45]. This way, a concise representation by prefix notation similar to those for packages in modelling can be used (the same prefix for two vertices indicating that they belong to the same graph).

When becoming too large (as is the case for the World Wide Web), graphs tend to lose their intuitiveness. Since human beings are solely capable of understanding a small number of concepts at the same time, an additional structure is needed for those large graphs with respect to human cognition. Inspired from the saying 'divide and conquer', the natural approach is to split the complex structure into smaller ones, which can be managed better. **Abstraction** is an important notion to be mentioned in this context, intuitively applied by humans hundreds of times. Meaningful **grouping** of nodes and edges, which are belonging together in some way (e.g. by proximity, hence a cluster based on topology, or by belonging to the same thematic layer e.g. pharmacies), into a group consequently results in creating a coherent layered structure.

Hierarchy: The hierarchical structure itself can be expressed as a graph, i.e. a tree or in more general contexts a directed acyclic graph (DAG).

Grouping: Groups of nodes and edges themselves form a new entity on a higher level (Busatto denotes it 'package'). When it is useful to couple the hierarchy to the graph structure, these groupings can be in turn nodes or edges.

Hierarchically nested graphs are the latter, in which nodes or edges may contain an entire graph,

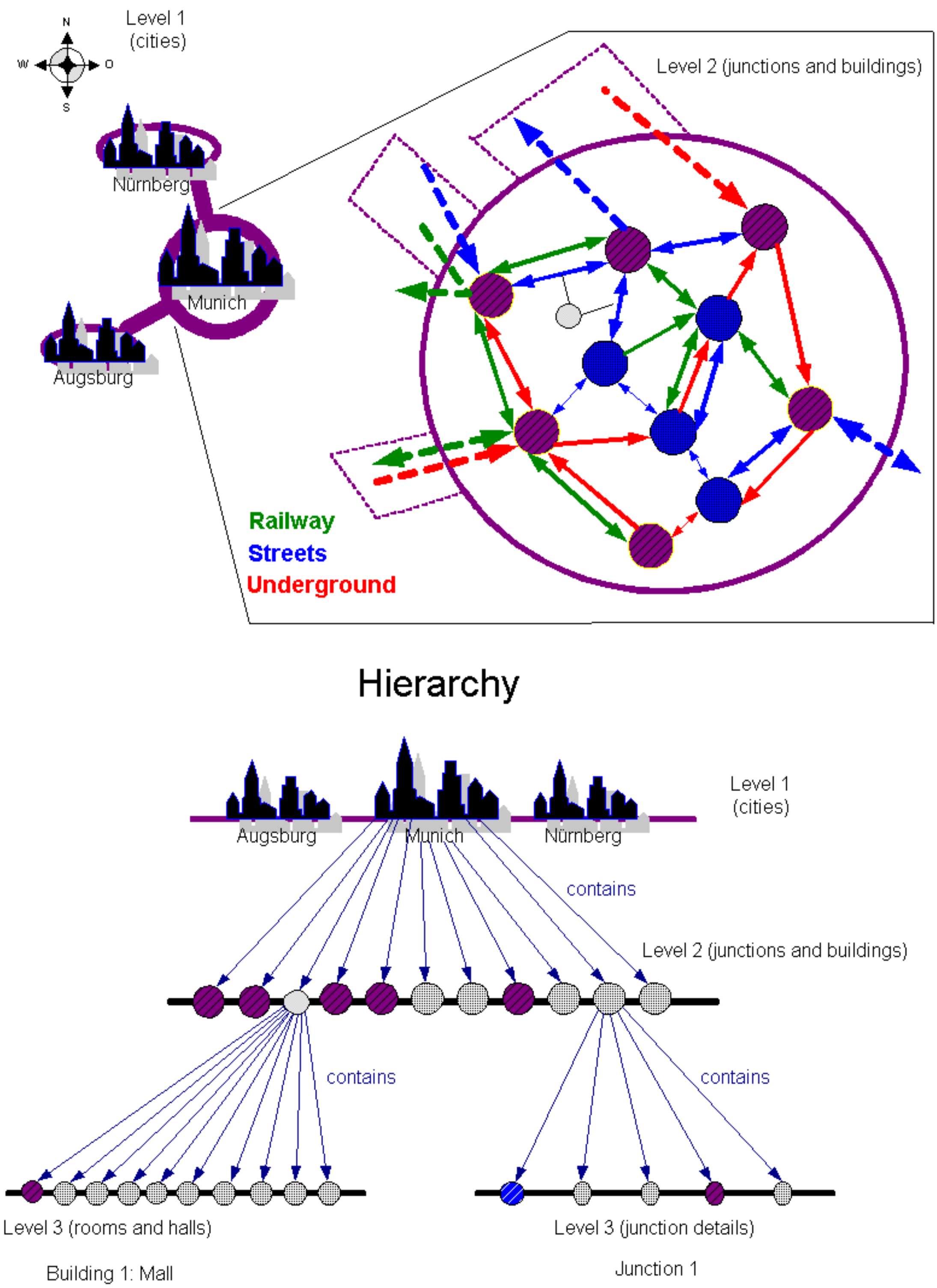


Figure 4.6: Multiple Levels of Abstraction for Modelling Cities

and the abstraction is from an entire graph to a single node, or from a bundle of parallel edges to a single one.

Some ideas of object-oriented programming and data abstraction in general come when thinking of hierarchical containment with encapsulation and information hiding, i.e. that inner elements are kept hidden. The notion of a package in UML also applies a hierarchical grouping on classes and interfaces, however being independent and not coupled to those. The benefit is that hierarchical structuring allows different views of data. Additional constraints restricting certain relationships between entities may also be formulated. In particular, attributed, typed graphs are viable candidates for enforcing constraints e.g. imposed on a certain type or attribute.

For our purposes, as reasoning is the main driving force, the hierarchy will be mainly based on topological grouping, hence a kind of aggregation to clusters (part-of relationship and sub-components). Containment in regions can be well utilised by reasoning techniques like the RCC-8 calculus.

4.4.2 Formal Definition

Busatto [45] points out the significant distinction between a coupled and decoupled approach: The former is characterised by using one primitive to describe distinct pieces of information, e.g. a complex node representing a node in a graph and a grouping of graph items at the same instant. In addition, it includes edges abstracting over bundles of edges, which may well be compact. The decoupled approach, keeping separate information which is separate, does not enforce any particular clustering a priori. By loosely coupling the hierarchy with the graph, one can profit by flexibility. However, every package has to be anchored to its corresponding grouping node or edge, requiring an additional linking mechanism. Linking nodes of different packages together faces another problem. Additional semantics treating these **boundary-crossing edges** have to be defined.

According to [45], the hierarchy with different levels can be considered as a "hierarchy graph" (i.e. a tree or DAG):

A plain hierarchical graph is a triple $H = (G, D, B)$ where G is the **original graph**, $D = (P_D, \succ_D)$ is a **rooted tree** describing the **hierarchy** and B is a **coupling graph**. Elements of P_D are **graph packages**, and $\succ_D$ is defined as **parent relation** between packages. Ancestor and descendant relations as transitive closures over parent and child relations respectively may be

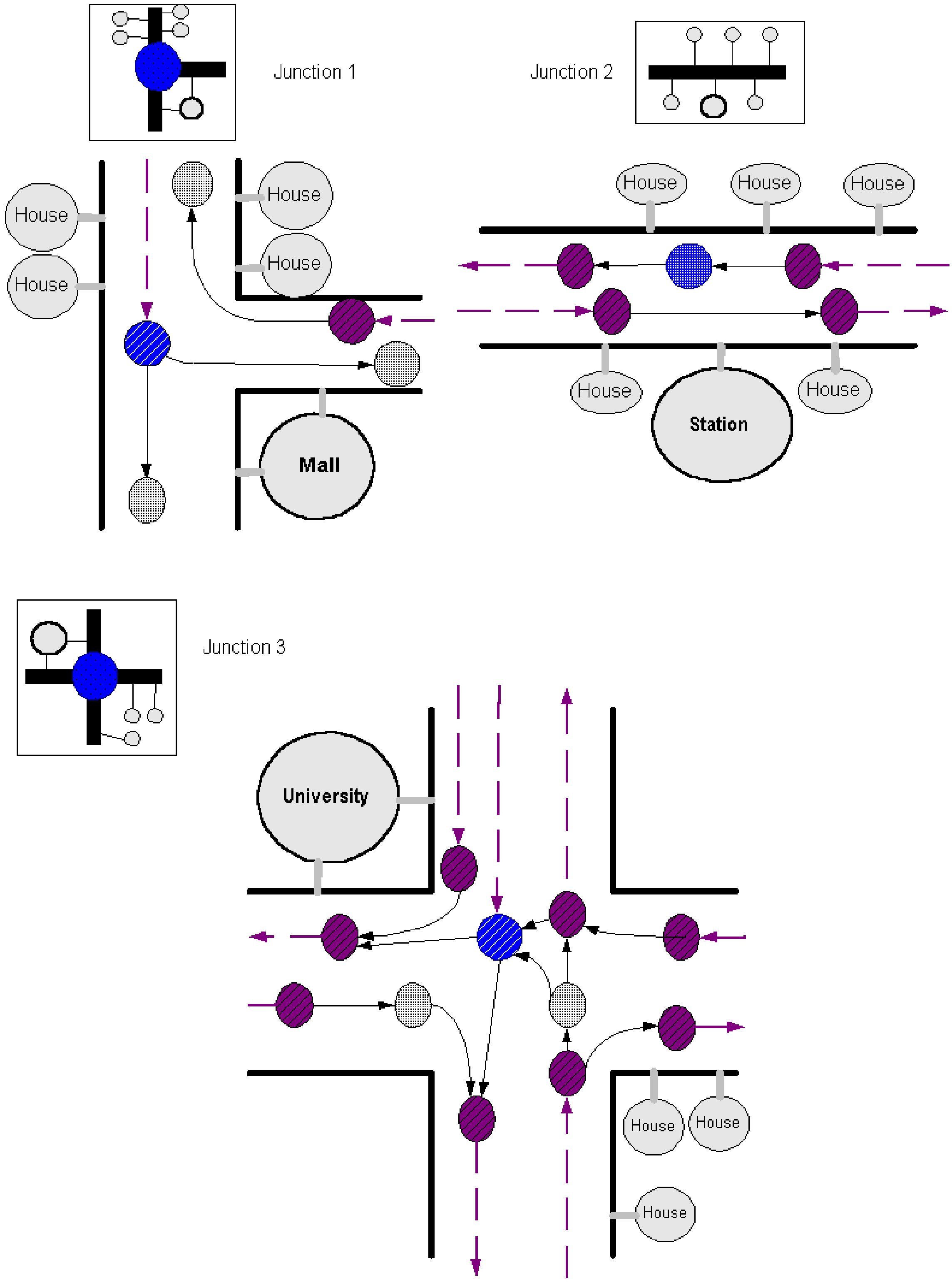


Figure 4.7: Various Intersections (Zoomed) as Hierarchic Graphs

useful as well. Yet another important relation is the sibling relation between two packages on the same level within a common super-package. $B = (P_B, A_B, C_B)$ is the coupling graph, where the packages $P_B = P_D$ and $A_B = V_G$ and C_B is a set of edges connecting the elements of P_B with A_B.

A full hierarchical graph is analogously defined as a triple $H = (G, D, B)$ with the same meaning. The only difference is that A_B is defined by the **union of** V_G **and** E_G. An integrity constraint is posed so that if an edge is contained in a package p, then all its incident nodes have to be contained in that very same package p, too.

A boundary-crossing edge e of a hierarchical graph $H = (G, D, B)$ is an edge $e \in E_G$ with $e = (u, v)$ where u and v are distinct vertices of different (distinct) packages: u is mapped to package p and v is mapped to package q.

The approach taken in this thesis is greatly inspired from these semantic concepts, however, object-oriented techniques are used for realising them. Most significantly, the coupling graph B is implicit by the use object links between different graph elements (see section 5.3.3).

Special modelling concerning buildings is needed for a uniform treatment of this case. Being hierarchic, they form a complex vertex with an inner graph, but also, they can be attached to an edge or several edges (in a bus-like system). For this purpose, the special semantics of an attachment edge are introduced. A street edge is being modelled as incorporating a graph containing the building as a vertex (a complex one though, with an inner graph for its interior). Special attachment edges can consequently be added from vertices within the graph of the street edge to the respective vertex representing the building, being on the same hierarchic level. Addresses and postal code may be interesting attributes for geo-coding buildings, as locations are mostly searched by these attributes.

Considering large buildings like airports, stations, universities or shopping malls, knowing the interior for routing and navigation can be useful, as experience has proven that people often get lost in complex facilities when not being familiar with their interiors.

From a block plan, a dual graph representing the paths from one room or corridor (modelled by nodes) to another through door (modelled by edges) can be made up [49]: This way, buildings neatly integrate into the graph structure as well. Despite of a greater freedom for choosing which point is representing the entire room, and which edge respectively represents a door or walk-through, one can construct a

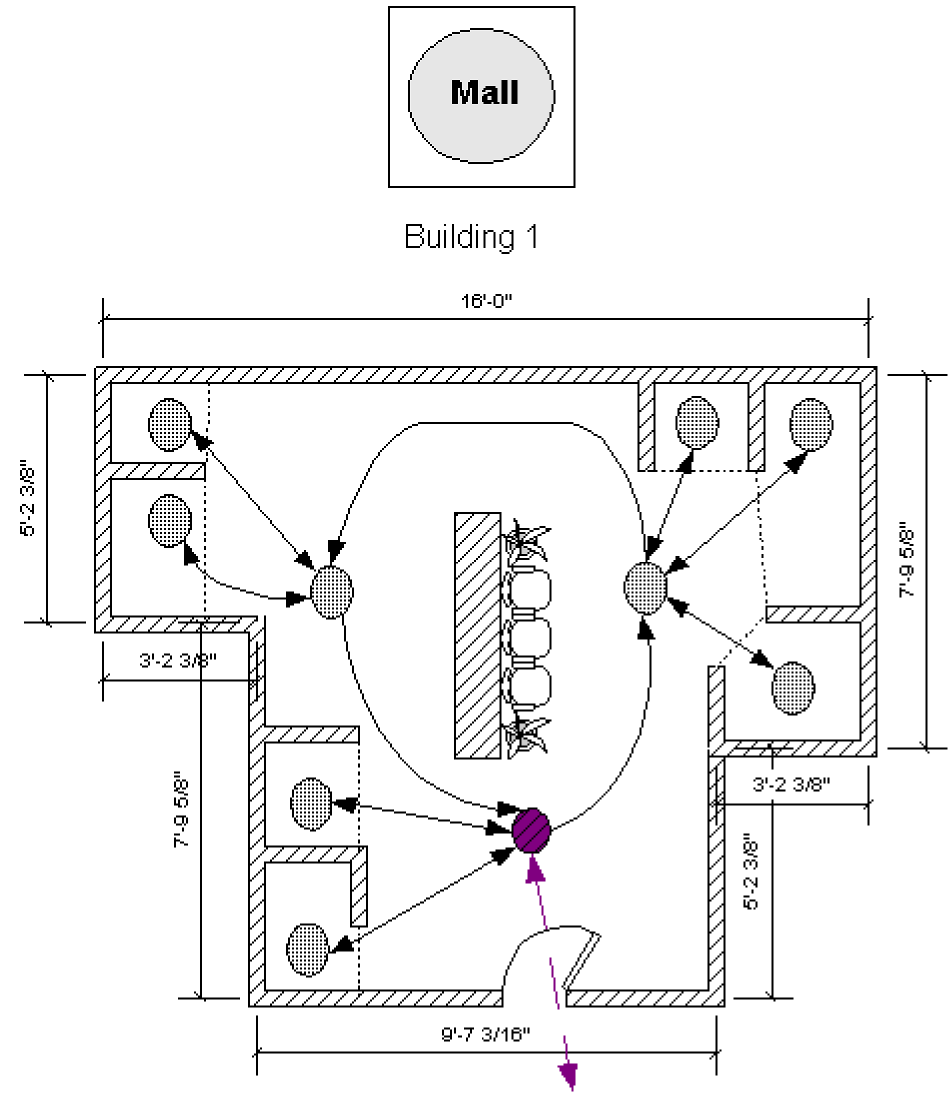


Figure 4.8: Interior of Buildings: Rooms Considered as Vertices, Doors as Edges

reasonable graph representation from the block plan only. Automating such a task on a computer with a given digital block plan as design may well be an interesting task for future work.

For the purpose of display, an abstraction according to certain classification constraints like size of a segment, its number of lanes, maximum capacity or speed limitation or length, into different representation levels is also suited. In more general view, only large segments may be displayed, and as soon as zoomed in, more and more details (smaller segments) become visible. [56, 68]

4.4.3 A Region Hierarchy for Reasoning

As an idea for application of this concept, with respect to the RCC-8 calculus [58], a geographical (and in the same instant political) hierarchy of regions within our planet comprising continents, countries and cities depending on the scale has been elaborated.

It has been realised in the $RegionHierarchy$ class of *TransRoute* (see chapter 5), which contains the tree-like graph in whose vertex representing the Earth situated at the top of the hierarchy naturally is the root. Two kinds of edges can be distinguished, the first ones indicating containment and part-of relationships (TPP and $NTPP$ in RCC-8) whereas the second ones represent neighbourhood relations (EC and PO in RCC-8). Containment of regions is indicated by edges from a higher level to one below (e.g. an edge from the $Europe$ vertex to $Germany$, $France$ and $Malta$ standing for the RCC-8 facts $Europe\ NTPP\ Germany.$ $Europe\ NTPP\ France.$ and $Europe\ TPP\ Malta.$). It may be appealing to automate transformations from declarative rules or facts of RCC-8 to a graph structure representing those. Apart from the edges realising the neighbourhood relation between two continents (also countries or cities), basically the region hierarchy is a tree.

This hierarchic structure will enable future reasoning tasks considering routing in networks of different countries (e.g. Germany and Malta). Even with essential information missing, sometimes it is possible to get to network properties from these abstract relations solely.

For instance, by looking at the region hierarchy, one can inference that Germany and Malta are divided by sea, since Malta is not contained in the transitive hull of the neighbourhood relation (defined as EC in RCC-8) of Germany. In turn, this information implies that in order to travel from Germany to Malta by car, one has to take a ferry. Also, other modes of transportation are not applicable due to this maritime division. Another alternative are flight connections, given that both countries share a

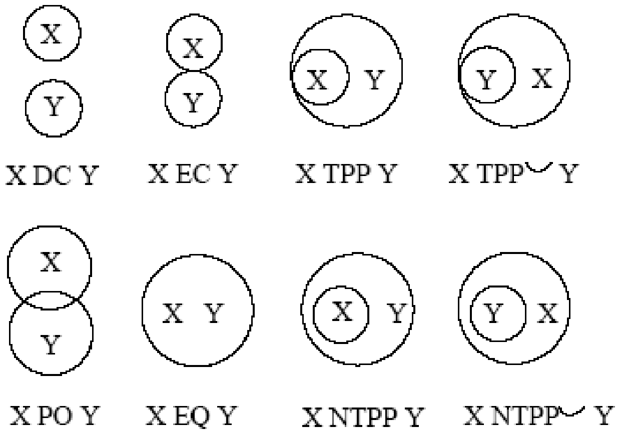


Figure 4.9: Examples Illustrating the Eight Basic Relations of RCC-8 (from [58])

connected air network.

Hence, together with distance, the region hierarchy can be used to confine the choice of transport modes involved in travel between these two countries reasonably. As a first step, this higher-level reasoning solves the problem of greatest scale, which can be refined to the next level then (solving the problem Munich-Valetta, for instance) and so forth, until the shortest path is determined. Further inference rules and reasoning patterns could well be elaborated based on regions and their relations, contributing to an expert system for travelling.

4.5 Multi-Modal Networks and Chains

4.5.1 Multi-Modality

Up to now, issues concerning networks which cross country borders (e.g. a road network between France and Germany), or connections between heterogeneous networks were not considered.

Modelling **multi-modal** networks uniformly is a key issue: Apparently, one of the most fundamental drawbacks of the contemporary systems (see section 2.3.3) for routing is the lack in representation of multi-modal transportation. These systems are mostly specialised in road networks inside of cities and highways, neglecting other transportation modes (even though metro stations might be represented,

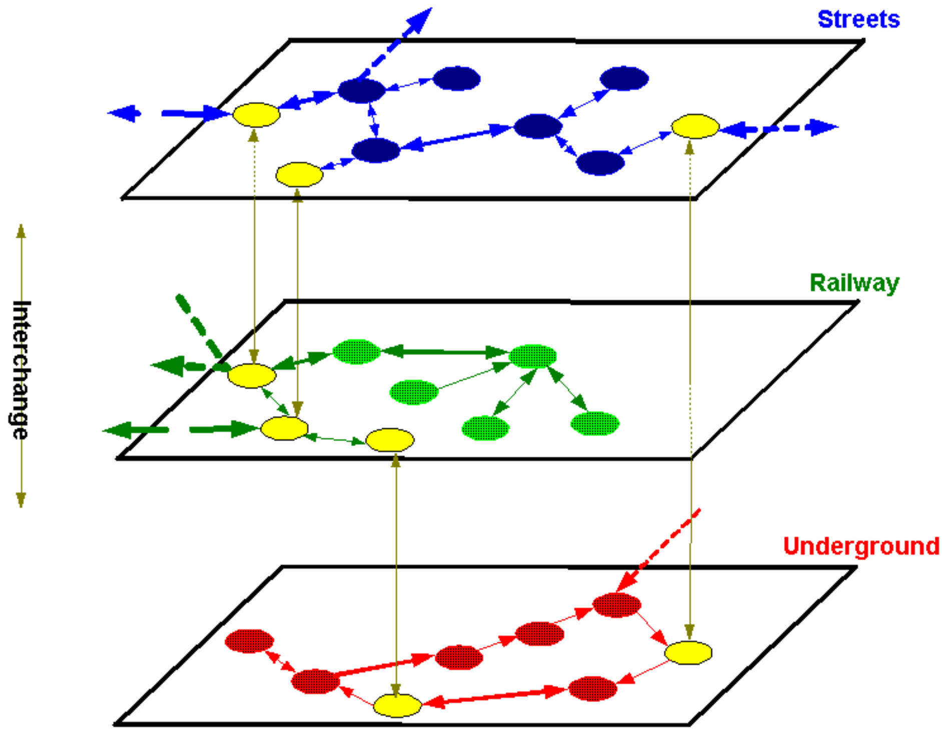


Figure 4.10: Separate Transport Modes as Layers (Shown in Red, Green and Blue)

they are often not taken into account for the routing process). It is individual travelling by car for which these systems are primarily designated. Most often, only road networks or public transit are taken into consideration by these systems.

If a larger journey is planned, routing may not consist of solely finding a path inside one city, but involves finding connecting flights between two cities as well as interior paths in each of these. When boundaries are crossed, there may be alternatives, e.g. a modal choice between flying and travelling by ship or train. Integrating all these different pieces of information together into a uniform representation may ease the task of finding a route under more complicated yet realistic circumstances.

By adopting a high-level viewpoint of transport networks (represented as a vertex on this abstract level), they can not only be matched to regions, but also to network types indicating the modes of transportation which may be used on them.

Unlike in the system developed by [69], the approach taken by this thesis conveys the notion of employing one network for each mode of transportation as a separate graph, which fits well into the hierarchic concept presented in section 4.4.

The basic notion underlying this viewpoint to multi-modality is intuitive (see figure 4.10 [4]): each separate mode of transportation on a plane (sheet), can be overlaid similar to the concept of **map overlay** in GIS (individual layers according to user preferences can be added and removed). Each sheet represents a vertex on a higher level considering different transport networks, but in the same time a graph representing the concrete transport network. For representing transfer connections between vertices in different networks, boundary crossing edges are a viable mean making use of the hierarchic semantics. Consequently, the modelling of a city network is performed by considering each transport network separately as a hierarchic graph and adding boundary crossing edges for modelling interchange of transport mode where necessary.

4.5.2 Regular Expressions for Modal Chains

Having considered modelling of multi-modal networks, now the perspective is changed towards routing, again. Within these multi-modal environments, travellers have a great choice which transport mode to employ.

By regarding the transport modes taken by people more closely, especially their ordered sequence, information about the users themselves are revealed. In particular, [71, 68] propose a *case-based reasoning* approach in which users learn from the experience gained by others categorised in the same user group, thus making route planning more **personalised**.

By setting up a specific user profile, determining travel preferences concerning modes of transportation may be eased. **Patterns of changing transport modes** and the resulting chain (ordered sequence) have been examined by [36], introducing **regular expressions** based on the labels of edges across the desired path in the graph for the representation of these chains. For instance, the regular expression $w^+t^+w^+$ (labels: w = walk, t = train) including two changes between wt and tw (entering and leaving train), abstracts over a path involving the specified chain of transport modes. Based on this formal language, general constraints on shortest paths concerning transport modes and their sequence can be analysed, which may be useful for reasoning.

By combining the two notions of personalisation together with modal chains, modal choices can be also inferred from distance and user preferences. For example, the issue whether private transport is

[4] Note that interchanges (special edges connecting the layers) are only possible at some nodes (marked yellow).

preferred to public transportation or vice versa is relevant within cities.

What do modal chains look like generally? Recognising the most common patterns surely contributes to the development of a sound heuristic. Some basic patterns may be for instance (notation: b = bus, p = plane, r = rail, s = ship, t = taxi, c = car, u = underground) $w^+c^+w^+$ or $w^+b^+u^+w^+$ for within cities, or $w^+b^+w^+\ p\ w^+t^+w^+$ for travelling between different countries. Taking the region hierarchy into account, these chains may show a symmetry, e.g. routing within the origin city may involve $w^+b^+w^+$ to the airport, p for a direct flight connection and $w^+t^+w^+$ from the foreign airport, taking a taxi since the city is unknown, or because it is cheaper in the respective country. In this case, the symmetry center is the flight connection solving the shortest path problem on the highest hierarchy level.

In contrast, sequences like $wtwcwtwcwbwtwcwbwcwtw$ containing plenty of transfers will scarcely be encountered. As rarely as possible will a mode of transportation be changed, due to the amount of time required. Complexity arises when several possibilities exist for each step, leading to a large combinatoric number in the worst case. As a general rule, the modal chain should be kept as simple as possible, i.e. modal change shall only take place when necessary and if an essential time bonus is expected.

5 A Framework for Routing in Transport Networks: *TransRoute*

The objective of this thesis was to set up an information system capable of processing users' routing queries on real transportation networks. As remarked in section 3.1.2, seeking an optimum route is equivalent to the task of finding the path with minimum weight by searching a graph. Due to the nature of routing, i.e. being an optimisation problem, it can be managed well by algorithms of graph theory. The benefits resulting from this formal representation are manifold, as seen in the last chapters.

TransRoute brings together the theoretical concepts of graph theory with the practical application concerning transport networks. Of course, object-oriented techniques form a vital aspect due to the choice of the programming language *Java* [22] for realisation. Some of the possibilities with respect to the new version *5.0*, for instance the use of generic classes and methods, will be employed here. Graphs and their elements can naturally be accessed in an object-oriented fashion.

5.1 System Analysis

5.1.1 Requirements Catalogue

In order to succeed in building up a practical software application, all its requirements have to be respected appropriately. This highly necessary and vital first process is well-known as *requirements analysis*. As the specific domain in question was studied more closely in chapter 2, now the task is to develop a computer model capable of aptly describing all these transportation networks with all their specifics. What kinds of applications shall be realised using the framework?

Of course, routing is the target to be striven for, i.e. finding ways through transportation networks which are represented by graphs (so essentially finding paths in graphs). Consequently, this will be the central application of the framework:

Modelling. The very core of the framework to be constructed will be the modelling of different transportation networks as graphs. Being the central building unit, the **graph structure** should be able to describe as many different real life entities as possible. A uniform approach to various concepts is needed, e.g. the interior of buildings or the change of transportation mode in a multi-modal environment.

Flexibility. On the other hand, the framework should be as **generic** as possible, offering enough general concepts for different applications, which can be adapted fittingly for each individual use. For instance, it should be made possible to add an additional data format (be it *GML* [11] or *GDF* [57] or others) from which a graph can be loaded without changing any existing code.

Web context. Since this project is embedded into *REWERSE* [16], a **Semantic Web** perspective is adopted. Therefore, interoperability on the World Wide Web is a crucial requirement. As will be shown in section 5.3.2, by the use of the *Ontology for Transport Networks* (*OTN*) [63] encoded in *OWL* [27] (a special markup language for ontologies defined by means of the generic markup language XML) along with *GXL* [8] (another XML derivate), the particular importance of on-line data exchange is accredited.

Integration is also an important aspect, since querying **distributed data sources** invariably entails handling highly **heterogeneous** formats for representing data. For practical purposes, putting data together from distributed sources into the uniform model is vital. Therefore, by sticking to open standards like those proposed by the *W3C* [29], this issue is eased.

However, as perceived in section 2.3.3, indeed a lot of on-line routing system already exist. So the requirements are to a great extent described by the functionality of those systems. Yet the important difference clearly is found on the **semantic** level: the universal approach for representing various transportation networks presented in chapter 4 will be adapted, thus enabling the system to be flexible for many different tangible application scenarios to come.

5.1.2 Use Cases

Basically, one can distinguish between an **interior view** of the framework as a **system** from the standpoint of a system administrator, and an **exterior view** as an **application** from a user requesting routing information. Loading a graph from a physical file and modifying it afterwards by adding or removing vertices, edges and attributes may be a list of use cases in which the actor 'system administrator' interacts. The former can be regarded as an obligatory preprocessing step required for initialising the system to fulfill the processing of queries on the loaded graph. In contrast, the exterior users' view of the framework as an application implies uses cases which entail interaction with end users of the system, which essentially are different forms of queries.

Categorisation of typical routing queries:

- **origin-destination pair** two different locations (mapped to source and target vertices) for which a shortest path shall be determined.
- additional **constraints** e.g. sequence of transport modes or constraints on the locations or path
- intermediate locations, either in an sequence (**ordered**) or in an arbitrary order (**unordered**) This problem is well-known in graph theory as the travelling salesman problem or Hamiltonian path.

By examining these types of queries more closely, one may get the idea of putting results together in a special query language for routing. However, an accurate formal definition of a query language would exceed the temporal restrictions of this work, and so is noted for future works.

5.2 Existing Components Used

Taking into account the large open source community offering plenty of frameworks and libraries for programmers in Java, for the sake of not reinventing the wheel, *TransRoute* is not implemented from scratch, but rather is assembled from some existing parts freely available. In the following, the selected application programming interfaces (APIs) which are used by *TransRoute* are presented in detail, their abilities and drawbacks being highlighted.

5.2.1 JUNG: The Java Universal Network and Graph Framework

For the central data structure forming the very core of the *TransRoute* system together with its algorithms, the *Java Universal Network and Graph* framework (*JUNG*) [21] has been handpicked, enabling internal representation and manipulation of graph structures since it respects well the principles of object-oriented programming and indeed offers a lot of objects and methods. Being an open-source framework freely available on the Internet and continuously under progress, the *JUNG* framework offers a rich application programming interface (API) with plenty of functionality for any Java developers preoccupied with writing applications which do use graphs in some way.

Inspired from different research areas all dealing with graph structures, *JUNG* is founded on the concepts of object-oriented modelling, first above all inheritance (i.e. abstract classes and interfaces), from which re-usability consequently results, making it ”a general language for handling graphs as data” [88]. It has proven to be useful in two different applications, one for simulating social networks with agents and the other for analysis and visualisation of a large authorship network.

The basic features, including representation of both directed and undirected graphs, multigraphs and hypergraphs, which can all be annotated with meta-data (attributes), but also its basic algorithm repertoire (shortest path, flows and ranking) are manifold. For graphs and their elements, a type hierarchy exists consisting of interfaces and abstract classes on the root levels and classes on the leaf levels which can be instantiated.

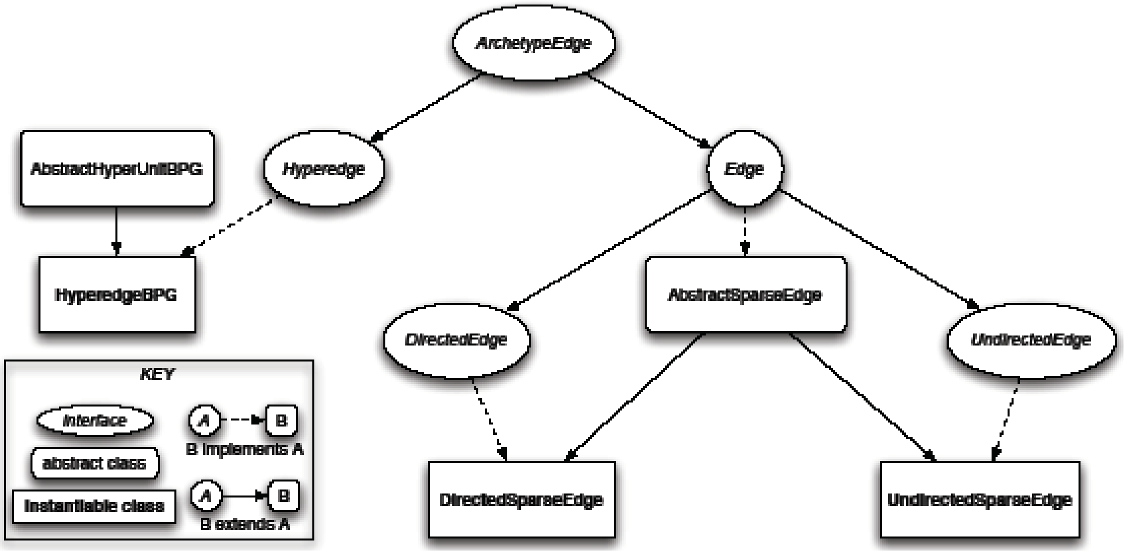


Figure 5.1: An Example of the JUNG Graph Type Hierarchy for Edges (from [88])

Among others, its generic **predicate and filtering mechanisms** are to mention especially, which allow extraction of certain portions of a graph, i.e. extracting sub-graphs. Essentially, predicates constitute flexible means for both constraining and filtering graphs.

A **predicate** is a high-level expression which yields a boolean result (either $true$ or $false$) when being evaluated on a graph element (graph, vertex or edge) including all its features which can be accessed in an object-oriented way. As a simple example, $Vertex.getDegree() > k$ can be expressed as a *VertexDegreeGreaterThanPredicate* class, which derives from the general *VertexPredicate* class in *JUNG*, using k as a parameter in the constructor. Since graph elements can either pass or fail a predicate, they are divided into two groups. Using a filter on the predicate, the first group of elements having passed the predicate is bundled into a sub-graph of the original.

Constraints for graph elements can be defined this way, such as the built-in constraints *JUNG* uses for checking integrity of directed graphs or those not permitting parallel edges, but also for user-defined predicates. For routing purposes, these user-defined predicates allow filtering for any attributes and therefore are highly valuable. Being very stringent indeed, the predefined constraints for graphs ensure consistent models. For transportation networks in particular, this strict approach for instance implies that vertices can only belong to one graph at the same time. However, orphaned vertices are possible which do not belong to any graph.

Offering a mechanism for extracting subgraphs which match a given predicate, **filters** can basically be regarded as pipes (like in the Unix shell). It is even possible to combine two or more filters by the use of the $SerialFilter$ class, which explains the comparison to pipes.

By offering a separate **implementation for sparse graphs** which is using adjacency lists and thus requiring $O(|V| + |E|)$ space only [88], the savings in space are remarkable (a factor of 10^4 according to [88]).

Indeed, sparse graphs can commonly be found in application areas, like in transport networks of large scale. In particular, street networks with an average vertex degree above two, but also more connected networks like train lines can be efficiently represented by these special sparse graph implementations proving to be highly valuable.

Just to give a basic idea, the methods which can be used for manipulating graph elements, taken from the *JUNG* manual (see [21], [20]) are enumerated:

Graphs:

newInstance(): Returns a graph of the same type as the graph on which this method is invoked.

addVertex(v): Adds the vertex v to this graph, and returns a reference to the added vertex.

addEdge(e): Adds the edge e to this graph, and returns a reference to the added edge.

getVertices(): Returns the set of all vertices in this graph.

getEdges(): Returns the set of all edges in this graph.

numVertices(): Returns the number of vertices in this graph.

numEdges(): Returns the number of edges in this graph.

removeVertex(v): Removes the vertex v from this graph.

removeEdge(e): Removes the edge e from this graph.

removeVertices(s): Removes all vertices in the set s from this graph.

removeEdges(s): Removes all edges in the set s from this graph.

removeAllEdges(): Removes all edges from this graph, leaving the vertices intact.

removeAllVertices(): Removes all vertices (and, therefore, edges) from this graph.

copy(): Performs a deep copy of the graph and its contents.

Vertices:

getGraph(): Returns a reference to the graph that contains this vertex.

getNeighbors(): Returns the set of vertices which are connected to this vertex (by edges).

getIncidentEdges(): Returns the set of edges which are incident to this vertex.

degree(): Returns the number of edges incident to this vertex.

getEquivalentVertex(g): Returns the vertex in the specified graph g, if any, that is equivalent to this vertex.

isNeighbor(v): Returns true if the specified vertex v and this vertex are both incident to at least one edge, and false otherwise.

isIncident(e): Returns true if the specified edge e is incident to this vertex, and false otherwise.

removeAllIncidentEdges(): Removes all edges that are incident to this vertex from the graph.

copy(g): Creates a copy of this vertex in the specified graph g.

Edges:

getGraph(): Returns a reference to the graph that contains this edge.

getIncidentVertices(): Returns the set of vertices that are incident to this edge.

getEquivalentEdge(g): Returns the edge in graph g, if any, that is equivalent to this edge.

numVertices(): Returns the number of vertices that are incident to this edge.

isIncident(v): Returns true if the specified vertex v is incident to this edge, and false otherwise.

copy(g): Creates a copy of this edge in the specified graph g.

Notably, creating edges and vertices (by using the constructor of the respective class) and adding them to a graph (by using the $addVertex$ or $addEdge$ methods of a graph object) are two distinct operations, which makes it possible to create edges and vertices not belonging to any graph, but also complex creation processes (e.g. by additionally decorating the graph with attributes). Of course, they

can be removed from a graph as well. Graph elements can be attached to and detached from graphs this way, after having been created.
Example:

```
Graph g    = new DirectedSparseGraph();
Vertex v1 = g.addVertex(new DirectedSparseVertex());
Vertex v2 = g.addVertex(new DirectedSparseVertex());
Edge e     = g.addEdge(new DirectedSparseEdge(v1, v2));
```

In the second step, after the creation of the according graph element, it is added to the graph. At this very moment, all integrity constraints imposed by the graph (users can additionally specify their own constraints) are checked, and if successful, the vertex or edge is added to the graph:

- A vertex or edge either has no graph or belongs to *exactly one graph*, to which it can be added only once.
- An edge can only be created to vertices belonging to the *same graph*. Especially, it is not possible to create edges to orphaned vertices with no graph. Of course, the edge type must correspond the the vertex and graph types.

The last restriction prohibits creating boundary crossing edges in hierarchic graphs (see chapter 4). A solution to this problem is the possibility of **copying vertices** by maintaining equivalence offered by *JUNG*. All vertex attributes are copied as well. So a copy of one vertex can be added to the graph of the other one, thus enabling the creation of such an edge. As a boundary crossing edge passes hierarchic elements, it is the connection of the uppermost vertices where this edge belongs to. So the graph of this super-edge contains the boundary crossing edge along with the two copies of vertices.

A large variety of **algorithms** ranging from connectivity and clustering over flows to shortest path computation are rounding out the abilities of *JUNG*. As an example, an implementation of Dijkstra's algorithm is available, but also other algorithms including the iterative $EdmondsKarpMaxFlow$ for solving the maximum flow problem between source and sink vertices in flow networks given edge capacities [35, 52], a $WeakComponentClusterer$ separating a graph into its weak components (the maximal undirected subgraph in which all its pairs of vertices are mutually reachable), a $KNeighborhoodExtractor$ extracting a subgraph whose vertices are within the distance k from at least one of the starting vertices given, a $VoltageRanker$ calculating the ranks of vertices "according to their 'voltage' in an approximate solution to the Kirchoff equations" [20] and many more.

There are two ways of associating data with graphs: either by extending the *JUNG* classes representing the graph elements by sub-classing (an object-oriented feature), or by using their built-in mechanism for storing user data. Both approaches are used by *TransRoute* for different purposes. The attributes not known in advance from the ontology classes are taken over in the user data repository offered by *JUNG*, whereas the hierarchic and geometric concepts pertaining to all transport networks are realised by sub-classing.

Basically, the user data repository of *JUNG* consists of built-in key-value pairs attached to any graph element. Manipulating it can be done by a command in the form of *graphElement.add*$[set]$*UserDatum(Object attributeKey, Object attributeValue, UserData.SHARED)* or *graphElement.get*$[remove]$*UserDatum(Object attributeKey)*

Higher-level access to this repository is also possible, e.g. by the *StringLabeller* class which implements a bidirectional mapping between vertices and $Strings$ representing those, for fetching vertices from a given $String$ representation.

The major disadvantage of *JUNG* currently is the support for persistence: *GraphML* [7], which is a solid, well-defined standard with origins from graph drawing, is used for this purpose. However, not yet fully integrated into *JUNG* except for the basic functionalities, only a small part of *GraphML* can be used with *JUNG*. For instance, hierarchic graphs which can be expressed in *GraphML* have no counterpart in *JUNG*, thus can't be used. As *JUNG* is still under active development, hopefully its persistence components will be enhanced.

5.2.2 GXL: The Graph Exchange Language with its API

In order to overcome this evident shortcoming, the *Graph eXchange Language* (*GXL*) [8] defined in an XML vocabulary together with its Java API [19], are used by *TransRoute* for exchanging graphs and making them persistent instead. Brandes et al. [41] profoundly investigate the commonalities and differences between these two standards, concluding that both are well for use as a persistent graph representation, since their advanced models include nested graphs and attributes. According to [41], both are offering about the same amount of functionality to describe graph structures and additional user data within, and can be converted into one another via *XSLT* stylesheets [30] .

Like in *GraphML*, nested graphs being "useful when a complex graph integrates various abstraction levels" [41] can be expressed in a natural way by a <graph> XML element recursively contained in another <node> or <edge> XML element of a higher <graph>.

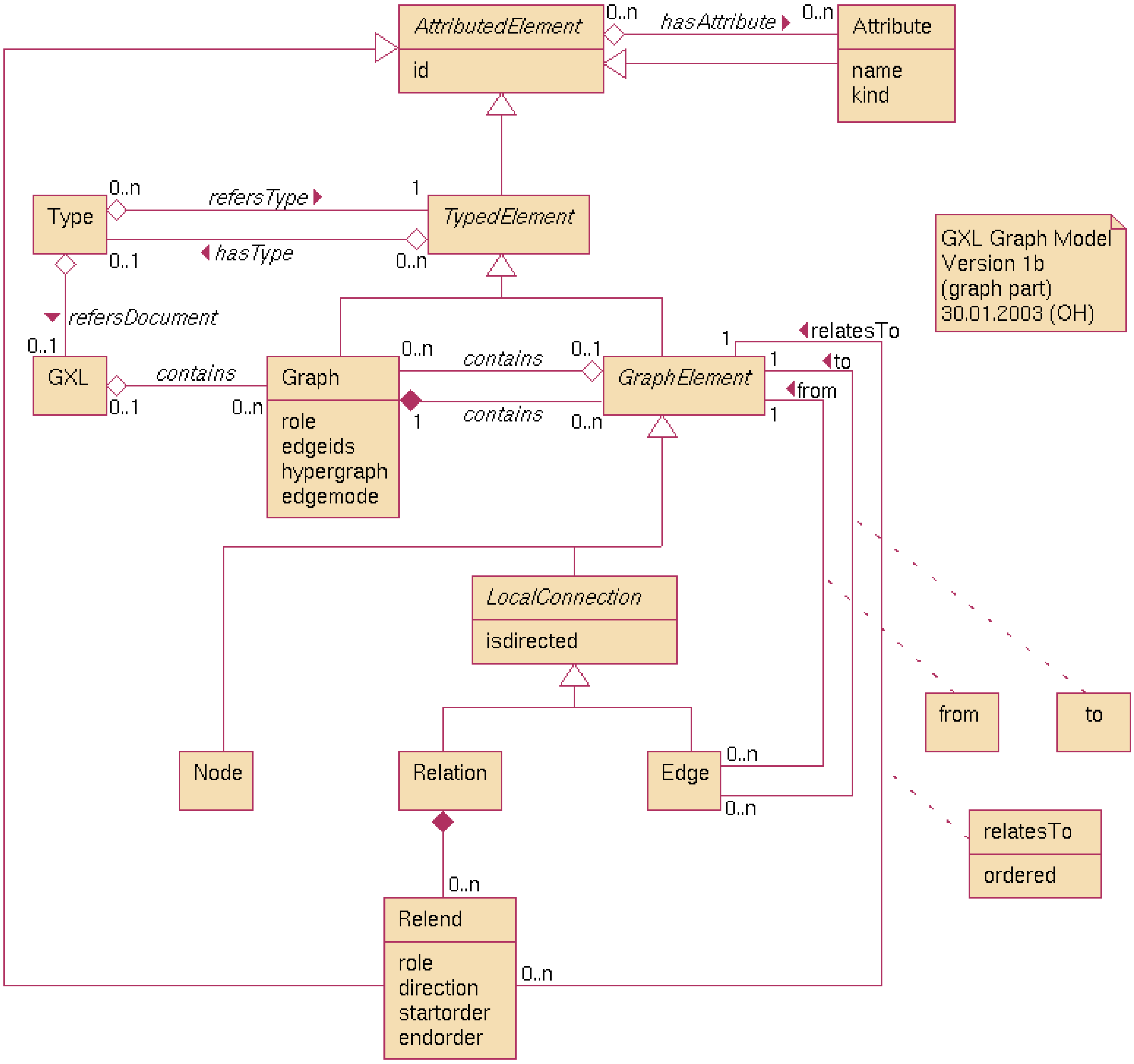


Figure 5.2: The GXL Graph Model (from [19])

The <attr> element allows to decorate a graph with meta attributes of several data types including atomic basic types (<string>, <bool> <int>, <float>, <enum>) and composite vector types (<bag>, <seq>, <set> and <tup>) [19]. As <node> and <edge> elements within a <graph> element are unordered, they need a unique id. In the annex (see B.2.1), the *GXL* document type definition (DTD) is listed which can be consulted for more detail on the syntax of *GXL*. Some instances of *GXL* files can be found there, too.

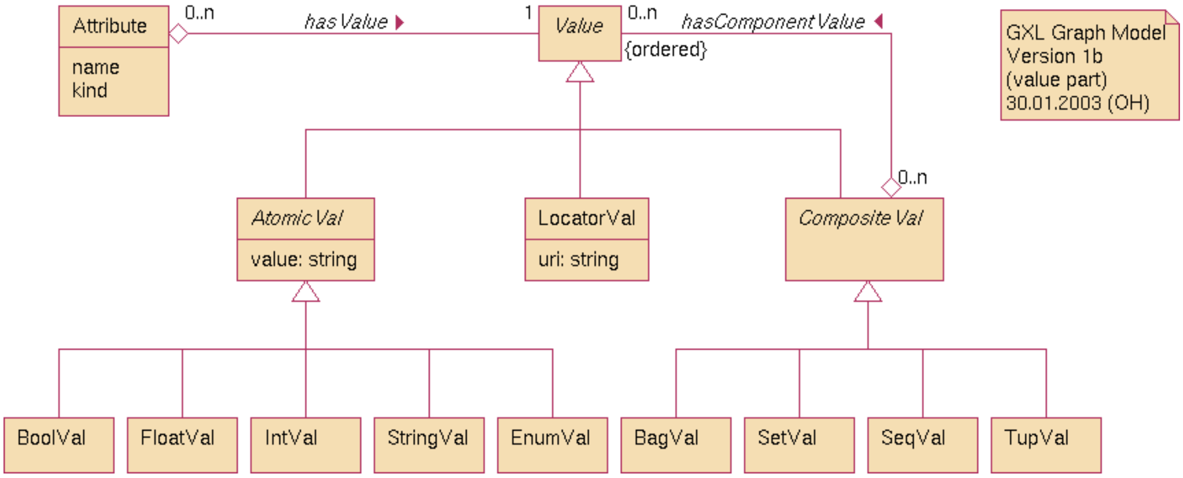


Figure 5.3: The GXL Attribute Data types (from [19])

In an earlier version, the XML standard *Geography Markup Language* (*GML*) [11] for Geographic Information Systems has been evaluated as a data exchange format for the use with its Java API *GML4j* [18]. However, up to date, the new concepts of the current version 3.0.1 are not yet fully supported by tools, but only those of version 2.0. Since an appropriate hand-written parser would entail a lot of work, the extraction of GML is postponed until hopefully the API further develops for enhanced support in Java. A possible alternative to this approach would be transforming *GML* files into *GXL* by the means of an *XSLT* stylesheet.

5.2.3 OWLAPI: An API for the Ontology Web Language

In terms of computer science, ontologies (originating from philosophy) are an important mean to conceptualise knowledge of a domain in a formal manner. By abstractly defining entities and relations prevailing in a domain with the aid of mathematical logics and thus offering unambiguous clear semantics, they exceed the capacities of simple schemata and data type definitions by far. Knowledge representation basically distinguishes between raw data, information and knowledge. Data solely cannot be evaluated in a proper way without knowing the meaning behind them. For semantic evaluation of data, ontologies offer the means to express further information than mere data, similar to dictionaries and thesauri [37, 27]. In a way, a schema can be seen as a primitive equivalent of an ontology, however not as expressive as its counterpart.

Current efforts of the *World Wide Web Consortium* (W3C) in this field have concentrated on the proposal of the *Ontology Web Language* [27], which shall be the building block the *Semantic Web* will be constructed from.

The *OWLAPI* [13, 37] offers application programmers access to *OWL* ontologies representing domain knowledge by building up an in-memory structure similar to *DOM* [28]. Additionally, it offers some basic functionality concerning the *OWL* classes and their relations. However, they are very limited, and had to be enriched by several functions in order to be applicable to *TransRoute* when checking integrity constraints imposed by the domain model.

5.3 Architecture

After having highlighted the existing components employed in *TransRoute*, a big picture of the system is shown, explaining the use of the APIs and extensions:

5.3.1 Design Principles

Like [70], who have built up a system representing transport networks at the dawn of object-orientation, *TransRoute* is consequently using benefits and techniques of this current programming paradigm: Abstraction is being achieved by inheritance (JUNG graph element classes as super-classes for generic $TransRoute$ graph model classes), which fosters re-usability remarkably, and furthermore by **distinguishing** a **specification from** its **implementation**, with the aid of abstract classes and interfaces: $IAlgorithm$ is used for a general graph algorithm, from which $IQuery$ for a general query depends, or $IReader$ and $IWriter$ for specifying the contract data exchange components must fulfill.

```
package de.lmu.ifi.pms.transroute.geometry;

public abstract class DistanceMetric {
    public abstract double getDistance(Position a, Position b);

    public double deltaX(Position a, Position b)
    { return Math.abs(a.x() - b.x()); }

    public double deltaY(Position a, Position b)
    { return Math.abs(a.y() - b.y()); }
}
```

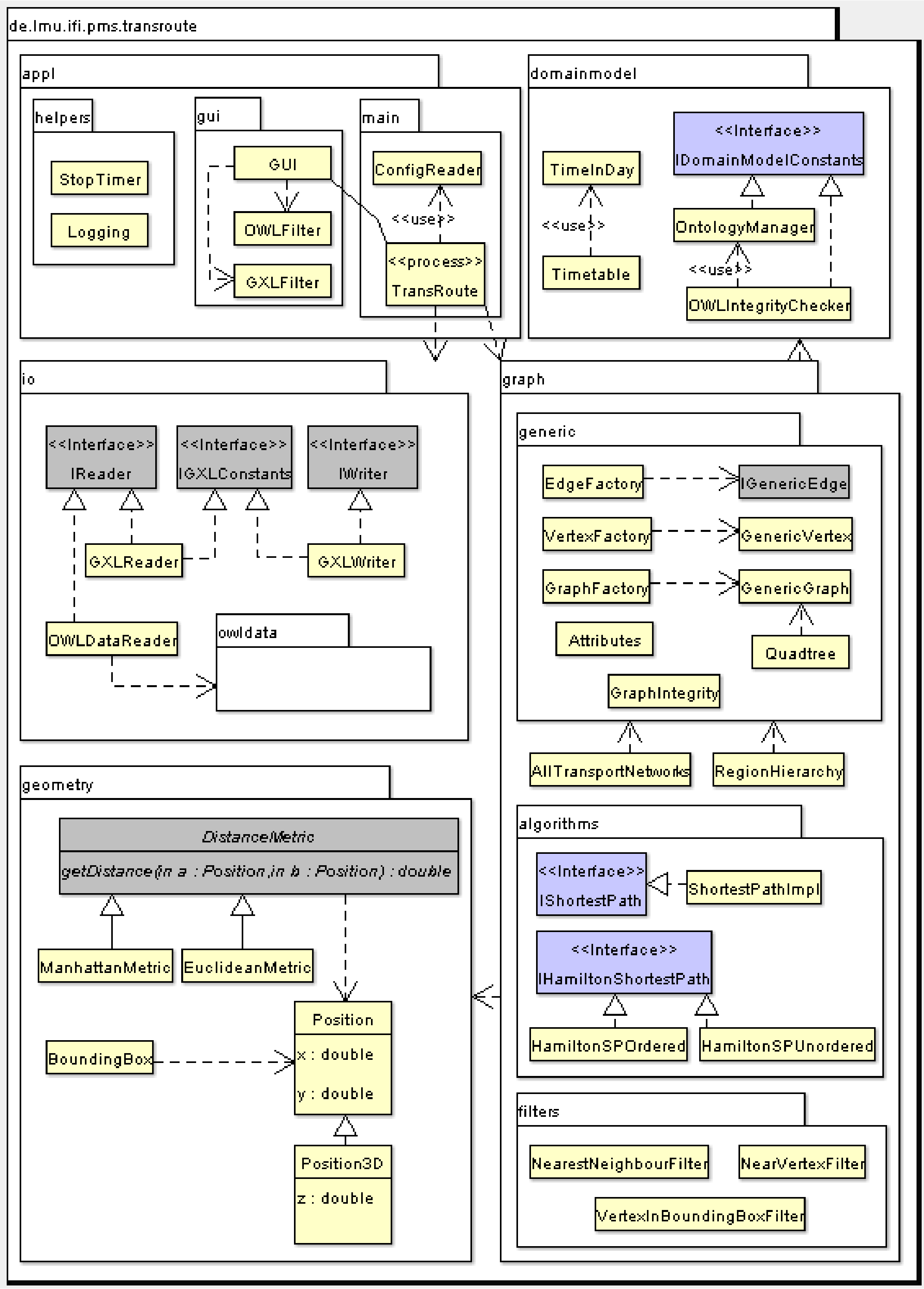


Figure 5.4: System Architecture of *TransRoute*

```
package de.lmu.ifi.pms.transroute.io;

public interface IReader {
    public GenericGraph     load(String fileName);
}
```

```
package de.lmu.ifi.pms.transroute.io;

public interface IWriter {
    public void        save(GenericGraph g, String fileName);
}
```

Separation of concerns is a higher-level architectural principle for object-oriented software, which can be achieved by the techniques mentioned before, supplemented by using **design patterns** as best practice knowledge for special recurrent problems in object modelling: Graphs, edges and vertices being modelled as separate classes, their according factory classes abstract over the creation of these objects. Beyond, at the level of classes are also queries and algorithms. Treated as command objects, queries may be executed by their *execute(IAlgorithm Alg)* method (an algorithm can be passed as parameter). Since one can choose from different algorithms which all implement the very same contract imposed by the interface if available (by applying the strategy pattern, i.e. more than one algorithm can implement the interface needed), algorithms are also objects implementing the $IAlgorithm$ interface. In order to compare properties of different algorithms, this notion can be used and accordingly is of practical value. Special filters and predicates, which are sub-classing the general ones being offered by the *JUNG* library, are first-level classes, too.

```
package de.lmu.ifi.pms.transroute.graph.algorithms;

public interface IShortestPath extends IAlgorithm
{
    public Vector<IGenericEdge> getPath(GenericVertex vFrom,
            GenericVertex vTo, String costAttribute);

    public Number getDistance(GenericVertex vFrom, GenericVertex vTo,
            String costAttribute);
}

package de.lmu.ifi.pms.transroute.graph;

public interface IQuery {
    public String execute(IAlgorithm Algorithm);
}
```

5.3.2 An Ontology for Formalising Domain Knowledge

Ontologies, belonging to knowledge representation, have proven to be well-suited to describe complex entities and features of the real world in a machine-processable way. In a previous work, the *Ontology for Transport Networks* (*OTN*) [63] has been developed inspired by the *GDF* standard [57]. The ontology used in this framework is using *OTN* in a lighter version[1], only containing the most important elements for graphs. Additionally, some concepts have been added, e.g. for reasoning on regions.

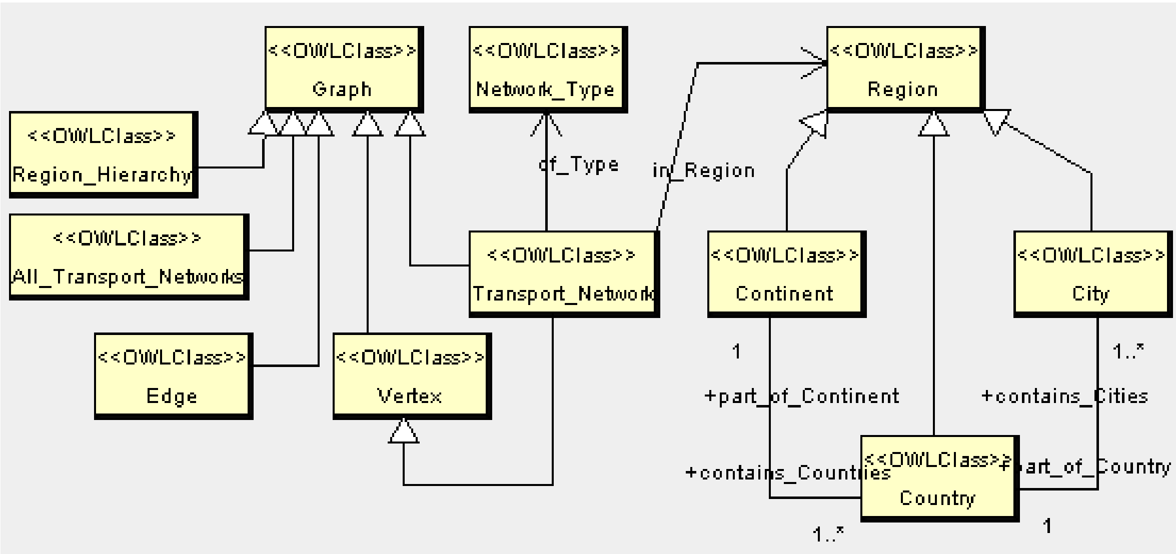


Figure 5.5: Important Entities and Relations in the Ontology

This subset of the *OTN* ontology is used for the abstract modelling layer for representing domain concepts. Graphs, vertices and edges can hence be created with a certain domain concept, being the $String$ representation of the corresponding OWL object. While creating these graph elements, this $String$ is checked to be matching an object in the ontology, otherwise the graph element cannot be created. It is basically this idea that ensures the integrity of the generic graph elements built with the appropriate domain concept, as they are always checked against the ontology to be valid. As the generic graph elements are depending on the ontology, extending the latter is the only way to open further concepts for the construction of graphs. A class in the ontology may have several graph representations, also depending on the level of abstraction. *TransRoute* recognises all concepts of this *OWL* ontology and checks that its graph elements do have a valid *OWL* concept.

[1] since $OWLAPI$ can't make use of any XML schema types defined in OTN

Figure 5.6: OWL Class Hierarchy of Domain Model in *Protégé* Editor [15].

A concept, which is a subclass of $Vertex$ (the OWL class) can be filled into a generic vertex, and the same applies for edges and graphs respectively. Due to the hierarchic nature, vertices and edges themselves are subclasses of the $Graph$ OWL class. $AttachableToEdge$ are vertex types with special semantics, for instance buildings or parking lots, which can be attached to edges since they are bounding to them.

Simple attributes in an object-oriented sense are modelled as data type properties in OWL, being mapped to the fitting OWL classes. Any property in OWL has both a domain (the OWL class it belongs to) and a range (an XML schema data type in case of a data type property, or another OWL class in case of a object property). Attributes have been slightly adapted by creating attribute groups as prefixes and giving them a data type (since the *OWLAPI* framework used to process the ontology cannot understand XML schema data types, but instead only provides a single class as a dummy implementation with no access to the corresponding XML Schema type).

Figure 5.7: Data type (Green) and Object Properties (Blue) in *Protégé* Editor [15].

For example, $freeParking.boolean$ (OWL data type property) belongs to the OWL class $Parking$, or $suitable\_for\_disabled.boolean$ applies to all $Stop\_Areas$. Unless being too special and therefore appertaining to OWL classes lower in the type hierarchy (e.g. the $phys.surface\_Condition$ is limited to $Road$ and $Road\_Element$ as it does not make any sense for an $Airway\_Element$ at all), the attributes prefixed with $legal$, $traf$ and $phys$ are of the OWL domain $Edge$.

5.3.3 Generic Graph Data Structure

The concepts from the ontology are filled into a graph structure, which is the core component of the framework, as it models graphs which will be the keystone of the algorithms altogether with the dynamic manipulation, i.e. the possibility to add and remove graph elements and properties. Generic is probably the best word for this structure, since it can be instantiated with the according ontology class explaining the domain semantics of the adequate graph element.

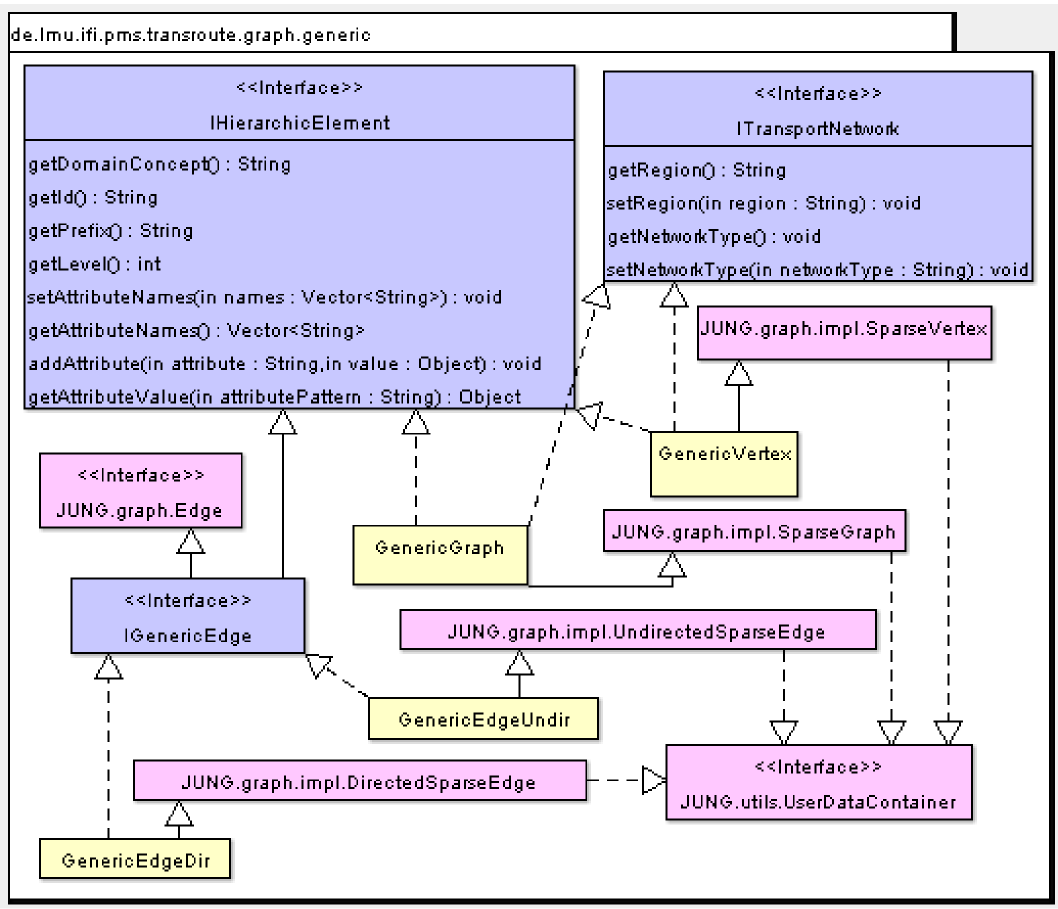


Figure 5.8: Inheritance of Graph Data Structure to JUNG Graph Elements

The graph elements of *TransRoute* extend those of the *JUNG* library insofar as they incorporate an inherent hierarchic structure (by the implementation of the $IHierarchicElement$ interface) and their associations (see figure 5.9: Navigating from a vertex to its nested graph and vice versa is possible, as well as for an edge to its attached vertices graph and other subgraph containing elements of a lower level. Attached vertices and their bounding edges are mutually reachable, too.

This notion of hierarchic graphs constitutes a true extension to the graph classes offered by *JUNG*, because of the additional possibilities of not only specifying a hierarchy of graphs, but also of connecting vertices from separate graphs by boundary crossing edges, together with their semantics (see [45]).

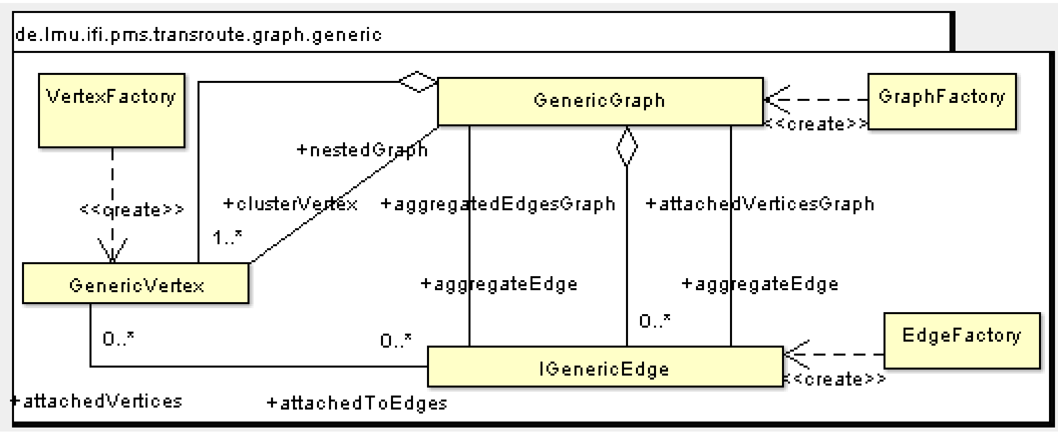


Figure 5.9: The Hierarchic Graph Elements Along With Their Factory Methods

```
package de.lmu.ifi.pms.transroute.graph.generic;

public class EdgeFactory {
    public static IGenericEdge create(String id, String DomainConcept,
             GenericVertex vFrom, GenericVertex vTo, boolean directed)
    {
        if (GraphIntegrity.inSameGraph(vFrom, vTo))
        { .. /* return new GenericEdgeDir or Undir */ }
        else return createBoundaryCrossingEdge(id, DomainConcept, vFrom,
                                                 vTo, directed);
    }

    protected static IGenericEdge createBoundaryCrossingEdge(String id,
        String DomainConcept, GenericVertex vFrom, GenericVertex vTo,
                                                     boolean directed)
    {
        GenericGraph gFrom = vFrom.getGraph(), gTo = vTo.getGraph();
        ..
        GenericVertex cFrom = gFrom.getClusterVertex(),
```

```
                    cTo = gTo.getClusterVertex();
        ..
        if (GraphIntegrity.inSameGraph(cFrom, cTo))
        { .. /* first possibility (left side of following picture) */ }
        ..
        if (GraphIntegrity.isReachableFromAttachedVertex(vFrom, vTo))
        { .. /* second possibility (right side of following picture) */ }
        ..
        /* return appropriate boundary crossing edge */
    }
}
```

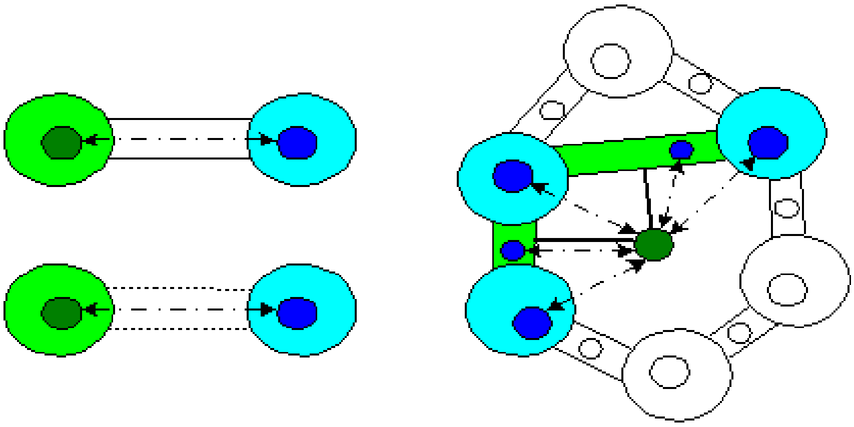


Figure 5.10: Boundary Crossing Edges From a Vertex (Green), Also For Attached Vertices (Right)

```
package de.lmu.ifi.pms.transroute.graph.generic;

public class GraphIntegrity {
    public static boolean isReachableFromAttachedVertex
     (GenericVertex attachedV, GenericVertex destinationV)
    {
        for (IGenericEdge edge : attachedV.getAttachedToEdges())
        { /* all edges attachedV is attached to */
            GenericGraph reachableGraph = edge.getAggregatedEdgesGraph();
            /* their aggregated edges subgraphs */         /* ret: */
            if (reachableGraph.equals(destinationV.getGraph())) return true;

            reachableGraph = edge.getStartsAtVertex().getNestedGraph();
            /* the nested graphs of their adjacent vertices */ /* ret */

            reachableGraph = edge.getEndsAtVertex().getNestedGraph();
            /* the nested graphs of their adjacent vertices */ /* ret */
        }
    }
}
```

Over and above, the specific graph elements contain geometric properties, for instance does a $GenericVertex$ have a location attribute of a $Position$ class (x and y coordinates mandatory, z coordinate is extra) as defined in the $geometry$ package. $BoundingBox$ is the equivalent for a $GenericGraph$, and a generic edge contains a vector of intermediate positions.

All graph elements implementing the $UserDataContainer$ interface of *JUNG* by sub-classing the *JUNG* graph elements are able to store meta-attributes (kept on a generic level, with some special predefined instances) in a container with key-value mapping decorating these elements. Indeed, they constitute valuable additional information, as reasoning cannot be applied well on the bare graph structure without any additional attributes.

Different levels of abstraction can be applied using the system: On a higher network level, each transportation network is represented as a separate vertex having a certain network type and being part of a region (continent, country or city) according to the RCC-8 calculus [58] (see chapter 4) as defined in the domain model ontology. Convenient methods for handling these types of graphs can be found in $AllTransportNetworks.java$. Further extensions of personalisation levels may use this approach for filtering appropriate vertices transportation networks are represented by, reflecting the users' preferences. Especially for reasoning, the pruning of networks a certain user is not willing to use in a preliminary step may accelerate the routing process drastically as less inter-connections between different networks have to be taken into account. Yet on a lower level, a transport network itself is modelled by a graph containing further vertices and edges of an abstraction level 1, which in turn can be refined according to the hierarchic structure to a level 2 and so forth. Encompassed by the graph structure are also buildings, which being represented as edges whose types are subclasses of the $Attachable\_To\_Edge$ ontology class, are contained in edges. This is a good example for demonstrating the flexibility of the generic graph structure, which shall be applicable for multiple purposes.

5.4 Proof of Concept

Probably one of the most annoying problems adhering to developing software which will be serving a large amount of users with a comparably huge extent of data is the sheer complexity of such a system. Speaking in terms of computer science, **scalability** and **performance** are of utmost importance.

For an analysis of the metropolitan network data of Munich, originating from the local department of construction, performant algorithms and rapid data access are essential due to the enormous amount of data on hand. As an initial **case study**, this data is well-suited to demonstrate the usage and capabilities of *TransRoute* by showing that it can cope with data of a realistic dimension: Merely the **digital street map of Munich** encoded in an XML vocabulary exceeds several megabytes of disk space, which makes the data access rather slow.

On this data encoded in an OWL instance file containing a total of *13300 vertices* and *19887 edges*, the components of the framework were unleashed in order to consider their realistic performance [2]:

file name	load time [s]	save time [s]	size [KB]
S4.owl	0.9	n.a.	28
S4.gxl	0.1	3	25
U2.owl	0.7	n.a.	46
U2.gxl	0.2	2.6	42
MunichNetwork.owl	405	n.a.	7478
MunichNetwork.gxl	43	18	18097

Table 5.1: Experimenting with *OWL* and *GXL* Data Source Formats

At first, a problem of insufficient heap size concerning the Java virtual machine ($OutOfMemoryError$) was encountered, which was solved by increasing the heap from initial 64 megabytes up to 512 megabyte, using the *-Xmx* 512000000 and *-Xms* 512000000 parameters for maximal respective minimal values when launching the virtual machine.

It has proven that converting the *OWL* data to *GXL* for the large file on the one hand needs more disk space (about a factor of 2), however traded off for a faster reading and construction of the graph structure by a factor of 10 on the other hand (the *GXL* file could be loaded within 40 to 50 seconds on a 2.2 gigahertz processor).

The response time for queries depending on their complexity varies between less than a second and several seconds, which obviously confirms the soundness of the approach taken by a greater preprocessing phase (loading the graph structure) for rightful query processing.

[2] System specifications for measured times: 2.2 GHz processor and 1024 MB of main memory.

6 Summary

6.1 Results

During the course of this thesis, applications of graph theory pertaining to transport networks have been investigated, above all the subtleties of uniformly modelling all different aspects in a generic graph structure and efficient algorithms for answering routing queries.

The framework *TransRoute* has been implemented which is capable of representing hierarchic, multi-modal graphs: Within the graph structure, not only hierarchic networks containing multiple levels of abstraction can be modelled, but also connections between different networks which are significant in a multi-modal environment.

Notably, a separation between graph structure and knowledge representation has been respected. In particular, graph elements can be created by specifying which concepts describing domain knowledge in the ontology are represented by the former.

As a first case study, real-life data of the Munich street network has been loaded into the graph structure, enabling basic routing queries to be answered by the system.

Concluding, these semantically enhanced graphs are a basic structure for representing all kinds of transport networks uniformly.

6.2 Future Work

By far, the achievements of this thesis have not exhausted research, on the contrary, still, there are a lot of prospects in this field.

Vehicles A first extension may be considering individual vehicles within transport networks, thus using a microscopic model for dynamics [75, 73, 47]. Every single vehicle could be tracked this way, which is especially interesting for trains. Also attributes which characterise vehicles may be taken into account for reasoning, e.g. for filtering fitting networks. Intermodal transportation has to be considered from this aspect as well.

Traffic Messages Since being up-to-date is important in traffic, the stream processing of the **Traffic Message Channel** (TMC) for propagating information on congestion and accidents is a further crucial issue. Thus, adding an updating component to the system, i.e. connecting it to the TMC evaluation, will be a future goal.

Data Integration Further I/O components enabling data exchange may be added as plug-ins, for instance for GML or other formats. Also, adding interchange edges between different modes of transportation will be a task having reached a point when more networks will be integrated into the model.

Displaying Maps A component for the graphical user interface (GUI) is under construction, which will be able to display various networks on a map, but also the results of routing queries. A client / server architecture will be needed, so that different users can log on to the system.

User Profiles In addition, modelling personal preferences in a user profile will be a central point for extending this framework. In order to accredit personalisation its due attention, setting up user profiles reflecting individual likes and dislikes and expressing both abilities and disabilities will influence the behaviour of a routing algorithm according to individual perceptions.

Reasoning Last but not least, a reasoning component as the core element beyond capabilities of usual graph algorithms employed for routing taking into account all additional aspects is the higher goal to strive for. For this, a more general notion of space and further ontologies will be required. Also, higher-level spatial query and navigation languages may be elaborated for this purpose.

A Glossary

A.1 Domain

Intermodal Transportation . Containers which can be loaded onto trains, cargo ships etc. are a good example for intermodal transportation. Ferries or trains carrying cars can also be classified into this category. So multi-modal transport may be intermodal, however not necessarily.

Mode of Transport(ation) . A specific instance of a transportation system, eg. a metropolitan underground system, shorthand underground, or a certain street network. Synonym: **Transport Mode**.

Multi-modal Transportation . Transportation consisting of several heterogenous transportation modes joined together. The modes can be changed, and on this occasion, a chain is well-suited for describing their sequential order. This notion is not to be confounded with **intermodal transportation**.

Navigation . Very similar to routing, except for one point: In contrast to routing, navigation is done ad-hoc, when the user is already active, using a vehicle in a transportation network. This may imply more adaptivity and dynamic aspects, as a route choice may alter from the proposed one, and a new calculations have to be ignited. In a nutshell, navigation can be seen as on-the-fly routing.

Network Topology . The basic structure of a network, thus the way its nodes are connected by edges. Substantially, topology is a concern of graph theory (see chapter 3). One distinguishes between several forms of network topologies. (see section 4.2)

Path / Way Finding . The process of searching for a sequence of connections linking together a source and a destination on a given network. Most often, graphs are the structure to be scanned.

Routing . In this context, routing is a special case of traffic assignment, where an optimal plan for a trip or a journey (referred to as a route) from one location to another is sought. This task may require the due consideration of both different transport modes (the ability to handle multi-modal transport) and human preferences and personal abilities. Routing is done in advance, as an essential part of planning activities (e.g. for a trip).

Traffic . The dynamic flow of a large number of minuscule (with respect to the magnitude of the entire network) vehicles through the net structure of a transportation mode.

Transit . A passing transportation through a certain realm. In the context of public transportation, (public) transit is virtually equivalent to (public) transportation.

Transport(ation) . A service providing a network for motion on it thus overcoming space.

Transport(ation) Network . The underlying physical infrastructure of a transport system linking different remote locations. Synonym: Infrastructure

Transport(ation) System . The amalgamation of a transport network together with its vehicles as well as the coordination and organisation of the delivered service.

Travel / Trip Planning . The organisation and arrangement of a voyage, choosing the right transportation modes at the right time in order to get to the target location.

A.2 Graph Theory

In the following, the basic terminology and concepts which are commonplace in graph theory [49, 52] are presented.

Acyclic . A graph with no cycles is called acyclic.

Adjacent Edges are edges with at least one node in common.

Adjacent Vertices are nodes with at least one edge joining them.

Arc is a synonym for a directed edge.

Branch . An edge of a tree.

Chain . An open edge sequence with distinct edges.

Circuit . A circuit is a closed chain, where the start node is equal to the last node.

Complete Graphs are graphs in which every pair of distinct vertices is joined by an edge.

Connected . A graph is connected if every pair of distinct vertices is joined by at least one simple chain.

Connected Component . An equivalence class in the transitive hull of the edge relation.

Cut-Set of a graph: The set of edges whose removal would disconnect the graph.

Cycle . A cycle is a closed path or a 2-regular subgraph of a digraph.

Degenerate Graphs are graphs in which the edge set E is empty, i.e. consisting only of vertices.

Degree . The degree of a vertex is the number of its incident edges.

Directed Acyclic Graph (DAG) . A directed graph which is acyclic.

Directed Graph (Digraph) . A graph $D_G = (V, E)$ whose edges are arcs.

Directed Tree . If there is a path from a node v_0 to all other nodes in the subgraph T, and T without directions is a tree, the T is a directed tree.

Distance . The weight of an edge can sometimes be encoding the distance between two nodes.

End-Points of an edge are the two vertices connected.

Eulerian Circuit . A circuit in which no edge appears twice.

Eulerian Chain . A chain in which no edge appears twice.

Hamiltonian Circuit . A circuit in which no vertex appears twice.

Hamiltonian Path . A path in which no vertex appears twice.

Incident . An edge is incident with its end-points.

In-Degree . The number of arcs entering a node, all being positively incident to it.

Isolated Vertex . A vertex with a degree of 0.

Isomorphic . Two graphs G_1 and G_2 are isomorphic, if a one-to-one correspondence of all vertices maintaining the connections and adjacency from G_1 to G_2 exists.

K-regular Graphs . Graphs in which all vertices have the same degree k are k-regular.

Minimum Spanning Tree . A spanning tree with minimal edge weight of a graph.

Networks are a special type of connected, directed and weighted graphs with distinct source and sink nodes describing a flow with capacity constraints among others.

Out-Degree . The number of arcs departing from a node, all being positively incidenct to it.

Root . The node v_0 of a directed tree connecting all others.

Path . An arc sequence with distinct arcs.

Pendant Vertex . A vertex with a degree of 1.

Self-Loop . An edge whose start and end vertex are the same.

Simple Chain a chain with distinct verties.

Simple Circuit . A connected 2-regular subgraph of an undirected graph.

Simple Path a path with distinct vertices.

Spanning Tree . A tree including all vertices of a graph.

Subgraph . A subgraph is completely contained in a graph.

Tree . A tree of an undirected graph is a subgraph where any two vertices are connected by exactly one chain.

Underlying Graph . For a directed graph D_G, the underlying graph (being undirected) is obtained by removing orientation from all arcs, thus considering them as edges.

Weighted Graph . A graph $G = (V, E)$ together with a weighing function $E \mapsto R$ mapping each edge a specific weight. Distance between two nodes is the sum of all edges on the path from the start node to its destination.

B Framework

B.1 Source Code in Extracts

B.1.1 GenericGraph.java

```
package de.lmu.ifi.pms.transroute.graph.generic;

import edu.uci.ics.jung.graph.decorators.StringLabeller;
..

/**
 * A graph for representing geographical, hierarchical networks like
 * transport networks and entities within those. ..
 * @author  Edgar-Philipp Stoffel
 */
public class GenericGraph extends SparseGraph implements IHierarchicElement,
                                                         ITransportNetwork {
    private String DomainConcept, Region, NetworkType;
    private GenericVertex clusterVertex;
    private IGenericEdge   aggregateEdge;
    private String prefix, id, level;
    private BoundingBox boundingBox;
    private Vector<String> attributeNames = new Vector<String>();
    private StringLabeller Labeller;

    /**
     * Use GraphFactory.create(id, owlClassName) to create a new instance of
     * a graph.
     * @param id The id to be assigned to this graph
     * @param DomainConcept domain concept in ontology (OWLClass) as String
     */
    protected GenericGraph(String id, String DomainConcept)
    {
        super();
        this.id = id; this.prefix = "";
        OWLEntity owlEntity = OWLIntegrityChecker.getOWLEntity(DomainConcept);
        if (owlEntity != null)
        {
            if (owlEntity instanceof OWLClass)
            {    // check if domain concept is valid in ontology
                if (OWLIntegrityChecker.isGraphConcept((OWLClass)owlEntity))
```

```
                this.DomainConcept = DomainConcept;
            else System.out.println("ntinvalid domain concept " + id +
                                    ": must be a graph type in ontology!n");
        }
        else if (owlEntity instanceof OWLIndividual &&
            OWLIntegrityChecker.isRegion((OWLIndividual)owlEntity))
                this.DomainConcept = DomainConcept;
            else if (OWLIntegrityChecker.isObjectProperty(owlEntity))
                this.DomainConcept = DomainConcept;
    }
    else System.out.println("ntinvalid domain concept for graph " + id
                                        + ": must be in ontology!n");

    if (owlEntity != null) Attributes.addAttributesFromDomainConcept(this);
    Labeller = StringLabeller.getLabeller(this);
}

..

/**
 * Finds a vertex in this graph by its id (recursively in its subgraphs).
 * @param id the id of the vertex to be searched for
 * @return the vertex
 */
public GenericVertex getVertexById(String id)
{
    GenericVertex Result = null;
    Result = (GenericVertex)Labeller.getVertex(id);
    if (Result != null) return Result;

    for (Object v: this.getVertices())
    {     GenericVertex genV = (GenericVertex)v;
        GenericGraph SubGraph = genV.getNestedGraph();
        if (SubGraph != null) Result = SubGraph.getVertexById(id);
        if (Result != null) return Result;
    }

    for (Object e: this.getEdges())
    {     IGenericEdge genE = (IGenericEdge)e;
        GenericGraph SubGraph = genE.getAggregatedEdgesGraph();
        if (SubGraph != null) Result = SubGraph.getVertexById(id);
        if (Result != null) return Result;

        SubGraph = genE.getAttachedVerticesGraph();
        if (SubGraph != null) Result = SubGraph.getVertexById(id);
        if (Result != null) return Result;
    }

    return null;
}

public Object getAttributeValue(String attributePattern)
{ return Attributes.getAttributeValue(this, attributePattern); }
```

```
    public void addAttribute(String attributePattern, Object attributeValue)
    { Attributes.addAttribute(this, attributePattern, attributeValue); }
}
```

B.1.2 GenericEdgeDir.java

```
package de.lmu.ifi.pms.transroute.graph.generic;

import edu.uci.ics.jung.graph.impl.*;
..

/**
 * A directed edge for representing geographical,
 * hierarchical network connexions. ..
 * @author  Edgar-Philipp Stoffel
 */

public class GenericEdgeDir extends DirectedSparseEdge implements IGenericEdge
{    private String DomainConcept;
    public static final String AGGR     = "AggregatedEdges";
    public static final String ATTACH     = "AttachedVertices";
    private GenericGraph aggregatedEdgesGraph;
    private GenericGraph attachedVerticesGraph;
    private String prefix, id, level;
    private Vector<Position> interpolationPoints;
    private Vector<String> attributeNames = new Vector<String>();

    /**
     * Use EdgeFactory.create() to create a new instance of an edge.
     */
    protected GenericEdgeDir(String id, String DomainConcept,
                        GenericVertex vFrom, GenericVertex vTo)
    {
        super(vFrom, vTo);
        this.id = id;  this.prefix = "";
        OWLEntity owlEntity = OWLIntegrityChecker.getOWLEntity(DomainConcept);
        if (owlEntity != null)
        {
            if (owlEntity instanceof OWLClass)
            { if (OWLIntegrityChecker.isEdgeConcept((OWLClass)owlEntity))
                this.DomainConcept = DomainConcept;
            }
        } else System.out.println("invalid domain concept for edge " +
                id + ": must be in ontology!");

        if (owlEntity != null) Attributes.addAttributesFromDomainConcept(this);
    }
```

```
public GenericGraph getGraph(){return (GenericGraph)super.getGraph(); }

/**
 * @return the origin vertex from which this edge starts
 */
public GenericVertex getStartsAtVertex()
{ return (GenericVertex)(this.getEndpoints().getFirst()); }

/**
 * @return the destination vertex where this edge points to
 */
public GenericVertex getEndsAtVertex()
{ return (GenericVertex)(this.getEndpoints().getSecond()); }

..

/**
 * @param pos an auxiliary coordinate pair for
 * drawing this edge as a curve
 */
public void addInterpolationPoint(Position pos)
{    if (!(interpolationPoints != null))
        interpolationPoints = new Vector<Position>();
    this.interpolationPoints.add(pos);
}

/**
 * Adds a GenericVertex v to the attached vertices graph of this edge,
 * if v is attachable to an edge according to the ontology.
 * @param v the vertex to be attached to this edge
 *             (in the subgraph for attached vertices)
 * @return true if successful
 */
public boolean addAttachableVertex(GenericVertex v)
{     if (OWLIntegrityChecker.isAttachableToEdge((OWLClass)
        (OWLIntegrityChecker.getOWLEntity(v.getDomainConcept())) ))
    {
        if (!(attachedVerticesGraph != null)) initAttachVerticesGraph();
        if (!v.getAttachedToEdges().contains(this))
                v.addAttachedToEdge(this);
        if (v.getGraph() != null)
        {
            GenericVertex NewVertex =
                (GenericVertex)v.copy(this.attachedVerticesGraph);
            NewVertex.concatToId(GenericVertex.COPY_VERTEX);
            System.out.println("Attached pointer " + NewVertex
                            + " added to " + this);
        }
        else
        {
            String childPrefix = "";
            if (this.attachedVerticesGraph.getPrefix() != "")
                childPrefix = this.attachedVerticesGraph.getPrefix() +
```

```
                        "." + this.attachedVerticesGraph.getId();
                else childPrefix = this.attachedVerticesGraph.getId();

                this.attachedVerticesGraph.addVertex(v);
                v.setPrefix(childPrefix);
            }
            return true;
        }
        return false;
    }

    /**
     * Delegate method to @see Attributes.
     * @param attributePattern an attribute by its pattern representation
     * @return its value
     */
    public Object getAttributeValue(String attributePattern)
    { return Attributes.getAttributeValue(this, attributePattern); }

    /**
     * Delegate method to @see Attributes.
     * @param attributePattern an attribute by its pattern representation
       * @param attributeValue its value
     */
    public void addAttribute(String attributePattern, Object attributeValue)
    { Attributes.addAttribute(this, attributePattern, attributeValue); }

    /**
     * @return the highest level graph this vertex is contained in
     */
    public GenericGraph getSuperGraph()
    {    GenericVertex Cluster = this.getGraph().getClusterVertex();
        if (Cluster != null) return Cluster.getSuperGraph();

        IGenericEdge Aggregate = this.getGraph().getAggregateEdge();
        if (Aggregate != null) return Aggregate.getSuperGraph();

        return this.getGraph();
    }
}
```

B.1.3 RegionHierarchy.java

```
package de.lmu.ifi.pms.transroute.graph;

import de.lmu.ifi.pms.transroute.domainmodel.*;
..

/**
 * Contains a tree-like graph of the earth (root), all continents (next
```

```
 * level), their countries and within each country all its cities. ..
 * @author Edgar-Philipp Stoffel
 */
public class RegionHierarchy
{
    //constant Strings representing respective classes in ontologies
    private final String RH = OWLIntegrityChecker.RH;
    ..

    private GenericGraph RegionHierarchyGraph;
    //tree-like structure: vertices = regions (continents, countries, cities)
    private OntologyManager OntoMngr = TransRoute.getOntoMngr();

    public RegionHierarchy()
    {
        RegionHierarchyGraph = GraphFactory.create(RH, 0, RH);
        GenericVertex Earth = VertexFactory.create(EAR, EAR);
        RegionHierarchyGraph.addVertex(Earth);

        for(OWLIndividual Individual :
                OntoMngr.getClassesIndividuals(OntoMngr.getClass(CONT)))
        { // go through all continents in ontology
            GenericVertex Continent = RegionHierarchyGraph.getVertexById(
                                         OntoMngr.getName(Individual));
            if (!(Continent != null))
            {    //vertex representing continent with this
                //id does not exist yet, so create it
                Continent = VertexFactory.create(OntoMngr.getName(Individual),
                                CONT);     //continent represented as vertex
                Continent = RegionHierarchyGraph.addVertex(Continent);
                IGenericEdge HasContinent = EdgeFactory.create(
                    OntoMngr.getName(Individual) + "Connect", IR, Earth,
                                        Continent, false);
                RegionHierarchyGraph.addEdge(HasContinent);
                //containment relation represented as edge
            }

            for (OWLIndividual adjContinentInd :
                OntoMngr.getIndividualsFromProperty(Individual,
                                OWLIntegrityChecker.ADJCONT))
            {    //go through all its adjacent continents
                GenericVertex adjContinent =
                    RegionHierarchyGraph.getVertexById(
                                OntoMngr.getName(adjContinentInd));
                if (!(adjContinent != null))
                {     // vertex representing adjacent continent with this
                    // id does not exists yet, so create it
                    adjContinent = VertexFactory.create(OntoMngr.getName(
                                             adjContinentInd), CONT);
                    adjContinent =
                        RegionHierarchyGraph.addVertex(adjContinent);
                    //adjacent continent represented as vertex
                    IGenericEdge HasContinent = EdgeFactory.create(
```

```
                        OntoMngr.getName(adjContinentInd) + "Connect", IR,
                        Earth, adjContinent, false);
                    RegionHierarchyGraph.addEdge(HasContinent);
                    //containment relation represented as edge
                }
                IGenericEdge InterContinent =
                    EdgeFactory.create(OntoMngr.getName(adjContinentInd) +
                     "InterConnect", ADJCONT, Continent, adjContinent, false);
                RegionHierarchyGraph.addEdge(InterContinent);
                //neighbourhood relation represented as edge
            }
            ..
        }
    }
    ..
}
```

B.1.4 HamiltonSPUnordered.java

```
package de.lmu.ifi.pms.transroute.graph.algorithms;

import de.lmu.ifi.pms.transroute.graph.filters.VertexInBoundingBoxPredicate;
..

/**
 * Offers methods for retrieving shortest paths by changing the
 * order of the intermediate vertices.
 * @author Edgar-Philipp Stoffel
 *
 */
public class HamiltonSPUnordered  implements IHamiltonShortestPath
{
    private Vector<GenericVertex> OrderOfVertices;
    private DistanceMetric Manhattan = new ManhattanMetric();

    private Vector<IGenericEdge> getPathRec(ShortestPathImpl SP,
        Vector<IGenericEdge> Path, GenericVertex vFrom, GenericVertex vTo,
        Vector<GenericVertex> Todo, String costAttribute)
    {
        switch (Todo.size())
        {
            case 0:
                Path.addAll(SP.getPath(vFrom, vTo, costAttribute));
                return Path;
            case 1:
                Path.addAll(SP.getPath(vFrom, Todo.get(0), costAttribute));
                Path.addAll(SP.getPath(Todo.get(0), vTo, costAttribute));
                OrderOfVertices.add(Todo.get(0));
                return Path;
```

```
            default:
                double currCost = 0, minCost = Double.MAX_VALUE;
                int count = 0, pos = 0;

                VertexInBoundingBoxPredicate inPred = new
                    VertexInBoundingBoxPredicate(new BoundingBox(
                        vFrom.getLocation(), vTo.getLocation()));
                Vector<GenericVertex> Outliers = new Vector<GenericVertex>();
                for (GenericVertex v : Todo)
                { if (!inPred.evaluateVertex(v)) Outliers.add(v); }

                if (Outliers.size() > 0)
                {    for (GenericVertex v : Outliers)
                    {    currCost = getDistance(Manhattan, vFrom, v);
                        if (currCost < minCost)
                        { minCost = currCost; pos = count; }
                        count++;
                    }
                }
                else
                {    for (GenericVertex v : Todo)
                    {    currCost = getDistance(Manhattan, vFrom, v);
                        if (currCost < minCost)
                        { minCost = currCost; pos = count; }
                        count++;
                    }
                }
                GenericVertex NearestFrom = Todo.get(pos);
                Path.addAll(SP.getPath(vFrom, NearestFrom, costAttribute));
                OrderOfVertices.add(NearestFrom);
                Todo.remove(NearestFrom);
                return getPathRec(SP, Path, NearestFrom, vTo,
                                            Todo, costAttribute);
        }
    }

    ..

    public Vector<IGenericEdge> getPath(GenericVertex vFrom, GenericVertex vTo,
                        Vector<GenericVertex> vInter, String costAttribute)
    {
        OrderOfVertices = new Vector<GenericVertex>();
        return getPathRec(new ShortestPathImpl(), new Vector<IGenericEdge>(),
                                vFrom, vTo, vInter, costAttribute);
    }
}
```

B.1.5 VertexInBoundingBoxPredicate.java

```
package de.lmu.ifi.pms.transroute.graph.filters;

import edu.uci.ics.jung.graph.ArchetypeVertex;
import edu.uci.ics.jung.graph.predicates.VertexPredicate;
import de.lmu.ifi.pms.transroute.graph.generic.GenericVertex;
import de.lmu.ifi.pms.transroute.graph.generic.GenericGraph;
import de.lmu.ifi.pms.transroute.geometry.*;

/**
 * Checks if a vertex is within the boundaries of a bounding box
 * @author Edgar-Philipp Stoffel
 * @see de.lmu.ifi.pms.transroute.geometry.BoundingBox
 * @see de.lmu.ifi.pms.transroute.graph.generic.GenericVertex
 */
public class VertexInBoundingBoxPredicate extends VertexPredicate  {

    private BoundingBox boundBox;

    public VertexInBoundingBoxPredicate(GenericGraph g)
    { this.boundBox = g.getBoundingBox(); }

    public VertexInBoundingBoxPredicate(BoundingBox boundBox)
    { this.boundBox = boundBox; }

    public boolean evaluateVertex(ArchetypeVertex v)
    {
        if (v instanceof GenericVertex)
        {
            GenericVertex gv = (GenericVertex)v;
            Position pos = gv.getLocation();
            if (pos != null && boundBox != null)
            {
                double x = pos.x(), y = pos.y();
                double minX = boundBox.min().x(), minY = boundBox.min().y(),
                       maxX = boundBox.max().x(), maxY = boundBox.max().y();

                return (minX <= x && x <= maxX && minY <= y && y <= maxY);
            }
            else return false;
        }
        else return false;
    }
}
```

B.1.6 OntologyManager.java

```
package de.lmu.ifi.pms.transroute.domainmodel;
```

```
import java.net.URI;
import org.semanticweb.owl.impl.model.*;
..

/**
 * Realised as a singleton, the OntologyManager class encapsulates access
 * to an OWL Ontology, which represents domain knowledge. The connection
 * and in-memory representation of an OWL Ontology is provided by the OWLAPI
 * framework. OntologyManager offers comfortable higher-level access to the
 * domain model ontology.
 * @author  Edgar-Philipp Stoffel
 */
public class OntologyManager implements IDomainModelConstants{
    ..
    public final String URI_PHYS =
        ConfigReader.instance().getOntologyURIPhysical();
    private OWLOntology Ontology;
    private Vector<OWLClass> ValidEdges, ValidVertices,
                             ValidAttachables, ValidGraphs;
    ..

    /**
     * Returns the corresponding OWLClass given its String representation.
     * @param className a String representation of an OWLClass
     * @return its corresponding OWLClass, if it exists
     */
    public OWLClass getClass(String className)
    {    OWLClass owlClass = null;
        try
        {
            owlClass = Ontology.getClass(new URI(URI_LOG + SHARP + className));
        }
        catch (OWLException OWLException)
        {System.out.println("OWL Exception occured!");
         OWLException.printStackTrace(); System.exit(-1);}
        catch(URISyntaxException URISyntaxException)
        {System.out.println("URISyntax Exception occured!"); System.exit(-1);}

        return owlClass;
    }

    public Vector<OWLIndividual> getClassesIndividuals(OWLClass owlClass)
    {
        Vector<OWLIndividual> Individuals = new Vector<OWLIndividual>();
        try {
            for(Object Individual : Ontology.getIndividuals())
            {
                OWLIndividual OWLIndividual = (OWLIndividual)Individual;
                OWLClass indivClass = getIndividualsClass(OWLIndividual);
                if (owlClass.equals(indivClass)) Individuals.add(OWLIndividual);
            }
        }
```

```
        catch (OWLException OWLException)
        {System.out.println("OWL Exception occured!");
         OWLException.printStackTrace(); System.exit(-1);}

        return Individuals;
    }

    /**
     * gets not only the direct superclasses of an OWLClass, but its entire
     * transitive hull of the superclass relation, i.e. the superclasses of
     * its superclasses and so forth.
     * @param Class an OWLClass
     * @return all its transitive superclasses, including the class itself
     */
    public Vector<OWLClass> getSuperClassesTransitive(OWLClass Class)
    {
        Vector<OWLClass> transSupers = new Vector<OWLClass>();
        transSupers.add(Class);
        try
        {
            for(Object s: Class.getSuperClasses(Ontology))
            {
                if (s instanceof OWLClass)
                {
                    OWLClass superClass = (OWLClass)s;
                    transSupers.add(superClass);
                    if (superClass.getSuperClasses(Ontology).size() > 0)
                      transSupers.addAll(getSuperClassesTransitive(superClass));
                }
            }
        }
        catch (OWLException OWLException)
        {System.out.println("OWL Exception occured!");
        OWLException.printStackTrace(); System.exit(-1);}

        return transSupers;
    }

    /**
     * Checks whether a class has the specified ancestor
     * @param child the child class
     * @param superC its ancestor to be searched for
     */
    public boolean hasSuperClassTransitive(OWLClass child, OWLClass superC)
    { return this.getSuperClassesTransitive(child).contains(superC); }
}
```

B.1.7 GXLWriter.java

```
package de.lmu.ifi.pms.transroute.io;

import java.io.IOException;
import net.sourceforge.gxl.*;
..

/**
 * de.lmu.ifi.pms.transroute.io.GXLReader.java
 * @author  Edgar-Philipp Stoffel
 */
public class GXLWriter implements IWriter, IGXLConstants {

    private static GXLWriter theInstance = new GXLWriter(); //singleton instance
    private Hashtable<GenericVertex, GXLNode> NodeTable;
    private int counter = 0;
    private StopTimer timer = StopTimer.instance();

    /* private constructor */
    private GXLWriter() {}

    /**
     * getter for single instance (singleton)
     * @return the GXL mapper
     */
    public static GXLWriter instance()
    {
        return theInstance;
    }

    public void addGXLFieldAttrs
                (GXLElement gxlElem, IHierarchicElement GenericElem)
    {
        gxlElem.add(new GXLAttr(DC,
                            new GXLString(GenericElem.getDomainConcept())));
        gxlElem.add(new GXLAttr(ID, new GXLString(GenericElem.getId())));
        gxlElem.add(new GXLAttr(PRE, new GXLString(GenericElem.getPrefix())));
        gxlElem.add(new GXLAttr(LVL, new GXLInt(GenericElem.getLevel())));

        Attributes.addAttributesToGXLElement(GenericElem, gxlElem);
    }

    public GXLSeq toGXL(Position pos)
    {     GXLSeq seq = new GXLSeq();
        seq.add(new GXLFloat((float)pos.x()));
        seq.add(new GXLFloat((float)pos.y()));
        if (pos instanceof Position3D)
            seq.add(new GXLFloat((float)((Position3D)pos).z()));
        return seq;
    }
```

```
public GXLAttr toGXL(BoundingBox bB)
{
    GXLBag bag = new GXLBag();
    bag.add(toGXL(bB.min())); bag.add(toGXL(bB.max()));
    return new GXLAttr(BB, bag);
}

public void addNestedGXLGraph(GXLElement gxlE,
                        GenericGraph g, String IdSuffix)
{
    GXLGraph gxlNestedGraph = null;
    /* convert non-empty inner graph into the GXL struture */
    if (g != null && g.getVertices().size() > 0)
    {    gxlNestedGraph = toGXL(g);
        gxlNestedGraph.setID(gxlNestedGraph.getID() + IdSuffix); }
    if (gxlNestedGraph != null) gxlE.add(gxlNestedGraph);
}

public GXLNode toGXL(GenericVertex Vertex)
{
    GXLNode gxlNode = new GXLNode("gxl" + counter++);
    addGXLFieldAttrs(gxlNode, Vertex);

    GenericGraph nestG = Vertex.getNestedGraph();
    addNestedGXLGraph(gxlNode, nestG, "");

    Vector<IGenericEdge> attchEs = Vertex.getAttachedToEdges();
    if (attchEs != null && attchEs.size() > 0)
    {
        GXLSeq seq = new GXLSeq();
        for(IGenericEdge ge : Vertex.getAttachedToEdges())
        {    seq.add(new GXLString(ge.getId())); }
        gxlNode.add(new GXLAttr(A2E, seq));
    }

    String Region = Vertex.getRegion(),
        NetworkType = Vertex.getNetworkType();
    if (Region != null) gxlNode.add(new GXLAttr(R,
                                  new GXLString(Region)));
    if (NetworkType != null)
        gxlNode.add(new GXLAttr(NT, new GXLString(NetworkType)));

    Position location = Vertex.getLocation();
    String posType = POS;
    if (location instanceof Position3D) posType += D3;
    if (location != null) gxlNode.add(new GXLAttr(
                                posType, toGXL(location)));
    return gxlNode;
}

public GXLEdge toGXL(IGenericEdge Edge)
{
    GenericVertex Src = Edge.getStartsAtVertex();
```

```
        GenericVertex Dest = Edge.getEndsAtVertex();

        GXLNode GXLDest = NodeTable.get(Dest);
        GXLNode GXLSrc = NodeTable.get(Src);
        GXLEdge gxlEdge = new GXLEdge(GXLSrc, GXLDest);
        addGXLFieldAttrs(gxlEdge, Edge);

        if    (Edge instanceof GenericEdgeUndir)
            gxlEdge.add(new GXLAttr(DIR, new GXLBool(false)));
        else if (Edge instanceof GenericEdgeDir)
            gxlEdge.add(new GXLAttr(DIR, new GXLBool(true)));

        addNestedGXLGraph(gxlEdge, Edge.getAggregatedEdgesGraph(), AGGR);
        addNestedGXLGraph(gxlEdge, Edge.getAttachedVerticesGraph(), ATV);

        Vector<Position> interpols = Edge.getInterpolationPoints();
        if (interpols != null && interpols.size() > 0)
        {
            GXLBag bag = new GXLBag();
            for(Position pos : interpols)
            {    bag.add(toGXL(pos)); }
            gxlEdge.add(new GXLAttr(IP, bag));
        }

        gxlEdge.setID("gxl" + counter++);
        return gxlEdge;
    }

    public GXLGraph toGXL(GenericGraph Graph)
    {
        GXLGraph _gxlGraph = new GXLGraph("gxl" + counter++);
        addGXLFieldAttrs(_gxlGraph, Graph);

        BoundingBox bB = Graph.getBoundingBox();
        if (bB != null) _gxlGraph.add(toGXL(bB));

        String Region = Graph.getRegion(),
               NetworkType = Graph.getNetworkType();
        if (Region != null) _gxlGraph.add(new GXLAttr(
                                     R, new GXLString(Region)));
        if (NetworkType != null)
            _gxlGraph.add(new GXLAttr(NT, new GXLString(NetworkType)));

        for (Object v: Graph.getVertices())
        {
            GenericVertex Vertex = (GenericVertex)v;
            GXLNode gxlNode = toGXL(Vertex);
            _gxlGraph.add(gxlNode);
            NodeTable.put(Vertex, gxlNode);
        }

        for (Object e: Graph.getEdges())
        {
```

```
            IGenericEdge Edge = (IGenericEdge)e;
            GXLEdge gxlEdge = toGXL(Edge);
            _gxlGraph.add(gxlEdge);
        }
        return _gxlGraph;
    }

    public void save(GenericGraph Graph, String fileName)
    {
        timer.start();
        NodeTable = new Hashtable<GenericVertex, GXLNode>();

        GXLDocument gxlDocument = new GXLDocument(); //    Create document
        gxlDocument.getDocumentElement().add(toGXL(Graph));

        try {         //   Write the document to file
            gxlDocument.write(new File(fileName));
            System.out.println(timer.stop() + " [GXLWriter] " +
                                             fileName  + " Done.");
        }
        catch (IOException ioe) {
            System.out.println("Error while writing to file: " + ioe);
            timer.stop();
        }
    }
}
```

B.2 GXL Examples

B.2.1 GXL-1.0.dtd

```
<!-- Extensions -->
<!ENTITY % gxl-extension "">
<!ENTITY % graph-extension "">
<!ENTITY % node-extension "">
<!ENTITY % edge-extension "">
<!ENTITY % rel-extension "">
<!ENTITY % value-extension "">
<!ENTITY % relend-extension "">
<!ENTITY % gxl-attr-extension "">
<!ENTITY % graph-attr-extension "">
<!ENTITY % node-attr-extension "">
<!ENTITY % edge-attr-extension "">
<!ENTITY % rel-attr-extension "">
<!ENTITY % relend-attr-extension "">
<!-- Attribute values -->
<!ENTITY % val "
           locator |
           bool    |
```

```
           int     |
           float   |
           string  |
           enum    |
           seq     |
           set     |
           bag     |
           tup
           % value-extension;">
<!-- gxl -->
<!ELEMENT gxl (graph* % gxl-extension;) >
<!ATTLIST gxl
    xmlns:xlink CDATA # FIXED "http://www.w3.org/1999/xlink"
    % gxl-attr-extension;
>
<!-- type -->
<!ELEMENT type EMPTY>
<!ATTLIST type
    xlink:type (simple) # FIXED "simple"
    xlink:href CDATA # REQUIRED
>
<!-- graph -->
<!ELEMENT graph (type? , attr* , ( node | edge | rel )* % graph-extension;) >
<!ATTLIST graph
    id ID # REQUIRED
    role NMTOKEN # IMPLIED
    edgeids (true | false) "false"
    hypergraph (true | false) "false"
    edgemode (directed | undirected | defaultdirected
                    | defaultundirected) "directed"
    % graph-attr-extension;
>
<!-- node -->
<!ELEMENT node (type? , attr*, graph* % node-extension;) >
<!ATTLIST node
    id ID # REQUIRED
    % node-attr-extension;
>
<!-- edge -->
<!ELEMENT edge (type?, attr*, graph* % edge-extension;) >
<!ATTLIST edge
    id ID # IMPLIED
    from IDREF # REQUIRED
    to IDREF # REQUIRED
    fromorder CDATA # IMPLIED
    toorder CDATA # IMPLIED
    isdirected (true | false) # IMPLIED
    % edge-attr-extension;
>
<!-- rel -->
<!ELEMENT rel (type? , attr*, graph*, relend* % rel-extension;) >
<!ATTLIST rel
    id ID # IMPLIED
```

```
    isdirected (true | false) # IMPLIED
    % rel-attr-extension;
>
<!-- relend -->
<!ELEMENT relend (attr* % relend-extension;) >
<!ATTLIST relend
    target IDREF # REQUIRED
    role NMTOKEN # IMPLIED
    direction (in | out | none) # IMPLIED
    startorder CDATA # IMPLIED
    endorder CDATA # IMPLIED
    % relend-attr-extension;
>
<!-- attr -->
<!ELEMENT attr (attr*, (% val;))>
<!ATTLIST attr
    id ID # IMPLIED
    name NMTOKEN # REQUIRED
    kind NMTOKEN # IMPLIED
>
<!-- locator -->
<!ELEMENT locator EMPTY>
<!ATTLIST locator
    xlink:type (simple) # FIXED "simple"
    xlink:href CDATA # REQUIRED
>
<!-- atomic values -->
<!ELEMENT bool (# PCDATA)>
<!ELEMENT int (# PCDATA)>
<!ELEMENT float (# PCDATA)>
<!ELEMENT string (# PCDATA)>
<!-- enumeration value -->
<!ELEMENT enum (# PCDATA)>
<!-- composite values -->
<!ELEMENT seq (% val;)*>
<!ELEMENT set (% val;)*>
<!ELEMENT bag (% val;)*>
<!ELEMENT tup (% val;)*>
```

B.2.2 Tram17.gxl Representing Line 17 in Munich

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
<gxl>
    <graph id="gxl0">
        <attr name="DomainConcept">
            <string>Transport_Network</string>
        </attr>
        <attr name="id">
            <string>MunichNetwork</string>
        </attr>
```

```
        <attr name="prefix">
            <string/>
        </attr>
        <attr name="level">
            <int>0</int>
        </attr>
        <attr name="BoundingBox">
            <bag>
                <seq>
                    <float>14768.0</float>
                    <float>10659.0</float>
                </seq>
                <seq>
                    <float>18345.0</float>
                    <float>13061.0</float>
                </seq>
            </bag>
        </attr>
        <node id="gxl1">
            <attr name="DomainConcept">
                <string>Stop_Point</string>
            </attr>
            <attr name="id">
                <string>Tivolistrasse</string>
            </attr>
            <attr name="prefix">
                <string>MunichNetwork</string>
            </attr>
            <attr name="level">
                <int>1</int>
            </attr>
            <attr name="Position">
                <seq>
                    <float>17017.0</float>
                    <float>10942.0</float>
                </seq>
            </attr>
        </node>

    ...
        <edge from="gxl12" id="gxl25" to="gxl2">
            <attr name="DomainConcept">
                <string>Route_Link</string>
            </attr>
            <attr name="id">
                <string>Effnerplatz-Effnerplatz2Link</string>
            </attr>
            <attr name="prefix">
                <string>MunichNetwork</string>
            </attr>
            <attr name="level">
                <int>1</int>
            </attr>
```

```
            <attr name="directed">
                <bool>true</bool>
            </attr>
            <attr name="interpolationPoints">
                <bag>
                    <seq>
                        <float>-11.0</float>
                        <float>-13.0</float>
                    </seq>
                    <seq>
                        <float>-11.0</float>
                        <float>-8.0</float>
                    </seq>
                    <seq>
                        <float>-9.0</float>
                        <float>-4.0</float>
                    </seq>
                </bag>
            </attr>
        </edge>
    ..
      </graph>
</gxl>
```

B.2.3 A Nested Graph Inside an Edge in GXL

```
    <edge from="gxl15" id="gxl34" to="gxl18">
            <attr name="DomainConcept">
                <string>Road_Element</string>
            </attr>
            <attr name="id">
                <string>Leopoldstr</string>
            </attr>
            <attr name="prefix">
                <string>munichStreets</string>
            </attr>
            <attr name="level">
                <int>1</int>
            </attr>
            <attr name="directed">
                <bool>true</bool>
            </attr>
            <graph id="gxl30AttachedVertices">
                <attr name="DomainConcept">
                    <string>Road_Element</string>
                </attr>
                <attr name="id">
                    <string>LeopoldstrAttachedVertices</string>
                </attr>
                <attr name="prefix">
                    <string>munichStreets</string>
```

```
        </attr>
        <attr name="level">
            <int>1</int>
        </attr>
        <node id="gxl31">
            <attr name="DomainConcept">
                <string>University</string>
            </attr>
            <attr name="id">
                <string>LMU</string>
            </attr>
            <attr name="prefix">
                <string>munichStreets.LeopoldstrAttachedVertices</string>
            </attr>
            <attr name="level">
                <int>2</int>
            </attr>
            <attr name="attachedToEdges">
                <seq>
                    <string>Leopoldstr</string>
                    <string>ed23</string>
                </seq>
            </attr>
        </node>
        <node id="gxl32">
            <attr name="DomainConcept">
                <string>Junction</string>
            </attr>
            <attr name="id">
                <string>InMuenchnerFreiheit--Pointer--</string>
            </attr>
            <attr name="prefix">
                <string>munichStreets.MuenchnerFreiheitNested</string>
            </attr>
            <attr name="level">
                <int>2</int>
            </attr>
        </node>
        <edge from="gxl31" id="gxl33" to="gxl32">
            <attr name="DomainConcept">
                <string>Road_Element</string>
            </attr>
            <attr name="id">
                <string>lmu-mf</string>
            </attr>
            <attr name="prefix">
                <string>munichStreets.LeopoldstrAttachedVertices</string>
            </attr>
            <attr name="level">
                <int>2</int>
            </attr>
            <attr name="directed">
                <bool>true</bool>
```

```
                </attr>
            </edge>
        </graph>
    </edge>
```

B.2.4 A Timetable Attribute in GXL

```
<attr name="ed23.timetable">
    <seq>
        <string>S4</string>
        <string>Mon</string>
        <string>09:59:00</string>
        <string>Tue</string>
        <string>10:15:00</string>
        <string>10:30:00</string>
        <string>10:45:00</string>
        <string>11:00:00</string>
        <string>11:15:00</string>
        <string>11:30:00</string>
        <string>11:45:00</string>
        <string>12:00:00</string>
        <string>12:15:00</string>
        <string>12:30:00</string>
        <string>12:45:00</string>
        <string>13:00:00</string>
        <string>Wed</string>
        <string>09:50:04</string>
        <string>09:59:00</string>
        <string>10:00:00</string>
        <string>21:44:00</string>
        <string>23:02:00</string>
        <string>Thu</string>
        <string>21:00:00</string>
        <string>Fri</string>
        <string>Sat</string>
        <string>10:15:00</string>
        <string>10:18:00</string>
        <string>Sun</string>
        <string>09:00:00</string>
    </seq>
</attr>
```

Bibliography

[1] EARTH AND MOON VIEWER. http://www.fourmilab.ch/earthview/vplanet.html.

[2] THE ECLIPSE PROJECT. http://www.eclipse.org/.

[3] FALK ROUTENPLANER. http://www.falk.de/routenplaner/controller_rp.jsp.

[4] GOOGLE EARTH. http://earth.google.com/.

[5] GOOGLE MAPS. http://maps.google.com/.

[6] GRAPH ALGORITHMS. http://users.informatik.uni-halle.de/$\sim$jopsi/dinf204/chap10.shtml.

[7] GRAPHML: THE GRAPHML FILE FORMAT. http://graphml.graphdrawing.org.

[8] GXL: GRAPH EXCHANGE LANGUAGE. http://www.gupro.de/GXL/.

[9] MAP24. http://www.de.map24.com/.

[10] MAPPY. http://www.mappy.org.

[11] OGC GML: A MARKUP LANGUAGE FOR GEOGRAPHY. http://opengis.net/gml/.

[12] OGC GML VERSION 3.00 OPENGIS IMPLEMENTATION SPECIFICATION. http://www.opengis.org/docs/02-023r4.pdf.

[13] OWL API PROJECT. http://sourceforge.net/projects/owlapi.

[14] PRACTICAL COURSE ”STREAMED TRAFFIC AND TRAVEL INFORMATION”. http://www.pms.ifi.lmu.de/lehre/praktikum/traveltraffic/04ws05/.

[15] THE PROTÉGÉ ONTOLOGY EDITOR AND KNOWLEDGE ACQUISITION SYSTEM. http://protege.stanford.edu/.

[16] REWERSE: REASONING ON THE WEB WITH RULES AND SEMANTICS. http://rewerse.net/.

[17] SAX: SIMPLE API FOR XML. http://www.saxproject.org/.

[18] SOURCEFORGE.NET GML4J PROJECT. https://sourceforge.net/projects/gml4j/.

[19] SOURCEFORGE.NET GXL API: GRAPH EXCHANGE LANGUAGE PROJECT. http://gxl.sourceforge.net/index.html.

[20] SOURCEFORGE.NET JUNG API: JAVA UNIVERSAL NETWORK/GRAPH FRAMEWORK. http://jung.sourceforge.net/doc/api/index.html.

[21] SOURCEFORGE.NET JUNG: JAVA UNIVERSAL NETWORK/GRAPH FRAMEWORK. http://jung.sourceforge.net.

[22] SUN JAVA PROGRAMMING LANGUGAGE. http://java.sun.com/.

[23] TERRA SERVER. http://www.terraserver.com.

[24] TRAFFIC MESSAGE CHANNEL FORUM. http://www.tmcforum.com/.

[25] TRANSPORTATION GEOGRAPHY ON THE WEB. http://people.hofstra.edu/geotrans/.

[26] VIA MICHELIN. http://www.viamichelin.com/viamichelin/deu/tpl/hme/MaHomePage.htm.

[27] W3C OWL: WEB ONTOLOGY LANGUAGE. http://www.w3.org/TR/owl-features/.

[28] W3C RECOMMENDATION DOM: DOCUMENT OBJECT MODEL. http://www.w3.org/DOM/.

[29] W3C RECOMMENDATION XML: EXTENSIBLE MARKUP LANGUAGE. http://www.w3.org/XML/.

[30] W3C RECOMMENDATION XSLT: XSL TRANSFORMATION. http://www.w3.org/TR/xslt.

[31] WIKIPEDIA: HAMILTONIAN PATH. http://en.wikipedia.org/wiki/Hamiltonian_cycle.

[32] WIKIPEDIA: NETWORK TOPOLOGY. http://en.wikipedia.org/wiki/Network_topology.

[33] WIKIPEDIA: ROUTING. http://en.wikipedia.org/wiki/Routing.

[34] WIKIPEDIA: TRAVELLING SALESMAN. http://en.wikipedia.org/wiki/Travelling_salesman.

[35] Ravindra Ahuja, Thomas Magnanti, and James Orlin: NETWORK FLOWS. Prentice-Hall, ISBN 0-13-617549-X, 1993. 1-8, 15, 32pp., 48, 89, 93pp., 294pp., 740pp.

[36] Chris Barrett, Riko Jacob, and Madhav V. Marathe: FORMAL LANGUAGE CONSTRAINED PATH PROBLEMS. In SCANDINAVIAN WORKSHOP ON ALGORITHM THEORY, pages 234–245, 1998. citeseer.ist.psu.edu/barrett98formal.html.

[37] Sean Bechhofer, Phillip Lord, and Raphael Volz: COOKING THE SEMANTIC WEB WITH THE OWL API., 2003.

[38] Lowell Beineke and Robin Wilson: SELECTED TOPICS IN GRAPH THEORY. Academic Press, ISBN 0-12-086250-6, 1978. 15 pp.

[39] Brandon Bennett: MODAL LOGICS FOR QUALITATIVE SPATIAL RESONING, 1996. citeseer.ist.psu.edu/bennett96modal.html.

[40] Gilad Bracha: GENERICS IN THE JAVA PROGRAMMING LANGUAGE, 2004. java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[41] Ulrik Brandes, Jürgen Lerner, and Christian Pich: GXL TO GRAPHML AND VICE VERSA WITH XSLT. http://www.inf.uni-konstanz.de/algo/publications/blp-ggvvx-04.pdf.

[42] Ulrik Brandes *et al.*: TRAVEL PLANNING WITH SELF-MADE MAPS, 2001. citeseer.ist.psu.edu/brandes01travel.html.

[43] Martin Breunig: INTEGRATION OF SPATIAL INFORMATION FOR GEO-INFORMATION SYSTEMS. Springer, ISBN 3-540-60856-7, 1996. 1 pp.

[44] Friedhelm Buchholz and Barnhard Riedhofer: HIERARCHISCHE GRAPHEN ZUR KÜRZESTEN WEGESUCHE IN PLANAREN GRAPHEN. Technical report, University of Stuttgart, Faculty of Computer Science, 1997. `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-1997-13`.

[45] Giorgio Busatto: AN ABSTRACT MODEL OF HIERARCHICAL GRAPHS AND HIERARCHICAL GRAPH TRANSFORMATION. PhD thesis, University of Paderborn, 2002.

[46] Gilberto Camara *et al.*: TOWARDS A UNIFIED FRAMEWORK FOR SPATIAL DATA MODELS, 2000. `citeseer.ist.psu.edu/camara00towards.html`.

[47] Giulio Erberto Cantarella *et al.*: A DOUBLY DYNAMIC TRAFFIC ASSIGNMENT MODEL FOR PLANNING APPLICATIONS. In 14TH INTERNATIONAL SYMPOSIUM ON TRANSPORTATION AND TRAFFIC THEORY, Jerusalem, July 20-23 1999.

[48] Adrijana Car, Henny Mehner, and George Taylor: EXPERIMENTING WITH HIERARCHICAL WAYFINDING, 1999. `citeseer.ist.psu.edu/car99experimenting.html`.

[49] Vinod Chachra, Prabhakar Ghare, and James Moore: APPLICATIONS OF GRAPH THEORY ALGORITHMS. Elsevier North Holland, ISBN 0-444-00268-5, 1979. 79-108; 324-334.

[50] John Clark and Derek Holton: GRAPHENTHEORIE: GRUNDLAGEN UND ANWENDUNGEN. Spektrum Akademischer Verlag, ISBN 3-86025-331-X, 1994. 93pp., 251pp.

[51] Anthony G. Cohn *et al.*: QUALITATIVE SPATIAL REPRESENTATION AND REASONING WITH THE REGION CONNECTION CALCULUS. GeoInformatica, 1(3):275–316, 1997. `citeseer.ist.psu.edu/article/cohn97qualitative.html`.

[52] Thomas Cormen *et al.*: INTRODUCTION TO ALGORITHMS. MIT Press, ISBN 0-262-03293-7, 2001. 525 pp.

[53] Barbara Corona and Stephan Winter: NAVIGATION INFORMATION FOR PEDESTRIANS FROM CITY MAPS, 2001. `citeseer.ist.psu.edu/barbara01navigation.html`.

[54] Frank Dickmann: WEB-MAPPING UND WEB-GIS. Westermann, ISBN 3-14-160340-5, 2001. 13 pp.

[55] Reinhard Diestel: GRAPHENTHEORIE. Springer, ISBN 3-540-67656-2, 2000. `http://www.math.uni-hamburg.de/home/diestel/books/graphentheorie/`, 1-30 pp.

[56] Peter Eades and Qing-Wen Feng: MULTILEVEL VISUALIZATION OF CLUSTERED GRAPHS. In PROC. GRAPH DRAWING, GD, number 1190, pages 101–112, Berlin, Germany, 18–20 1996. Springer-Verlag. `citeseer.ist.psu.edu/eades97multilevel.html`.

[57] Rob van Essen and Volker Hiestermann: ISO-GDF - THE GEOGRAPHIC WORLD STANDARD FOR ITS, 2002. `http://roadmap.teleatlas.com/iso_gdf.pdf`.

[58] Alfonso Gerevini and Bernhard Nebel: QUALITATIVE SPATIO-TEMPORAL RESONING WITH RCC-8 AND ALLEN'S INTERVAL CALCULUS: COMPUTATIONAL COMPLEXITY, 2002. `citeseer.ist.psu.edu/gerevini02qualitative.html`.

[59] Alan Gibbons: ALGORITHMIC GRAPH THEORY. Cambridge University Press, ISBN 0-521-28881-9, 1985. 1-8, 39-42,57pp., 67pp., 96pp.

[60] Karen Zita Haigh, Jonathan Richard Shewchuk, and Manuela M. Veloso: EXPLOITING DOMAIN GEOMETRY IN ANALOGICAL ROUTE PLANNING. JETAI, 9(4):509–541, 1997. `citeseer.ist.psu.edu/article/haigh97exploiting.html`.

[61] Brian Hayes: GRAPH THEORY IN PRACTICE: PART I, 2000. `citeseer.ist.psu.edu/hayes00graph.html`.

[62] Bruce Hoppe and Eva Tardos: THE QUICKEST TRANSSHIPMENT PROBLEM. In SODA: ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS (A CONFERENCE ON THEORETICAL AND EXPERIMENTAL ANALYSIS OF DISCRETE ALGORITHMS), 1995. `citeseer.ist.psu.edu/article/hoppe95quickest.html`.

[63] Frank Ipflkofer: BASISONTOLOGIE UND ANWENDUNGSFRAMEWORK FÜR VISUALISIERUNG UND GEOSPATIAL REASONING, 2004. Master Thesis. University of Munich, Faculty of Computer Science.

[64] R. Jacob, M. Marathe, and K. Nagel: A COMPUTATIONAL STUDY OF ROUTING ALGORITHMS FOR REALISTIC TRANSPORTATION NETWORKS, 1998. `citeseer.ist.psu.edu/article/jacob98computational.html`.

[65] Pertti Järvinen: ON A ROUTING PROBLEM IN A GRAPH WITH TWO TYPES OF COSTS ASSOCIATED WITH THE EDGES, 1973. 1 pp.

[66] B. J. Kuipers: THE SPATIAL SEMANTIC HIERARCHY. Artificial Intelligence, 119:191–233, 2000. `citeseer.ist.psu.edu/kuipers00spatial.html`, http://www.cs.utexas.edu/users/qr/papers/Kuipers-aij-00.html.

[67] Ron Lake *et al.*: GML - GEOGRAPHY MARKUP LANGUAGE: FOUNDATION FOR THE GEO-WEB. Wiley, ISBN 0-470-87154-7, 2004. 1-174.

[68] Bing Liu: INTELLIGENT ROUTE FINDING: COMBINING KNOWLEDGE AND CASES AND AN EFFICIENT SEARCH ALGORITHM. In EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE, pages 380–384, 1996. `citeseer.ist.psu.edu/article/liu96intelligent.html`.

[69] J. McCormack: THE SPECIFICATION OF TRIP PLANNING SYSTEMS: REPORT ON THE DEVELOPMENT OF THE TRIP PLANNING SYSTEM PROTOTYPE, 1994. `citeseer.ist.psu.edu/mccormack94specification.html`.

[70] J. McCormack and S. Roberts: EXPLOITING OBJECT ORIENTED METHODS FOR MULTI-MODAL TRIP PLANNING SYSTEMS, 1994. `citeseer.ist.psu.edu/mccormack94exploiting.html`.

[71] Lorraine McGinty and University College Barry Smyth Smart Media Institute: SHARED EXPERIENCES IN PERSONALIZED ROUTE PLANNING, 2002. `citeseer.ist.psu.edu/624490.html`.

[72] Kai Nagel: TRAFFIC NETWORKS. 2002. `citeseer.ist.psu.edu/nagel02traffic.html`.

[73] Anna Nagurney: SUSTAINABLE TRANSPORTATION NETWORKS. Edward Elgar, ISBN 1-84064-357-9, 2000. 1 pp.

[74] Beng Chin Ooi: EFFICIENT QUERY PROCESSING IN GEOGRAPHIC INFORMATION SYSTEMS. Springer, ISBN 3-540-53474-1, 1990. 1 pp.

[75] Bin Ran and David Boyce: MODELING DYNAMIC TRANSPORTATION NETWORKS. Springer, ISBN 3-540-61139-8, 1996. 69pp.; 311pp., 330pp.

[76] Rob Raskin, Michael Pan, and Chris Mattmann: ENABLING SEMANTIC INTEROPERABILITY FOR EARTH SCIENCE DATA, 2004. `citeseer.ist.psu.edu/713722.html`.

[77] Jochen Renz: QUALITATIVE SPATIAL REASONING WITH TOPOLOGICAL INFORMATION. Springer, ISBN 3-540-43346-5, 2002. 13-15, 20p., 32-40.

[78] Urs-Jakob Rüetschi and Sabine Timpf: MODELLING WAYFINDING IN PUBLIC TRANSPORT: NETWORK SPACE AND SCENE SPACE, 2004. `www.geo.unizh.ch/~timpf/docs/Ruetschi_SC404.pdf`.

[79] Bernhard Riedhofer: HIERARCHISCHE STRASSENGRAPHEN, 1997. `elib.uni-stuttgart.de/opus/volltexte/1999/7/`, Master Thesis. University of Stuttgart, Faculty of Computer Science.

[80] Dr.-Ing. Martin Rose: MODELING OF FREEWAY TRAFFIC. In 10TH INTERNATIONAL CONFERENCE ON COMPUTING IN CIVIL ENGINEERING, Weimar, June 02-04 2004. `http://www.bauinf.uni-hannover.de/publikationen/ICCCEBPaperRose.pdf`.

[81] Jörg Schließer: ERSTELLUNG EINES SYSTEMS ZUR MODELLIERUNG UND ANALYSE HIERARCHISCHER GRAPHEN. Technical report, University of Stuttgart, Faculty of Computer Science, 1997. `elib.uni-stuttgart.de/opus/volltexte/1999/217/`.

[82] Robert Sedgewick: ALGORITHMEN. Addison-Wesley, ISBN 3-89319-301-4, 1995. 473 pp.

[83] S. Shekhar and A. Fetterer: PATH COMPUTATION IN ADVANCED TRAVELER INFORMATION SYSTEMS, 1996. `citeseer.ist.psu.edu/shekhar96path.html`.

[84] Christoph Tietz: ADAPTIVE PFADPLANUNG MIT VERFAHREN ZUR HEURISTISCHEN GRAPHENSUCHE. PhD thesis, University of Bielefeld, 1996.

[85] Sabine Timpf and Corinna Heye: COMPLEXITY OF ROUTES IN MULTI-MODAL WAYFINDING. Technical report, University of Zurich, Department of Geography, 2002. `http://e-collection.ethbib.ethz.ch/show?type=bericht\&nr=170`.

[86] Dorothea Wagner and Thomas Willhalm: GEOMETRIC SPEED-UP TECHNIQUES FOR FINDING SHORTEST PATHS IN LARGE SPARSE GRAPHS, 2003. `citeseer.ist.psu.edu/wagner03geometric.html`.

[87] REISEPLANUNG.

[88] Scott White *et al.*: ANALYSIS AND VISUALIZATION OF NETWORK DATA USING JUNG. `http://jung.sourceforge.net/doc/JUNG_journal.pdf`, Preprint.

[89] Chun−Hsin Wu *et al.*: AN ADVANCED TRAVELER INFORMATION SYSTEM WITH EMERGING NETWORK TECHNOLOGIES, 2003. http://citeseer.ist.psu.edu/wu03advanced.html.