Diplomarbeit

SIMULANT

Design and implementation of a simulation environment for ant algorithms in peer-to-peer networks

ausgeführt am

Institut für Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

unter Anleitung von
ao.Univ.Prof. Dr. Silvia Miksch
und
Projektass. DI Elke Michlmayr

von

Arno Pany
Strassergasse 43/1/4
1190 Wien
Matr.Nr. 9425000

Wien, im August 2006 ______________________

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, im August 2006 __________________________

Abstract

In the beginnings of the Internet, the connected host computers operated on peer-to-peer protocols. This mode of operation has been superseded by the well-known client-server architectures. In the past few years, peer-to-peer networks have again become a focus of public interest. The popularity of peer-to-peer applications is mostly due to file-sharing software. Less controversial applications of the peer-to-peer paradigm include the distribution of workload between host computers or aim at efficient network routing.

This thesis deals with a class of peer-to-peer algorithms which are inspired by the food gathering behaviour of certain species of ants. When searching for food, ants of a colony communicate indirectly by marking the path to the food source with chemical substances called pheromones. A marked path becomes more attractive for other ants, which will increase its pheromone concentration, and as a consequence a short path, which is used by most of the ants, emerges. This trail-laying and trail-following principle has been adapted to solve distributed problems from the domain of computer science. Agents travel the links of a network graph and mark their routes according to the quality of the solution described by the route. Ant-based algorithms have been applied with great success to tackle problems like the travelling salesman, graph coloring, network routing, and many others.

SemAnt is an ant-based resource location algorithm which has been designed to efficiently route queries through a network. The network's nodes hold repositories of documents which are annotated with search keys. The aim of *SemAnt* is to find a maximal number of documents matching the search keys specified in the query, while using the least possible network resources. When searching for documents, semantic similarities between the search keys may be exploited by *SemAnt*.

As of now, the performance of the algorithm has not been verified due to the lack of a suitable simulation environment. In the course of this work, a simulation environment dubbed *SimulAnt* has been implemented. The design and implementation issues of *SimulAnt* are presented in this thesis. Although *SimulAnt* has been developed for the *SemAnt* algorithm, it can be reconfigured to simulate other ant-algorithms with little effort.

The simulation results obtained with *SimulAnt* indicate that *SemAnt* is a suitable algorithm for the problem at hand, and encourage further research efforts for the algorithm. *SemAnt* outperforms standard reference algorithms like the k-random-walk strategy. The simulation runs give insight into the distribution of pheromones throughout the network, and show that *SemAnt* can be applied to both, maximizing the retrieved documents and minimizing the network load.

Kurzfassung

In den Anfängen des Internet kommunizierten die vernetzten Computer mittels Peer-to-Peer Protokollen. Diese Form der Vernetzung wurde größtenteils von den bekannten Client-Server Architekturen abgelöst. Aufgrund der Popularität von Filetauschbörsen hat das öffentliche Interesse an Peer-to-Peer Anwendungen in der letzten Zeit stark zugenommen. Weniger kontroversielle Anwendungen, die auf dem Peer-to-Peer Prinzip beruhen, ermöglichen die Verteilung von rechenintensiven Aufgaben auf mehrere Computer oder haben ein effektives Routing von Datenpaketen durch Netzwerke als Ziel.
Diese Diplomarbeit behandelt eine Klasse von Peer-to-Peer Algorithmen, welche das Nahrungssuchverhalten von Ameisen nachahmen. Auf der Suche nach Nahrung markieren Ameisen ihre Wege mit chemischen Duftstoffen (Pheromonen), und kommunizieren so indirekt miteinander. Die Duftspuren ziehen andere Ameisen an, und ein kurzer Pfad, der von vielen Ameisen verwendet wird, entsteht. Dieses "Spuren legen und verfolgen" kann adaptiert werden, um Probleme aus dem Bereich der Informatik zu lösen: Agenten bewegen sich auf den Pfaden zwischen Netzwerkknoten und markieren ihren Weg. Die Stärke der Markierung ist abhängig von der Qualität der Lösung, die durch den Weg beschrieben wird. Ameisenalgorithmen wurden mit großem Erfolg eingesetzt, um Probleme wie das Handelsreisendenproblem, Kartenfärben, Netzwerk-Routing, und viele andere zu lösen.
SemAnt ist ein Ameisenalgorithmus, der ein effizientes Finden von Resourcen in Netzwerken ermöglicht. Die Knoten im Netzwerk speichern Dokumente, welchen Suchkriterien zugeordnet sind. Suchanfragen sollen so durch das Netzwerk befördert werden, dass eine maximale Anzahl von Dokumenten bei einer minimalen Netzwerklast gefunden wird. Während der Suche können semantische Ähnlichkeiten der Anfragen von *SemAnt* ausgenützt werden. Die Leistungsfähigkeit von *SemAnt* konnte bis jetzt noch nicht getestet werden, da eine entsprechende Simulationsumgebung fehlte. Im Rahmen dieser Diplomarbeit wurde *SimulAnt*, eine Simulationsumgebung für *SemAnt*, implementiert. In dieser Arbeit werden Designüberlegungen und die Implementierung präsentiert. *SimulAnt* kann mit geringem Aufwand zur Simulation von anderen Ameisenalgorithmen als *SemAnt* adaptiert werden.
Simulationsergebnisse, die mittels *SimulAnt* generiert wurden, lassen erkennen, dass *SemAnt* dazu geeignet ist, die vorgegebene Problemstellung zu lösen, und ermutigen zu weiterer Arbeit am Algorithmus. Die Leistungsfähigkeit von *SemAnt* wird durch den Vergleich mit dem "k-random-walk" Algorithmus gezeigt. Die Auswertung der Simulationsläufe geben Einsichten in die Verteilung der Pheromone im Netzwerk und zeigen, dass *SemAnt* geeignet ist, sowohl die Anzahl der gefundenen Dokumente zu maximieren, als auch die Netzwerklast zu minimieren.

Danksagung

An dieser Stelle möchte ich mich bei meinen Eltern Alfred und Irene bedanken, die mir dieses Studium ermöglicht haben, und in Anbetracht der Geschwindigkeit meines Studienfortschritts nahezu endlose Geduld bewiesen haben.

Acknowledgements

Special thanks go to my supervisor Elke Michlmayr. I consider her to be the best supervisor any student could ask for. Without her constant motivation and advice, I'd probably still be working on this document.

Weiterer Dank gebührt...

auch allen Personen, die an der Umsetzung der deutschen Rechtschreibreform beteiligt waren. Die Reform war letztendlich ausschlaggebend dafür, diese Arbeit in englischer Sprache zu verfassen.

Contents

List of Figures

List of Tables

1 Introduction

In the last few years, peer-to-peer networks have received an increased public interest. This can be readily seen by the amount of different peer-to-peer applications that can be found on the Internet today. Peer-to-peer networks exist on top of established physical network architectures, and the participating nodes form networks of their own. For these reasons, peer-to-peer networks are also known as "overlay networks".

Peer-to-peer networks are used for a variety of purposes including – first and foremost – file sharing, which has earned them an ill reputation due to its capacity for copyright infringements. Filesharing, although very popular, is only one out of many applications for which peer-to-peer networks can be employed. Less controversial applications of peer-to-peer technologies are the distribution of workload and storage space between different host machines, or supporting collaboration between persons which log on to networks on an infrequent basis.

A main aspect of peer-to-peer networks is the location of resources which are available throughout the network. Location of resources is also referred to as *search* or *query routing*. Strategies to tackle the resource location problem range from uninformed search algorithms, like network flooding [53] and random walking [37], to sophisticated solutions which maintain local indices [16] to efficiently find resources on remote nodes.

Peer-to-peer structures cannot only be found in computing environments but also in social networks, economic principles or biological phenomena. Peer-to-peer technologies can therefore be used not only to replicate natural networks and exploit "algorithms" created by evolution, but also to simulate them and gain more insight into how these complex constructs function.

Ant algorithms are a class of algorithms that are inspired by the food gathering strategy of certain species of ants. Algorithms based on insects' "swarm intelligence" have successfully been implemented to solve optimization problems which can be mapped onto network graphs. The travelling salesman [17], or map coloring [14] are two examples out of many for the problems that have been successfully solved by employing ant algorithms. This thesis deals with employing ant algorithms for location of resources in peer-to-peer networks.

1.1 Contribution

SemAnt [40] is an ant-based algorithm intended for resource location in peer-to-peer environments. Resource location in this case means searching for documents which are distributed between the nodes of an overlay network. The search keys of the documents are taken from a hierarchical classification scheme (a taxonomy), which allows to infer semantic similarities between documents. These similarities between documents can be exploited during the search process. In addition, it is assumed that each node's owner is interested in a specific topic, and for this reason, owns many documents which are related to this topic. These patterns in the content distribution are another source for improvemnet of the search algorithm.
While the *SemAnt* algorithm has already been formulated, its performance could not be validated because it has never been tested in a real system or a simulated network. The goal of this thesis is to create a simulation environment for the *SemAnt* algorithm. The main challenges of creating the simulation environment are the following:

Efficient memory usage The lack of hardware resources, the overhead of (re-)configuring network nodes, and retrieving simulation data between simulation runs makes it unfeasible to test *SemAnt* in a real network. *SemAnt* has therefore to be tested in a virtual network on a single host computer. Since the network's size may well reach a few thousand nodes and millions of ants are expected to travel the network, it is imperative that memory resources are carefully managed, and that memory space is not taken up by objects which are no longer needed.

Synchronization A great number of ants will be travelling along the links of the *SemAnt* network. These ants update routing information located in the network's nodes. It has to be ensured that all ants arrive at the correct time to perform their updates so that subsequent ants may profit from the gathered information.

Storage of Results Huge amounts of data will be generated during the simulations of the *SemAnt* algorithm. Some efficient way of storing and evaluating the collected data has to be found.

Graphical User Interface Althought it would be possible to create the simulator with a command line interface, the commands to configure the simulations would be fairly complicated. This could be avoided by using configuration files, which have to be edited in between simulation runs. To alleviate these problems, it is desirable that the simulation environment has a clearly laid out graphical user interface, which is also capable of remembering the previous algorithm settings. This way only a few parameters, and not the whole list of available algorithm parameters have to be edited when tweaking the algorithm's performance.

Visualization of Results The results of the simulation runs need to be visualized in some way. Simple columns of numbers are inadequate to determine the quality of the simulation results. Between the simulation runs, the parameters of *SemAnt* will be tweaked and the effect of these changes on the performance needs to be evaluated. Therefore some metrics which specify the algorithm's performance have to be found and the results have to be displayed in some sort of chart to allow direct visual comparison of different simulation runs.

1.2 Organization of the thesis

Chapter 2 gives a general overview of peer-to-peer technologies. In the first part of this chapter, characteristics and possible architectures of peer-to-peer networks are discussed. The second part deals with network topologies which are typical for peer-to-peer systems.

Chapter 3 focuses on a specific group of algorithms which can be employed to solve distributed problems in peer-to-peer environments: ant-based algorithms. Basic concepts of ant-based algorithms are explained and two algorithms that build the foundation of the *SemAnt* algorithm are explored in more detail.

Chapter 4 presents the *SemAnt* algorithm, an ant-based algorithm designed to locate documents which are associated with metadata taken from controlled vocabularies or hierarchical classification schemes, such as taxonomies.

Chapter 5 discusses the design considerations of *SimulAnt*. *SimulAnt*, which is developed in the course of this diploma thesis, is a novel simulation environment for the *SemAnt* algorithm.

Chapter 6 deals with implementation issues of the *SimulAnt* platform. Software libraries and software development tools which are used to create *SimulAnt* are described.

Chapter 7 presents the results of *SemAnt* simulation runs. The k-random walk algorithm, which is used as a reference algorithm, is explained.

Chapter 8 contains the conclusions which can be drawn from the work with the *SemAnt* algorithm and proposes directions for future work on the algorithm and simulation environment.

2 Peer-to-Peer networks

This chapter gives an overview of peer-to-peer systems. In Section 2.1, a definition of the peer-to-peer concept can is given. Following this, key concepts, architectures, and a classification system for peer-to-peer systems are presented. The building blocks of a peer will then be explained. Section 2.2 deals with patterns which can be commonly observed when analyzing the properties of the connections between the peers in a network.

2.1 Peer-to-peer concepts

Although peer-to-peer technologies have become rather popular in the last couple of years, their concept is not a new one. The peer-to-peer approach to design distributed applications is as old as the Internet itself. In the beginnings of the Internet, all the host computers in the network had the capability to bidirectionally communicate with each other, and all of them participated in the routing of IP-packets. Each host could act both as server and as client for certain protocols such as FTP or telnet, and was in this respect equal to each other host.

What is peer-to-peer? There are probably as many definitions for peer-to-peer systems as there are systems. I try to give a rough idea of what peer-to-peer is all about by analyzing a quotation of Clay Shirky [56]:

> *Peer-to-peer is a class of applications that take advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, peer-to-peer nodes must operate outside the DNS and have significant or total autonomy of central servers.*

- *Peer*: Every peer is an equal among equals. All nodes participating in the network have the same rights and responsibilities.

- *. . . taking advantage of resources . . .*: Resources are available throughout the Internet. These resources include CPU cycles, storage space and files. By making them accessible to the public, (unused) resources of a computer may be tapped and of benefit for others who are in need of those resources.

- *. . . at the edges of the Internet . . .*: The majority of hosts capable of providing these resources are located on the fringes of the Internet. They are home and personal computers connected to the network via modems or broadband connections.

- *. . . environment of unstable connectivity . . .* Personal computers are not connected to the network all around the clock. They are available only when their owners power them up and establish a connection to the Internet.

- *. . . operate outside the DNS . . .* The IP address of a host computer repeatedly joining and leaving a network may change. This can happen due to dynamic IP addresses assigned by service providers or by connecting laptop computers to different networks. It is therefore not possible to identify a certain node by its IP address. Nodes form a network on top of the IP layer – which is also referred to as 'overlay network'.

- *. . . autonomy of central servers.* There is no instance which has a global view of the peer-to-peer network or controls nodes other than itself. The network built by peer-to-peer nodes is self-organized by the participating nodes themselves.

2.1.1 System characteristics

In the following section the major characteristics of peer-to-peer systems, which may be a driving force for their development, are presented:

Decentralization In client-server architectures, information and processing power are located in a single centralized server. This aggregation of resources in a single spot may lead to performance bottlenecks, waste of resources, or system failure in the case where the server becomes inoperable. Furthermore, centralized systems are expensive to set up in terms of money and maintenance cost. Centralization may be necessary for systems where a single dedicated server is needed – such as security or access-right management systems – but many of the problems entailed by it may be avoided by a decentralized architecture.

In fully decentralized systems it might be problematic to join the network, since no server which knows about the participating network nodes exists. To join the decentralized system, the IP address of at least one node which is currently a member of the network has to be known. One further aspect of decentralization is that a node's owner has full control over the resources (data and performance) he or she wants to make available for the network.

Scalability Directly tied to decentralization is the issue of scalability. The scalability of a system is affected by factors such as centralized computation necessary, maintenance of system state, or inherent parallelism of the application and programming model. Peer-to-peer systems which have a low ratio of bandwidth usage to computation scale extremely well. Examples for those applications would be code breaking or prime number searches, where each node is given a part of the solution space to work upon and then is able to work without communicating with other nodes until the computation has finished.

Self-Organization Self-organization can be defined as a process where the organization of a system increases in complexity without any guidance from outside the system [31]. Self-organization is necessary in peer-to-peer systems, since the connection of a node to the peer-to-peer network is temporary by nature. Nodes can join or leave the network anytime and without prior announcement. In addition, the system load of the network or the amount of nodes which are connected at any time is impossible to predict. An increase of scale of the network brings with it an increase of the probability that nodes in the system will fail or leave. To deal with the dynamic connection of nodes and expected failure of nodes, peer-to-peer systems require self-maintenance and self-repair capabilities.

Cost Sharing the cost of system hardware, content, and system maintenance allows for cheap yet powerful systems. The typical node in a peer-to-peer network is a private personal computer which is available at relatively low cost. By sharing computing resources with other nodes it is possible to outperform super-computers at a fraction of their system cost. Sharing of files enables users to access huge amounts of available data (or storage space) while the local system alone has only little storage capability.

Performance Performance in peer-to-peer systems is influenced by the following resources: processing power, storage capacity, and networking bandwidth. In peer-to-peer systems it is not sensible to measure performance in how long an operation takes in milliseconds, but rather how

quick a query to the system can be answered and how much communication overhead is associated with the query.
Key approaches to gaining performance in a distributed system are caching or replication of data, and intelligent network routing strategies. Caching and replication reduce the network distance between a querying peer and the requested object, while routing strategies allow for efficient network usage and object location.

Fault tolerance Eliminating the risks entailed by architectures with a central point of failure is a key goal of peer-to-peer systems. While peer-to-peer systems don't suffer from a complete cessation of service if the central server fails, network disconnections, unreachability of hosts, network partitions, and node failures are an issue nonetheless. In some systems it is desirable that a node which becomes unavailable for a period of time can resume its function after becoming connected again. To transmit the messages which the node has missed during its absence, different strategies can be employed. In some systems designated nodes (relays) store messages for nodes which are currently unreachable and transmit them when they become active again. Another strategy is to queue messages at their source and send them when the receiver reappears on the network.
In resource location systems, it is important that resources don't become unavailable due to node failures. To tackle this problem, resources can be replicated. In systems such as Napster or Gnutella [32] replication is implemented in a passive and uncontrolled way. The availability of a resource is based on its popularity and on how many nodes have already downloaded the resource. Other peer-to-peer applications like Freenet [11] use a controlled strategy for data replication.

2.1.2 System architectures

There are several approaches how a distributed system can be set up. Although a lot of systems are called peer-to-peer systems, a distinction can be made according to which extent the peer-to-peer paradigm is implemented. One extreme is the traditional client-server approach which offers no peer-to-peer functionality at all. On the other side of the spectrum there are *pure* peer-to-peer networks where each node is an equal among equals. Hybrid networks that are in between client-server architectures and peer-to-peer architectures are also in use. In the following, these architectures and their advantages and disadvantages are explained.

In the client-server approach to distributed systems, one (possibly redundant) host computer is responsible for servicing all requests sent by its clients (see Figure 2.1.a). Two major drawbacks of this architecture can be identified. The first one is that the server has limited network and computing resources at its disposal. Should the server run out of resources, more computing power or network bandwidth has to be provided to service the clients. This is an approach which doesn't scale well for big systems.
The second drawback is that a single server architecture brings with it a single point of failure. Should the server fail (hardware fault, network disconnection, attack on the server) all of the clients become inoperable.

In pure peer-to-peer networks (Figure 2.1.c), each node of the network functions both as server and client. Because of this double function, the term "servent" is often used to describe nodes in peer-to-peer networks. The functionalities of each node are exactly the same as that of any other node in the network. This architecture allows for nodes to join and leave the network at will. The operation of all other nodes will not be impaired should any single node fail. Although the communication overhead in the whole system will increase since no central source of information can be identified, and queries may have to be relayed by several nodes, the work which has do be performed by each node is considerably less than that of a single server.

Hybrid networks maintain information about resource location on a single centralized server (Figure 2.1.b). This approach allows for efficient searching and frees the server from providing the actual resources, which uses most of the network bandwidth. Peers which search for documents are given the identifier of the node which holds the desired document, and the download is done in a peer-to-peer context.
The single point of failure is the server which holds the centralized search directory. Therefore, hybrid systems are susceptible to attacks or network failures.

2.1.3 Classification of approaches

Peer-to-peer systems cover a wide range of application areas, and many different approaches to search in peer-to-peer networks exist. Hauswirth and Dustdar [30] propose a classification of peer-to-peer systems according to three criteria: degree of strucurization, degree of hierarchy and degree of coupling.

Degree of structurization The degree of structurization is determined by the amount of information peers have about resources which are located

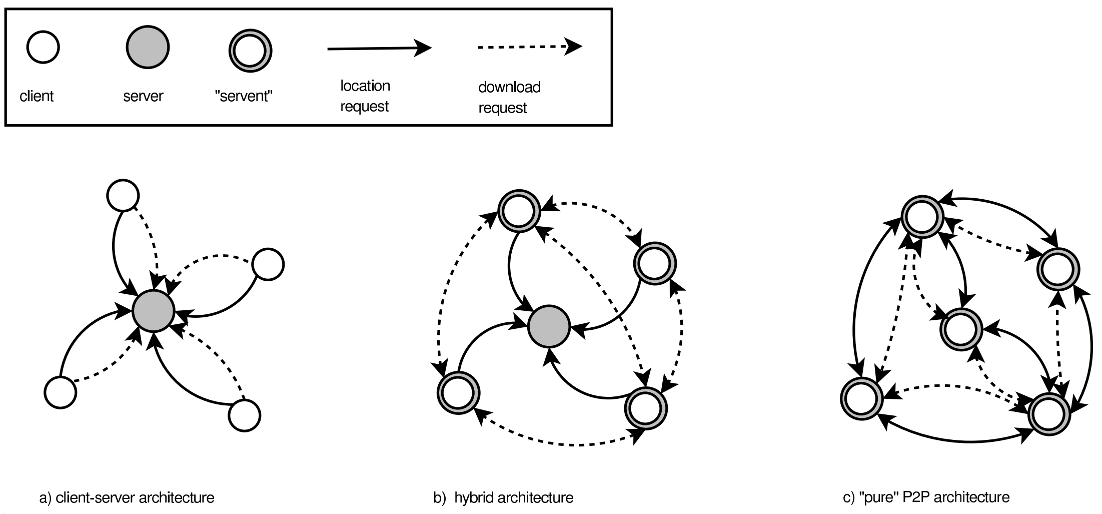


Figure 2.1: System architectures

at remote peers. In unstructured peer-to-peer systems, a peer holds no knowledge about resources stored at other nodes. This allows for highly independent and fault resilient systems, but since no data which can be used for a well directed routing of queries is available, the network load of such systems is rather high.
Structured systems maintain routing tables or other mechanisms to store data about other peers. This data allows for an efficient routing of queries through the system, and therefore the communication overhead between peers is reduced. The reduction of network activity comes at the cost of keeping the routing information up to date.
The most prominent examples for structured systems are distributed hashtables (DHTs) such as CAN [52], Chord [44], Tapestry [69] and Pastry [54]. In distributed hashtables, a hashvalue is calculated for each resource. It serves as a search key. Each node is responsible for the storage of resources matching a certain range of the keyspace. If a node wants to store a resource, the resource has to be copied to the node which is responsble for its key. To allow for efficient searching a small-world graph (see Section 2.2.1) is created by having the routing tables hold references to nodes with exponentially increasing distance to the own keyspace. Distributed hashtables allow a highly efficient search at the cost of relocating whole resources.

Degree of hierarchy The degree of hierarchy describes the role each peer plays in the network. It is differentiated between flat and hierarchical systems. In flat peer-to-peer systems, there is no distinction between the nodes. Each node has exactly the same rights, responsibilities, and tasks as each other node. In hierarchical systems, there are some nodes, referred to as super-peers, which have functionalities that others don't have. Super-peer networks [68] are an attempt combine the efficient search capability of hybrid networks with the autonomy and robustness of pure peer-to-peer networks. Super-peers act as servers to a number of clients. They also interoperate with other super-peers in the fashion of pure peer-to-peer systems. Super-peers are equals where search capability is concerned, and all nodes in the network, including the clients, are equals for downloading purposes. Super-peer networks are easier to service and search operations are less costly in terms of network load, but these advantages come at the price of reduced fault tolerance.

Degree of coupling Two degrees of coupling are possible: tight and loose coupling. In tightly coupled systems, there is only one group of peers at every point in time. Every peer in the network is part of this global group. Peers joining the group are assigned identities which also determine which function the peer plays in the network. There is no possibility for peer groups to form separate networks and evolve independently from the rest of the network. In loosely coupled systems these restrictions do not apply. Peers are free to join and leave networks at will and subnetworks may separate or merge.

The classification of several peer-to-peer applications [1] according to the above criteria is shown in Table 2.1.

	Structurization		**Hierarchy**		**Coupling**	
	unstructured	structured	flat	hierarchic	lose	tight
Gnutella	×		×		×	
Freenet		×	×		×	
CAN		×	×			×
Chord		×	×			×
Tapestry		×	×			×
Pastry		×	×			×
SemAnt	×		×		×	

Table 2.1: Peer-to-peer system classification

The *SemAnt* algorithm, which will be introduced in Chapter 4, can be classified as an unstructured, flat, and loosely coupled system. Although *SemAnt* maintains tables which facilitate the efficient routing of queries, it cannot be

[1]An overview of these applications (other than SemAnt) is given in [1].

considered a structured system, because it lacks an explicit knowledge about the content of other nodes. Nodes in *SemAnt* are equal in every respect and therefore *SemAnt* qualifies as a system with a flat hierarchy. Since it is expected that *SemAnt's* nodes fail, and network separation or merging is possible, *SemAnt* is a loosely coupled peer-to-peer system.

2.1.4 "Servent" components

Each node of a peer-to-peer system needs at least the following components for interaction with other peers in order to be able to perform a search for resources:

Networking component Peer-to-peer systems are dynamic by nature. Nodes are not permanently connected to the overlay network but may join and leave the group at any time. Even the IP-addresses of participants cannot be said to be constant. In between subsequent connections to a peer-to-peer network computers may be assigned different IP-addresses by their service providers each time they connect to the Internet. Therefore, simple DNS lookups to identify nodes are not sufficient.

The purpose of the network component is to take account of this dynamic connection aspects and manage the connection and neighbourhood properties of nodes. To perform the task of mapping peer-to-peer specific node identifiers to actual network addresses, a node must provide a *join*-method which allows other nodes to inform the node they want to connect to of their presence. A node wishing to join a peer-to-peer network needs to know the identity of at least one peer of the group to which it can send the *join*-message. Joining the network may – depending on the architecture – of the network result in further messages which reorganize the networks neighbourhood relations.
Leaving the network is usually not announced. Peers are likely to fail or leave the peer group at any time without warning. This may be due to network congestion or disconnections, shutting down the peer-to-peer protocol or simply powering down the hosting machine. The network component should have some capability to detect the unavailability of nodes which have been connected to the network and reorganize the network structure accordingly.
It is possible that a peer joining a network is already a member of another peer group. This can create a bridge between the two groups and thus merge the two separate networks into a single one.

Resource management component As the name implies, the resource management components purpose is to organize the resources provided by the host computers or their users. The following functionalities are crucial to resource management:

- **insert**(k, r) This function is used to add a resource r to those which are published by the node. The resource is associated with a key k which serves as search criterion to locate r.
- **delete**(r) This function removes the resource r from the node's repository, thus making it unavailable for search requests.
- **search**(k) This function retrieves resources tied to the key k from a node's repository of available resources. If the resource is not found at the node receiving a *search* call, it can forward the request to other nodes in its neighbourhood.

In tightly coupled peer-to-peer systems, the *insert* and *delete* functions may cause a relocation of the resources between the peers since the services they provide are directly tied to their node identifiers.

2.2 Network structures

Peer-to-peer networks consist of nodes connected by links. Due to the self organizing capabilities of peer-to-peer networks, some interesting phenomena concerning the topologies of these networks may emerge. There is a class of networks where, although their size is huge, each node is just a couple of links away from each other node. These networks are called small-world networks. Moreover, there are networks in which the amount of links to a node (its degree) is indirectly proportional to the availability of the node. Networks with such a property are called power-law networks. In this section, these two types of networks will be explored in more detail.

2.2.1 Small-world networks

Small-world networks are based on the assumption that individuals have lots of acquaintances in their immediate neighbourhood and know only some other individuals who are located at a distance from them. The Small-world theory has originally been investigated as a series of sociological experiments by Stanley Milgram [43]. In these experiments, persons from Nebraska attempted to forward postcards to persons they didn't know in Massachusetts.

The postcards had to be forwarded via people that were known by their first name to the current holder of the card. The conclusion of these experiments was that on average six steps were needed to deliver the postcards and that individuals who act based on their own local knowledge only are good at finding short paths to the destination persons.

Duncan Watts and Steven Strogatz [67] showed that the results of Milgram's experiments are caused by the *small-world* effect. If only a few people in a social network have very diverse acquaintances, these people may serve as a bridge between highly interconnected parts of the network that would otherwise be unaware of each other. These bridges can significantly reduce path lengths through the network. Small-world networks have the property that, compared with the amounts of nodes in the network, the networks have a small diameter. Each node in the network is only a few hops away from each other node.
In [47] it has been shown that by randomly changing connections in a regular network-graph, the small-world effect may emerge. A regular graph has each node connected to its neighbours in the same way and is highly clustered: nodes that are near to each other are interconnected by a dense mesh of links. Changing about only one percent of those links to point to distant parts of the network may reduce the network diameter while the clustering in the network is preserved.
Watt's *beta small-world model* is generated by starting with a one-dimensional ring lattice network where each node is connected to k neighbours. Each path in the network has a probability β to be placed between two random nodes. For certain values k and β a network with a low characteristic path length and high clustering coefficient may be created.

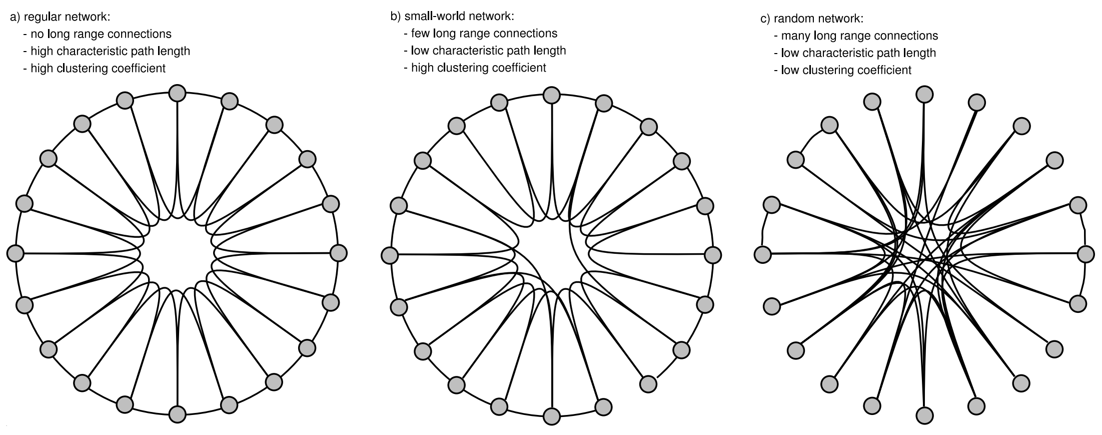

a) regular network:
- no long range connections
- high characteristic path length
- high clustering coefficient
b) small-world network:
- few long range connections
- low characteristic path length
- high clustering coefficient
c) random network:
- many long range connections
- low characteristic path length
- low clustering coefficient

Figure 2.2: Increasing network randomness

An example of how increasing randomness of links affects a network is shown in Figure 2.2.

Jon Kleinberg [34] has further investigated how algorithms perform in specially structured small-world networks. His conclusions might also be used to gain some insights into how efficient the *SemAnt* algorithm works. In Kleinberg-small-worlds, the nodes of the network are placed on a toroid with a lattice size of $n \times n$ nodes. Each node is connected to its immediate neighbours and has one connection to another node which is farther away. The destination of the long distance connection is controlled by a clustering exponent α, which is based on a probability function of the network distance between the two nodes it connects: The node u is connected to v with a probability that is proportional to $r^{-\alpha}$, where r is the lattice distance between u and v (see Figure 2.3).

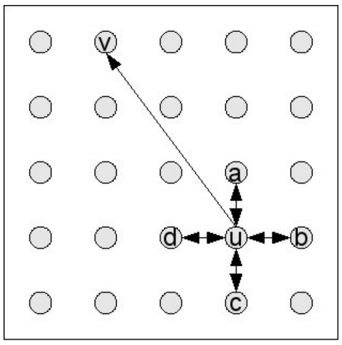


Figure 2.3: Kleinberg-Small-world network lattice

Kleinberg states that the diameter of small-world networks is a logarithmic function of their size. Any two nodes in a network with N nodes are $logN$ steps away from each other at the maximum. When the long range connections in small-world networks are distributed with an inverse-square function – the clustering exponent $\alpha = 2$ – any decentralized algorithm may find a path between any two nodes in the network in a time T, where T is bounded by $logN$. With increasing exponent α, the expected time for a decentralized algorithm to find a path between two network nodes rises asymptotically.

2.2.2 Power-law networks

Power-law distributions [2] describe phenomena where "large" events are rare and "small" events occur very often. An example for this distribution can be seen in the frequency in which words are used in texts: a few words

like "the" and "and" are used often, while other words, which take most of the space of any dictionary, are used only infrequently.
The power-law distribution $P(k) \sim k^{-\gamma}$ describes that the size of an event k is inverse proportional to its occurrence. In a $log-log$ plot, a power-law function can be seen as a straight line with a slope of $-\gamma$.
Networks in which the degrees of the nodes follow a power-law distribution are also called scale-free networks [7]. They can be created by consecutively adding nodes n to the network and linking them to other nodes i with a probability

$$P \sim \frac{k_i}{\Sigma_j k_j}$$

where k is the degree of a node.

Power-law distributions can be observed in various aspects of the Internet:

- access frequency to websites,
- interconnection of documents on the WWW,
- links between Internet routers, etc.

Govindan and Tangmunarunkit [26] developed *Mercator*, a program which maps the connection between Internet routers by sending hop-limited probes from a single host computer. Maps on different scales generated by *Mercator* are shown in Figure 2.4. It can be seen that nodes close to the center of the maps are highly interconnected while the peripheral nodes, which are more frequent have only a few links to other nodes.

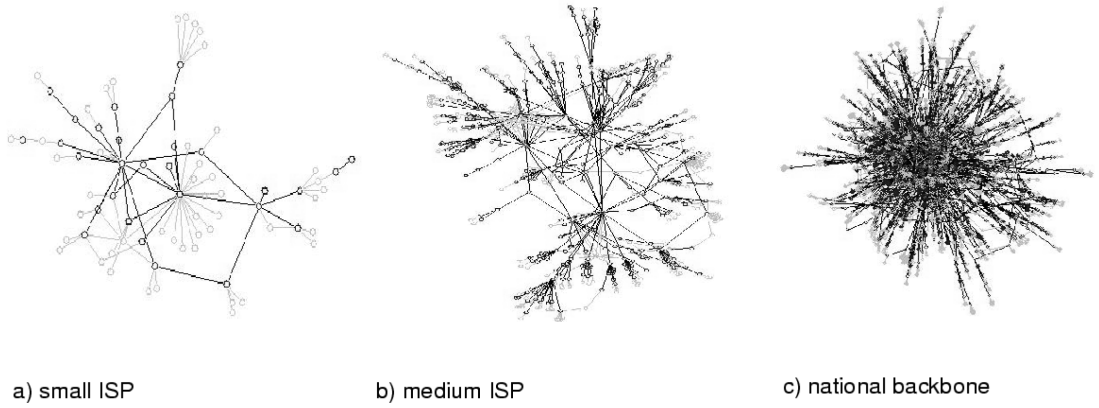


Figure 2.4: Maps generated by *Mercator*

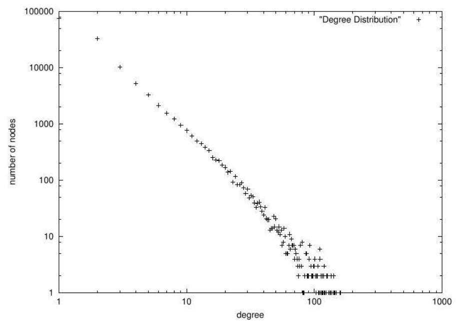


Figure 2.5: Degree distribution of *Mercator* maps

The degrees of connections between nodes, which follows a power-law distribution, is plotted in Figure 2.5.

3 Ant-based approaches to routing in peer-to-peer environments

This chapter shows how the food gathering (foraging) behaviour of real ant colonies can be exploited for finding solutions to graph-based optimization problems in computer science. In Section 3.1, the basic concepts of ant-based algorithms are explained. Section 3.2 presents the *Ant Colony Optimization* meta-heuristic which is a generalization of techniques for applying the foraging behaviour of real ants to optimization problems. Ant Colony Optimization and especially Ant Colony Routing, which is described in Section 3.3, are the foundations of the *SemAnt* algorithm which will be introduced in Chapter 4.

3.1 Introduction

Ant algorithms are inspired by the behaviour and by the communication methods of real ants. In this section, the properties of communication methods used by ant colonies are explained. It is shown how marking a path with chemical substances efficiently leads individual ants of a colony to a food source. This method can be adapted and extended to solve optimization problems in artificial intelligence.

3.1.1 Natural Ants

Natural ants as individuals are very simple animals when their behaviour is considered. They have only little memory and seem to act mostly at random. Ant colonies, which are composed of a number of individual ants, are able to perform various complex tasks by working as a collective. To coordinate tasks such as efficient food gathering, the individual ants need to communicate with each other.

Ants do not communicate directly, but use a type of communication called *stigmergy* [27] instead. Stigmergy is a form of indirect communication in which the environment is used to convey information. A single ant's behaviour is largely controlled by reactions to stimuli it gets from its environment. If an ant changes its local environment in such a way that a specific stimulus is altered, this will affect the behaviour of other ants which pass the location.
The environment can be changed in two ways: by altering it physically in a task-related manner, or sign-based by depositing something which is not directly related to the current task but will affect task-related behaviour. Physically altering the environment by, for example, depositing a ball of mud somewhere will cause other ants also to drop mud-balls near the first one and thus cause the construction of some kind of structure. When using sign-based stigmergy, ants are dropping a variety of volatile chemical substances called pheromones to leave messages for other ants. This form of communication is employed in the food gathering process of ant colonies.

3.1.2 Trail-laying and trail-following

Using a colony of Argentine ants (*iridomyrmex humilis*), Goss et al. [25] investigated the food gathering behaviour of those creatures. In the experiment, the colony was separated from a food source. It was then given access to the source by two bridges of different length (see Figure 3.1). An ant travelling to the food source faces the choice of taking either the longer or the shorter route. Since ants do not have a global view of the area, they can only make their decision at the fork of the path. The first ants choosing a path have no information at all about how to reach the food source quickly, and therefore make a decision at random. If two ants start at the same time and take different branches, the one taking the shorter path will be the first one to return to the nest with some food item. Since the ants are marking the paths they have been travelling with pheromone trails, the shorter path will have a higher concentration of pheromones at the time the first ant returns (see Figure 3.2). If another ant starts its way and comes to the bifurcation of the path it can make its choice by analyzing the pheromone concentrations on the paths. With a high probability, it will choose the shorter path because more pheromones have already been dropped there. By choosing the shorter path and marking it as well, this decision will further increase the attractiveness of the path for subsequent ants.
In this experiment, after a few minutes most ants will use the shorter path. The emergence of the problem's solution is caused by *positive feedback* and by *differential path length*. Goss et al. observed that the amount of ants that will be travelling either path is directly proportional to how great the lenght difference between the two paths is.

Two further experiments show that there are problems associated with the pheromone trail laying. In the first experiment, the ants are presented only the longer path first, and after a while the shorter path is made accessible. As a consequence the ants won't choose the shorter path since the pheromone concentration on the longer path is already high and will be further reinforced. In the second experiment, the shorter path is blocked after awhile. The ants will still try to follow the path only to realize they can not reach the food source. It takes rather long for the colony to discover and start to utilize the longer path. Although the pheromone trails will evaporate over time, the evaporation rate of natural pheromones is too low to be of any real use for quickly accounting to the two situations described above.

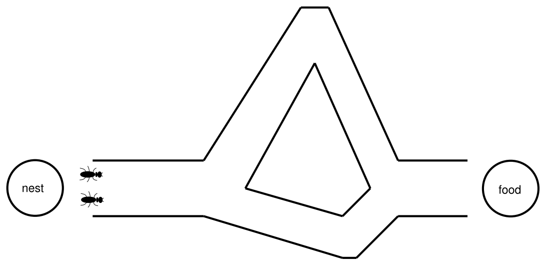


Figure 3.1: Differential pathlength, initial state

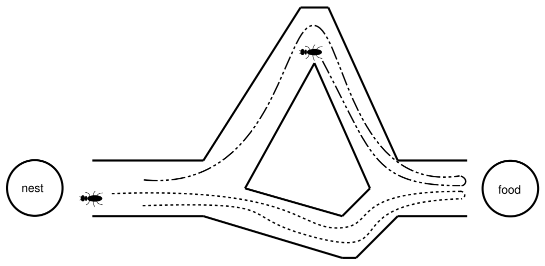


Figure 3.2: Differential pathlength, advanced state

3.1.3 Application of trail-laying and trail-following to optimization problems

Ant-based algorithms mimic the pheromone trail laying communication strategies of real ant colonies to solve optimization problems which can be mapped onto a network graph. They employ the stigmergic principle realized by trail-laying and trail-following, which has been described in the previous section. The selection of a short path between two points is extended to find the best path between any two vertices in a graph as shown in Figure 3.3.

Two strategies for the evaluation of the paths found by the ants are employed: implicit and explicit solution evaluation. In the case of implicit solution evaluation, the differential path length effect is exploited. Ants which find solutions faster are the first to bias the search process of the following ants. They drop a fixed amount of pheromones, and thus reinforce their found solutions. When evaluating the solution explicitly, the actual quality of the found solution has

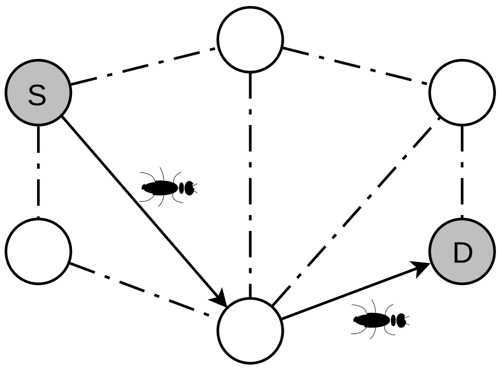


Figure 3.3: Solution building

Problem	**Algorithms**
Travelling Salesman	AS [19], MMAS [60]
Network routing	ABC [55], AntNet 3.3
Graph colouring	ANTCOL [15]
Quadratic Assignment	HAS-QAP [22], MMAS-QAP [61], AS-QAP [39]
Machine scheduling	ACS-SMTTP [8]
Vehicle routing	AS-VRP [9], MACS-VRPTW [21]
Multiple knapsack	AS-MKP [35]
Frequency assignment	ANTS-FAP [38]
Sequential ordering	HAS-SOP [20]

Table 3.1: Problems solved by ant-algorithms

to be calculated in some way, and the amount of pheromones dropped by ants depends on the solution's quality. Usually the combination of implicit and explicit solution evaluation is applied in ant algorithms.
As already mentioned, pheromones evaporate with passing time, and so paths that have been marked by ants are "forgotten" again. While natural pheromones evaporate very slowly, increasing the evaporation rate of artificial pheromones can improve the optimization process. Paths which are good, but less than optimal, do not get too much attention by ants if the pheromone amounts dropped there diminish quickly.

Different optimization problems ranging from the Travelling Salesman Problem (TSP) to graph coloring have been tackled by employing ant algorithms. Table 3.1 shows some of these problems and some algorithms which have been used to solve them.

3.2 Ant Colony Optimization Meta-Heuristic

The Ant Colony Optimization meta-heuristic as proposed by Marco Dorigo and Gianni Di Caro [17] can be applied to solving optimization problems with the following characteristics:

- The problem can be mapped onto a graph $G(C, L)$ composed of the components $C = \{c_1, c_2, \ldots, c_n\}$ and links $L = \{l_{c_i c_j} | (c_i.c_j) \in (C \times C)\}$ between a subset of the components. The components are the nodes of the graph.
- Each link $l_{c_i c_j} \in L$ has an associated cost function $J_{c_i c_j} \equiv J(l_{c_i c_j}, t)$ where t is a parameter possibly denoting time.
- A set of constraints $\Omega \equiv \Omega(C, L, t)$ is assigned over the components of the graph.
- The states of the problem are defined as sequences $s = \langle c_i, c_j, \ldots, c_k, \ldots \rangle$ over the elements of C. S is the set of all possible sequences and $\tilde{S}$ is a subset of S containing all sequences which are feasible under the constraints defined in Ω.
- A neighbourhood structure of states is defined as follows. The state s_2 is a neighbour of s_1 if $s_1, s_2 \subseteq S$ and s_2 is reachable from s_1 in one logical step. This means that if c_1 is the last component of s_1, there has to be a $c_2 \in C$ with $l_{c_1 c_2} \in L$ and $s_2 \equiv \langle s_1, c_2 \rangle$
- A solution Ψ is an element of $\tilde{S}$ which satisfies all requirements of the problem.
- Each solution Ψ has a cost $J_\Psi(L, t)$ which is calculated as a function of all costs $J_{c_i c_j}$ of connections belonging to the particular solution.

In Ant Colony Optimization a problem defined as above, is solved by employing the trail-laying and trail-following priciples as described in Section 3.1.2. The ants will travel the links, and upon finding a solution will update pheromone trails τ_{ij} for each link $l_{c_i c_j}$ which is part of their solution.
Associated with the ant colony and its individual ants are the properties listed below:

- Each ant has enough complexity to find a solution to the defined problem. The solution is probably poor. Good solutions are found only through the collaboration of the ants.

- Ants make use only of information they themselves possess or information which is stored at the node they are currently visiting. Ants do not have any global knowledge about the system in which they are operating.
- Communication between ants is performed only by laying and following pheromone trails.
- Ants do not adapt to a problem. Instead, they change the representation of the problem and how it is seen by other ants.
- Ants search for feasible solutions with a minimal cost.
- Each ant k has a memory $\mathcal{M}^{\parallel}$ to store information about the path that has been travelled. The memory is used to build solutions, evaluate the solution, and backtrace the paths they have followed.
- An ant k in a state $s_r = \langle s_{r-1}, i \rangle$ can move to any node c_j in its feasible neighbourhood so that the next state s_{r+1} is valid in the sense that $s_{r+1} \in \tilde{S}$.
- An ant k is assigned a start-state ${s^k}_s$ and can have multiple termination-states e^k.
- Ants start at their start-state ${s^k}_s$ and incrementally build a solution by travelling between neighbouring nodes. The solution construction stops when any termination-state e^k is reached by at least one ant.
- An ant can move from a node c_i to any neighbouring node c_j as long as there is no constraint in Ω prohibiting it. Which node it will travel to is chosen in a probabilistic manner. The function which governs the choice is influenced by (i) the problem constraints, (ii) the content of the ants memory $\mathcal{M}^{\parallel}$, and (iii) information from the c_i's routing table $\mathcal{A}_{\rangle}$ which is derived from the pheromone levels τ_{ij} for paths to c_i's neighbours.
- When travelling from node c_i to node c_j in order to find a solution, an ant may update the pheromone levels τ_{ij} on the link between the two nodes. This action is called "online step-by-step pheromone update".
- If a feasible solution has been found and the ant retraces its steps to the originating node it may also update the pheromone levels on the paths it passes. This is called "online delayed pheromone update".
- If an ant has found a solution and has retraced its path to its originating node, it dies and all resources allocated to the ant are freed.

Besides the ants' activities, there are two other possible actions in algorithms which implement the ACO heuristic. These actions are *pheromone evaporation* and *daemon actions*. Globally reducing pheromone levels is performed to prevent that solutions which are good but not optimal receive too much attention. "Forgetting" paths by evaporating pheromones leads to a greater area of exploration by the ants. Daemon actions are any actions which cannot be performed by the ants themselves. Daemon actions may have a global view of the system and may update pheromones to control the search process. Pheromone updates by a daemon are called "offline pheromone updates".

A pseudocode representation of the above heuristic is given in Figure 3.4.

The travelling salesman problem (TSP) was the first problem to be tackled by this approach. In the case of the TSP, C would be the set of cities, L the available paths between the cities, and a solution Ψ would be a Hamiltonian circuit (a sequence containing all elements of C exactly once) through all the cities.

3.3 Ant Colony Routing

Ant based optimization can also be applied to routing data packets through communication networks in an efficient manner. Routing in this case means building routing tables, which are consulted to select the path along which a data packet is sent from its source to its destination node. While always selecting the shortest path for a data packet reduces the overall load of the network, individual nodes may experience a high communication load, become congested, and thus increase the time it takes the data to reach its destination. It may therefore be desirable to distribute the workload over the network, and also select longer paths through less used parts of the network, to avoid delays or packet-loss due to node congestion.
In ant based routing algorithms, ants are used to gather information about the workload at the nodes in the network, and the time it takes to travel the links between the nodes. This information is used to update routing tables at each node. The routing tables hold probabilistic values that are used to select the links along which the data packets are sent to their respective destinations.

Prominent routing systems based on ant algorithms are Ant Based Control (ABC) [55] by Ruud Schoonderwoerd et al., and AntNet [10] by Gianni Di Caro and Marco Dorigo. Concepts from the AntNet algorithm have been adapted to *SemAnt* (see Chapter 4). The remainder of this chapter will explain the AntNet algorithm, starting with the data structures and concepts used by AntNet.

```
procedure acoMetaHeuristic() {
    while (!terminationCriterionSatisfied){
       scheduleAntCreation();
       newActiveAnt();
       pheromoneEvaporation();
       daemonActios(); // optional
    }
}
procedure newActiveAnt() {
    initializeAnt();
    $\mathcal{M}$ = updateAntMemory();
    while (currentState != targetState) {
       $\mathcal{A}$ = readLocalAntRoutingTable();
       $\mathcal{P}$ = computeTransitionProbabilities($\mathcal{A}$, $\mathcal{P}$, $\Omega$);
       nextState = applyAntDecisionPolicy($\mathcal{P}$, $\Omega$);
       moveToNextState(nextState);
       if (onlineStepByStepPeromoneUpdate) {
          depositPheromoneOnTheVisitedArc();
          updateAntRoutingTable();
       }
       $\mathcal{M}$ = updateInternalState();
    }
    if (onlineDelayedPheromoneUpdate) {
       foreach (visitedArc $\in$ $\Psi$) {
          depositPheromoneOnTheVisitedArc();
          updateAntRoutingTable();
       }
    }
    expire();
}
```

Figure 3.4: ACO meta-heuristic pseudocode

P_{11}	P_{12}	$\ldots$	P_{1N}
P_{21}	P_{22}	$\ldots$	P_{2N}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
P_{L1}	P_{L2}	$\ldots$	P_{LN}

Table 3.2: AntNet routing table

Forward-Ant Forward-Ants are agents which are sent at regular time intervals by all nodes in the network. The aim of a Forward-Ant is to find a good (in terms of time) path to a destination node, and measure the time of each intermediate hop to the destination node. Because a Forward-Ant's job is also to detect network congestions, Forward-Ants are sent through the same queue as data-packets in the network. Therefore, if a Forward-Ant encounters a node which is currently experiencing heavy traffic, it will be somewhat delayed by the waiting data-packets. To store information about the route it took, each Forward-Ant has a stack where it puts the data collected from each node on its way to the destination.

Backward-Ant If a Forward-Ant has arrived at its destination, the timing data that it has acquired on its way is copied to a Backward-Ant. The Forward-Ant dies and the Backward-Ant retraces the path taken by the Forward-Ant, back to its originating node. On its way back, the routing tables of the nodes it passes may be updated. Backward-Ants do not use the same queue as data packets and Forward-Ants, but a high priority queue, so that the network state can be quickly and accurately updated.

Routing table Each node in AntNet owns a routing table $\mathcal{T}$, holding probabilistic values which influence the routes taken by Forward-Ants and data packets. The size of the routing table is $N \times L$, where N is the amount of nodes in the network and L the amount of neighbors of the node (see Table 3.2). An entry P_{nd} in the routing table describes the probability that n will be the next node chosen by a Forward-Ant, if its destination node is d. Obviously the following equation has to be satisfied:

$$\sum_{n \in \mathcal{N}_k} P_{nd} = 1, \quad d \in [1, N], \quad \mathcal{N}_k = \{neighbors(k)\}$$

Model Besides the routing table each node owns a model $\mathcal{M}$, which describes the network traffic distribution as it is perceived the node. For each node n in the network there is an entry in $\mathcal{M}$ storing the best time, mean time, and variance it took to reach the node in a defined time window. The model is used to get an estimation how long it will take a packet to reach a certain destination node. Upon return of a Backward-Ant the data gathered by the ant is compared with the model, and the routing table is updated according to the quality of the path chosen by the Forward-Ant.

The AntNet algorithm itself can be described as follows.

- At regular time intervals at each node s in the network a Forward-Ant $F_{s \to d}$ is generated. The probability p_d for choosing a node d as destination is calculated as
$$p_d = \frac{f_{sd}}{\sum_{d'=1}^{N} f_{sd'}},$$
where f_{sd} is a measure of the data flow in bits or packets from s to d. This probability favors the exploration of paths to nodes which are preferred destinations for data packets from s.

- At a node k, the selection of the next node n in a Forward-Ant's path is based on a probability P'_{nd}, which takes into account the probabilistic values stored in $\mathcal{T}$ and the current workload of the transport queues to the neighbors of k. In the equations below, q_n is the size of the queue to k's neighbor n in bits, and α is a parameter weighting the influence of the queues' workloads.
$$P'_{nd} = \frac{P_{nd} + \alpha l_n}{1 + \alpha(|\mathcal{N}_k| - 1)}, \quad \text{where } l_n = 1 - \frac{q_n}{\sum_{n'=1}^{|\mathcal{N}_k|} q_{n'}}$$

- On its way to the destination node d, the Forward-Ant collects information about the traffic conditions encountered at the intermediate nodes k on its way. The node identifiers and the time it took to reach the nodes are pushed on the ant's stack $S_{s \to d}(k)$.

- If the ant has moved in a cycle, all data collected during the time in the cycle is deleted from the ant's stack. If the ant has spent more than half of its lifetime in the circle the ant itself is deleted, otherwise it continues its way to its destination.

- If the Forward-Ant $F_{s \to d}$ reaches its destination, a Backward-Ant $B_{d \to s}$ is generated. $B_{d \to s}$ travels back to s along the same path that has been taken by $F_{s \to d}$. It uses a high priority queue so that the routing tables can be quickly updated.

- When, on its way back, $B_{d \to s}$ arrives at a node k coming from a node f the model $\mathcal{M}_k$ and routing table $\mathcal{T}_k$ are updated. Entries in those data

structures which correspond to the destination node d' are affected by the updates. If the time it took to reach a node on the way other than d (i.e. a node in the path from k to d) is considered to be good (by comparing it with the estimates in $\mathcal{M}$), the entries for these nodes may also be updated.
In $\mathcal{M}$ the mean, and variance estimates, as well as the best time to reach d' from k are recalculated according to the timing information collected in the ants stack.
In $\mathcal{T}$ the value of $P_{fd'}$ is increased. All other values P_{fn} are decreased by normalization. The transition rule for the increase is

$$P_{fd'} \leftarrow P_{fd'} + r(1 - P_{fd'})$$

where $r \in (0, 1]$ is a reinforcement factor measuring the quality (depending on trip time and estimations in $\mathcal{M}$) of the path. The above transition rule favors the increase of $P_{fd'}$ which have low values, so newly found good paths will be explored more quickly.

A pseudocode description of the AntNet algorithm is shown in Figure 3.5.

```
t = CurrentTime;
t_end = EndTimeOfSimulation;
t_int = AntGenerationInterval;
foreach (Peer) {    # Concurrent activity
    $\mathcal{M}$ = localTrafficModel;
    $\mathcal{T}$ = nodeRoutingTable;
    while (t <= t_end) {
       in_parallel {    # Concurrent activity at each node
          if (t % t_int == 0) {
          destinationNode = selectDestinationNode(dataTrafficDistribution);
          launchForwardAnt(destinationNode, sourceNode);
          }
          foreach (ActiveForwardAnt) {
             while (currentNode != destinationNode) {
                nextNode = selectLink(currentNode, destinationNode, $\mathcal{T}$,
                                      linkQueues);
                putAntOnLinkQueue(currentNode, nextNode);
                waitOnDataLinkQueue(currentNode, nextNode);
                traverseLink(currentNode, nextNode);
                pushOnStack(nextNode, elapsedTime);
                currentNode = nextNode;
             }
             launchBackwardAnt(destinationNode, sourceNode, stackData);
             expire();
          }
          foreach (ActiveBackwardAnt) {
             while (currentNode != destinationNode) {
                nextNode = popFromStack();
                waitOnHighPriorityLinkQueue(currentNode, nextNode);
                traverseLink(currentNode, nextNode);
                updateLocalTrafficModel($\mathcal{M}$, currentNode, sourceNode, stackData);
                reinforcement = getReinforcement(currentNode, sourceNode,
                                                 stackData, $\mathcal{M}$);
                updateLocalRoutingTable(($\mathcal{T}$), currentNode, sourceNode,
                                        reinforcement);
             }
          }
       }
    }
}
```

Figure 3.5: AntNet pseudocode

4 The SemAnt algorithm

This chapter gives a description of the *SemAnt* algorithm which has been proposed by E. Michlmayr *et al.* in [40]. The algorithm presented here is the base of the practical work – the implementation of a simulation environment for this algorithm – which will be discussed starting with chapter 5.

In Section 4.1, an application scenario for *SemAnt* is constructed. The algorithm's specification is presented in Section 4.2. Exploitation of hierarchical structures will be covered in Section 4.3, and in Section 4.4 it is described how *SemAnt* deals with dynamic network behaviour.

4.1 Application scenario

The *SemAnt* algorithm is a distributed search engine designed to operate in an overlay network where each peer stores a number of documents in a local repository. The stored documents are classified with keywords or concepts which are taken from a hierarchical classification scheme or taxonomy. Hierarchical classification schemes are employed for example by the ACM Computing Classification System [5] which is used to describe scientific articles from the domain of computer science, or the DMOZ - Open Directory Project [46], which aims to classify websites according to their content. One or more concepts taken from the classification scheme may be assigned to one document. Users operating the nodes in the network may employ keywords, which of course have to be taken from the classification scheme, to query for documents. Documents meeting the search requirements of this query have to be associated with all of the keywords in the query.
If the node that has been queried cannot satisfy the query by retrieving documents from its local storage, it generates messages to ask its neighbour nodes for matching documents.

Peers are identified by a unique id, for example their IP address. Documents can be identified by their filenames and may be added or removed from a peers repository while the peer is connected to the network.

The algorithm doesn't provide for peer discovery, so it is assumed that when joining the network, each peer is already aware of its neighbours. Another requirement is that the peers' clocks are synchronized or a single reference clock is used for all peers.
The aim of *SemAnt* is to create routing tables at each peer to optimize the network so that querying for a document will result in a minimal network load while a maximal amount of documents is found.

4.2 Specification of the algorithm

SemAnt is a pure peer-to-peer search algorithm. No super-peers or special peers providing meta-information about the network are employed in this solution. Each peer shows exactly the same behaviour and offers the same functionality.
The *SemAnt* design is based on Ant Colony Optimization (see Section 3.2) and Ant Colony Routing (see Section 3.3). *SemAnt* incorporates modified features of ACO and ACR to provide distributed query routing functionality in a dynamic environment.

4.2.1 Data structures of SemAnt Peers

Each peer in *SemAnt* is aware of its neighbours and holds a table η where the cost of sending a message to a neighbouring peer P_u is stored for each neighbouring peer. Each peer also holds a table τ to store the quality of links represented by pheromones. Since the network has to be optimized not only for a single concept, but for all concepts in the taxonomy, there have to be as many different pheromones as there are concepts. Consequently the size of τ is $|C| \times n$, where $|C|$ is the number of concepts in the taxonomy and n is the number of neighbours of the peer owning τ. The usage of multiple pheromone types to optimize a network is elaborated by Sim and Sun in [57].
Together the tables τ and η will be used to route messages between peers. A path for a message is more likely to be chosen if its pheromone levels are high and the cost for using the path is small. Initially, each entry in τ will be initialized with the same small amount of pheromones.

4.2.2 Types of Ants

Messages between peers will be represented by ants in *SemAnt*. The *AntNet* strategy to employ forward and backward ants to solve distributed problems is adopted in *SemAnt*. *AntNet* aims at finding the optimal path between a starting node and known destination node. In *SemAnt* the destination of an ant depends on the content of the query. Since each node is only aware of its own routing tables the destination node is unknown or might not even exist if there are no documents matching the query. To prevent ants from searching for a document that does not exist, each ant has a lifetime parameter T_{max} specifying the maximum time an ant will spend to look for documents. Upon finding appropriate documents, backward ants will return them to the user's node along the path that has been taken by the forward ant. This will limit the maximum time between dispatching a query and receiving documents to $2 \cdot T_{max}$.

Basically, there are three different kinds of ants:

- **Forward Ants** are used to locate documents in the networks. A query for documents will be carried from peer to peer by a forward ant. When searching for documents, the forward ants employ two different strategies: exploitation and exploration. Choosing the exploitation strategy causes an ant to follow a known and often travelled path. When a forward ant is exploring, it tries to create new paths by finding documents on paths that are less frequented. Exploration in *SemAnt* may cause a forward ant to be cloned so that more than one path can be investigated.

- **Backward Ants** are spawned by forward ants which have discovered a peer that provides documents matching the query they are transporting. Whenever documents are found, a backward ant carries these documents back to the peer where the query has been issued. The backward ant will backtrace the path that the forward ant has taken to locate the document source. On its way back to the original node the backward ant will instruct intermediate nodes to update their pheromone tables so that the path to the document source will be marked.

- **Change Ants** will be created by peers which add or remove documents from the network. They are sent to inform other peers of this change and cause them to modify their pheromone tables to reflect this change. Change Ants are described in Section 4.4.

4.2.3 Routing the ants

The routing information at each peer is provided by the tables η (link cost) and τ (pheromone amounts). These tables will be maintained and updated by backward ants that are returning with documents. The amount of pheromones dropped on an entry τ_{cu} depends on the number of documents that have been found and the path cost associated with finding those documents. Since all peer clocks are synchronized, a forward ant can determine the time it takes to travel between two peers. Backward ants carry this information back and update the link costs stored in η accordingly.

Since forward ants generate backward ants which update the pheromone tables, paths that are heavily travelled will be reinforced while paths that are seldom taken vanish due to pheromone evaporation. The degree of optimization for a concept depends on the amount of backward ants that update pheromone tables for this concept. It therefore is directly dependent on the popularity of a concept in queries.

A general overview of the algorithm is presented as pseudo-code in Figure 4.1 and as flowcharts in Figures 4.2 and 4.3. A detailed description of the algorithm will be given starting with Section 4.2.4.

4.2.4 Step-by-step description

In this section a step by step description of the original version of the algorithm is given. The algorithm starts when a query Q is issued at a peer P^Q. Q is a query used to search for documents that have been assigned the keyword c.

Step 1 The document repository at the peer that issues the query (P^Q) is checked. If it contains any documents matching the search criteria, they are returned to the user. If the number of documents found in P^Q's repository is less than a predefined parameter D_{max}, the algorithm will continue at Step 2. If enough documents are returned, the algorithm will terminate.

Step 2 A forward ant F^Q is created at P^Q. The forward ant is assigned a starting time T_{Fstart} and a timeout parameter T_{max} limiting the lifetime of the ant. P^Q is put on the ants empty stack $S(F^Q)$ of already visited peers. A list $LC(F^Q)$ storing the link costs of all paths travelled by F^Q is initialized.

```
t = CurrentTime;
t_end = EndTimeOfSimulation;
foreach (Peer) {     # Concurrent activity
    initializePheromoneTables();
    initializeLinkCostsToNeighbourPeers();
    while (t < = t_end) {
       in_parallel {     # Concurrent activity at each peer
          if (Query Q) {
             checkLocalDocumentRepository();
             createForwardAnt(query_parameter);
          }
          foreach (StartingForwardAnt) {
             while (Timeout_not_reached) {
                applyTransitionRule ();
                t_a = CurrentTime;
                foreach (ActiveForwardAnt) {
                   GoToPeer(P_j);
                   t_b = CurrentTime;
                   checkLocalDocumentRepository();
                   if (DocumentsFound > 0) {
                      createBackwardAnt(stackData, P_j);
                   }
                   addDataToStack(P_j, t_b-t_a);
                }
             }
          }
          foreach (StartingBackwardAnt) {
             do {
                peer = pepStackData_Peer();
                GoToPeer(peer);
                applyPheromoneTrailUpdateRule();
                updateListCosts(popStackData_Costs());
             } while (peer != source_peer);
          foreach (PeriodicalTimeInterval t_e) {
             applyEvaporationRule();
          }
       }
    }
}
```

Figure 4.1: *SemAnt* pseudo-code

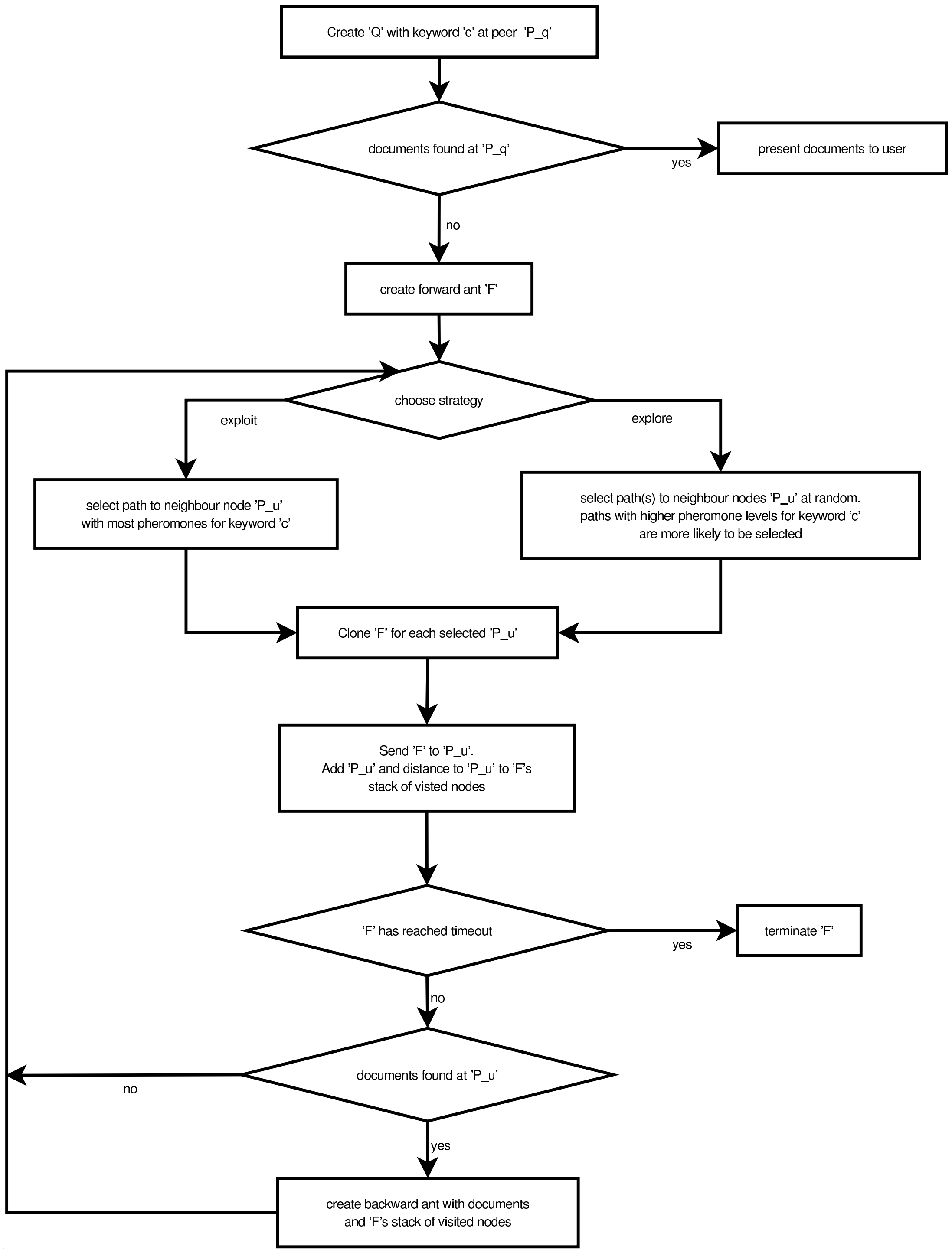

Create 'Q' with keyword 'c' at peer 'P_q'
documents found at 'P_q'
yes
present documents to user
no
create forward ant 'F'
choose strategy
exploit
explore
select path to neighbour node 'P_u' with most pheromones for keyword 'c'
select path(s) to neighbour nodes 'P_u' at random. paths with higher pheromone levels for keyword 'c' are more likely to be selected
Clone 'F' for each selected 'P_u'
Send 'F' to 'P_u'. Add 'P_u' and distance to 'P_u' to 'F's stack of visted nodes
'F' has reached timeout
yes
terminate 'F'
no
documents found at 'P_u'
no
yes
create backward ant with documents and 'F's stack of visited nodes

Figure 4.2: Forward Ant Flowchart

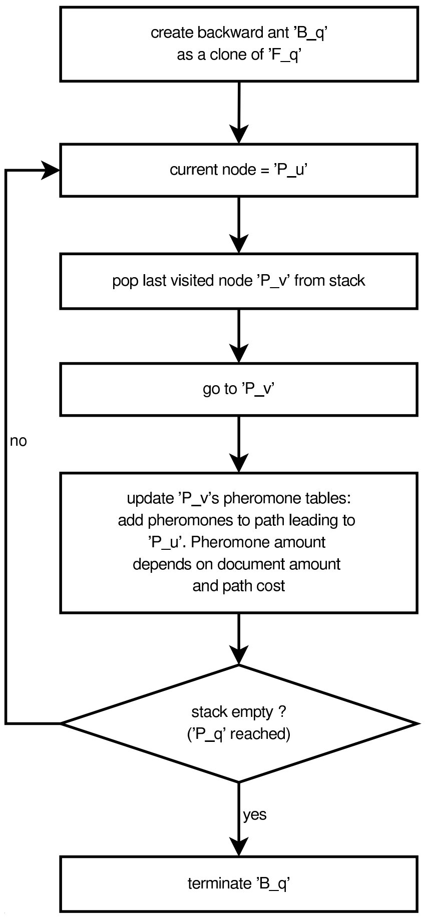

create backward ant 'B_q' as a clone of 'F_q'
current node = 'P_u'
pop last visited node 'P_v' from stack
go to 'P_v'
update 'P_v's pheromone tables: add pheromones to path leading to 'P_u'. Pheromone amount depends on document amount and path cost
stack empty ? ('P_q' reached)
no
yes
terminate 'B_q'

Figure 4.3: Backward Ant Flowchart

Step 3 Routing forward ants to their destinations is done by employing exploitation and exploration strategies which are also used in *Ant Colony System* [18]. Each time a route for an ant has to be found, one of these strategies is chosen at random. The exploitation strategy is pretty straightforward: the ant follows the path with the highest quality - depending on pheromone amounts and link cost.
Since more than one node can be the destination of a forward ant, the exploration strategy where new paths are discovered, has to be adapted. When exploring, a random value (again depending on pheromone amounts and link cost) denoting whether a path should be chosen is calculated for *each* possible path. Should more than one path be selected for exploration, the forward ant has to be duplicated and copies of the original ant continue in different directions.

The transition rule is applied to find out which path or paths should be travelled by F^Q. With a probability ω_e the forward ant will employ

the exploiting strategy. The exploring strategy is chosen with probability $1 - \omega_e$.
If F^Q makes use of the *exploiting strategy*, the following transition rule is applied to choose a path to the present best neighbour peer P_j:

$$j = arg\ max_{u \in U \wedge u \notin S(F^Q)} \left([\tau_{cu}] \cdot [\eta_u]^\beta\right)$$

where U is the set of neighbour peers to the current peer P_i and $S(F^Q)$ is the set of peers that have already been visited by F^Q. τ_{cu} is the amount of pheromones on the path to the neighbour node u and η_u is the cost to travel to P_u which again is weighted with a parameter β.
If the *exploring strategy* is chosen, the transition rule below is checked for *every* neighbour peer P_j to see if a forward ant should be sent to P_j.

$$p_j = \frac{[\tau_{cj}] \cdot [\eta_j]^\beta}{\sum_{u \in U \wedge u \notin S(F^Q)} \left([\tau_{cu}] \cdot [\eta_u]^\beta\right)}, \quad GOTO_j = \begin{cases} 1 & \text{if } q \leq p_j \wedge j \in U \\ 0 & \text{else} \end{cases}$$

where q is a random value $q \in [0, 1]$ and $\sum_{j \in U \wedge j \notin S(F^Q) p_j} = 1$. If $GOTO_j$ equals 1, a clone of F^Q is created and sent to P_j.

Step 4 When F^Q arrives at P_j, the local document repository of P_j is checked to find documents d that match keywords c of the query Q.The result of the local search is a set D containing every found document.

Step 5 If documents were found and thus D is not an empty set, a backward ant B^Q is generated. The document set D is attached to B^Q. P^D – the node that stores D – is passed to B^Q so that upon return of the backward ant a direct connection between P^Q and P^D can be established to download the found documents. B^Q also receives a copy of $S(F^Q)$.
The total path cost T_D from P^Q to P^D is calculated by summing up all entries in the set $LC(F^Q)$. B^Q then travels back to its originating node P^Q along the path connected by the nodes that have been accumulated on the stack $S(F^Q)$. At each node it passes B^Q drops an amount of pheromones Z to mark the quality of the travelled path according to the following rule:

$$\tau_{cj} \leftarrow \tau_{cj} + Z, \text{where } Z = \omega_d \cdot \frac{|D|}{d*} + (1 - \omega_d) \cdot \frac{T_{max}}{2 \cdot T_d}$$

The amount of pheromones dropped is dependent on the number of documents that have been retrieved and the cost of the path that has been travelled by F^Q to find the documents. In the above rule, ω_d is a parameter weighting the influence of the amount of found documents against the path cost to reach P^D from P^Q. Z is derived by comparing the quality of the found solution against an optimal solution. Since no known optimal solution exists, an 'optimum' for the total path cost is assumed to be $\frac{T_{max}}{2}$ and the 'optimal' number of documents is estimated by using a constant parameter d^*.

Step 6 P_i is put on F^Q's stack of visited peers $S(F^Q)$ and the cost of the last travelled path is added to $LC(F^Q)$.

Step 7 If the forward ant has exceeded its lifetime, i.e., $T_{Fstart} + T_{max} > CurrentTime$, the forward ant is terminated. Otherwise, the algorithm continues at Step 3.

4.2.5 Multiple keywords

The algorithm as presented in the previous section works on the assumption that queries will be used to search for documents with a single keyword. The aim of *SemAnt* is to use multiple keywords though. Some adaptations to the algorithm need to be made to provide for the application of multiple keywords. In case more than one keyword is used, the pheromone trails for all used keywords $c_1, ..., c_n$ need to be updated.

4.2.6 Evaporation

Since documents can be removed from repositories, or nodes may leave the network altogether, pheromone trails will become inaccurate or obsolete. To prevent ants from following outdated pheromone trails, the pheromones need to be updated as a function of passing time. Pheromone trails will gradually become weaker and paths that are not used anymore due to missing documents will become subject to deterioration, since no backward ants will drop pheromones on them anymore. Thus the so called evaporation of the pheromones will stop ants from travelling in the wrong direction for an extended period of time.

In defined intervals t_e each node has to update the pheromones on the paths to its neighbour nodes. Pheromones for all concepts c on paths to each neighbour node P_u are reduced by an evaporation parameter $\rho \in [0,1]$ according to the evaporation rule below.

$$\tau_{cu} \leftarrow (1-\rho) \cdot \tau_{cu}$$

4.3 SemAnt and taxonomies

SemAnt is supposed to locate documents which have been classified with concepts taken from a taxonomy. It is assumed that there exists a locality as far as document distribution between peers is concerned. This means that if a node holds some documents that fit a certain concept it is also likely that documents of related concepts can be found at this node [59]. For example if a node has some documents on the subject '/recreation/sports/martial-arts/ju-jutsu' there is a good chance that the nodes user may also show interest in sports such as karate, judo, aikido, etc. which can also be classified as martial-arts.

As shown in [42], it is possible to increase the search performance of the algorithm by exploiting the relation between concepts and the locality mentioned above. To do so, the proposition is that backward ants also drop an equal amount of pheromones on each superconcept of the keywords c used in the query.
Forward ants searching for a document additionally check the amount of pheromones on the superconcepts to decide upon which path to take. This behaviour leads forward ants to find the desired documents faster. This is done by changing the transition rule for the exploiting strategy to

$$j = arg\ max_{u \in U \wedge u \notin S(F^Q)}(\sum_{i=0}^{n-1} \tau_{c_i u} \cdot \frac{1}{x^i})$$

where i is the distance between a concept c and its super-concept c_i, n is the amount of super-concepts which are included in the search process, and $x \in [2, 4, 8, 16, \ldots]$ determines to which extent they are considered.

4.4 Dynamic aspects of SemAnt

SemAnt operates in a peer-to-peer environment which is dynamic by nature. Peers may join or leave the network at any time. Peers entering or leaving the network will add documents to the network or reduce the overall amount of documents. Also users may choose to add documents to a nodes repository or remove some documents. To improve the performance of the search algorithm, these changes should be reflected in the pheromone trails. If a node fails unexpectedly (possibly due to a network failure) no action can be taken to correct the pheromones. In this case, evaporation of pheromones as described in Section 4.2.6 will be responsible for the adaptation of the pheromone levels.

In the η-strategy presented by Guntsch and Middendorf [29] it is proposed that the closer a peer is to the peer where the change occurs, the greater has to be the amount of pheromones which are added or removed to account for the change. *SemAnt* employs a simplified adaptation of the η-strategy, where the distance to the change event is measured in hops between peers. The change will affect only those peers that are up to three hops away from the peer which is responsible for the modification of the overall amount of documents in the network.

If a peer P_x leaves the network in a controlled way, or documents are removed from the peers repository, it can send a 'leave'-ant L^x to peers in its vicinity to inform them that the documents will no longer be available. The closer a peer receiving L^x is to P_x, the greater the amount of removed pheromones will be. L^x is given a list C_x that contains the keywords c and total amount of documents $|D_c|$ that will be deleted. L^x is initialized with a parameter $h = 1$ denoting the distance of L^x from its originating peer P_x.
As soon as a node P_u receives L^x, it updates the pheromones on the path to the node from which L^x was sent. The amount of pheromones to evaporate is calculated as follows:

$$\tau_{cu} \leftarrow \begin{cases} \tau_{cu} - \tau_{cu} \cdot \lambda_h & \text{if } |D_c| \geq d^* \\ \tau_{cu} - \tau_{cu} \cdot \lambda_h \cdot \frac{|D_c|}{d^*} & \text{else} \end{cases}$$

where the list $\lambda = \{\lambda_1, \lambda_2, \lambda_3\}$ holds predefined factors reflecting the degree of pheromone reduction. To decrease the pheromone reduction with growing distance from P_x it has to be ensured that $\lambda_1 < \lambda_2 < \lambda_3$ and $\lambda_h \in [0, 1]$.
P_u increases the distance parameter h by one and sends the 'leave'-ant to its neighbour nodes which again update their pheromone trails and send the ant to all peers that are three hops away from P_x. L^x will expire after it has gone to a node that is three hops from its originating node.

If a node P_y joins the network, each entry τ_{cu} for each concept c and each neighbouring node P_u is initialized with a small pheromone amount τ_{init}. Joining nodes and nodes where documents are added to the repository send 'join'-ants J^y to their neighbours to inform them that new documents are available. The principle for joining peers is the same as for leaving nodes. A list with keywords and corresponding number of documents is passed to J^y, and J^y will travel to nodes P_u up to three hops away from P_y. Each P_u has to apply the modification rule above where $\lambda_h \geq 1$.

5 Design

In the course of this work a simulation environment, dubbed *SimulAnt*, is developed to evaluate the performance of the *SemAnt* algorithm. In this chapter design issues of the simulation environment are discussed. In Section 5.1 the requirements for *SimulAnt* are specified. Section 5.2 deals with *SimulAnts* architecture – the main packages and classes are described. In Section 5.3, the data storage concept for the simulator is explained. In Section 5.4 presents existing simulators for peer-to-peer routing algorithms, which were considered for adaptation or reuse in *SimulAnt*.

5.1 Introduction

In Chapter 4 the *SemAnt* algorithm which aims at routing queries through a peer-to-peer environment has been presented. Up to now, the performance of *SemAnt* has not been tested due to the lack of a suitable simulation environment. *SimulAnt*, which is developed in the course of this diploma thesis, is a new simulator for the *SemAnt* algorithm. The basic demands of *SimulAnt* will be presented in this section.

Realism A major issue in the development of *SimulAnt* is realism. The simulations should generate results which can be related to an application running on the Internet.
SemAnts domain of operation is a peer-to-peer environment. It is considered vital to test *SemAnt* in network graphs which closely correspond to the network structure of the Internet. Smallworld and power-law phenomena (see Section 2.2) can be observed in this structure. It is desirable that *SimulAnt* supports the generation and manipulation of those types of network-graphs.
The distribution of resources which are the target of queries is expected to have great influence on the simulation results. It is desired that the resources be distributed among the networks nodes in a manner which can be observed in a real environment. Again, a power-law distribution

is deemed to be a likely function for the initial allocation of resources to network nodes.
One of *SemAnt's* goals is to exploit interest-based localities – similar resources are expected to be located close to each other (i.e., at one network node). The domain from which the resources are taken should therefore support hierarchic structures. The ACM classification scheme [5], which comprises about 900 topics in four hierarchical levels is chosen to be used to annotate the resources with metadata.

Usability It should be possible for the simulator to be used by persons other than the developer, and with a minimum of prior knowledge about the internal workings or data structures of the software. This goal cannot easily be accomplished with a command line interface and configuration files. A graphical user interface shall free the user from the need to learn command line parameters and the syntax of configuration files.
Although *SimulAnt's* main field of application lies in scientific research of *SemAnt's* performance, it should also be possible to extend it to visualize the execution of the algorithm. With such a visualization it is possible to use the simulator for giving people a quick introduction to ant-based algorithms.

Platform independence A great number of simulation runs has to be performed to obtain expressive data about *SemAnt's* behaviour. Simulation runs are expected to work in fairly big networks – thousands of nodes, ten thousands of documents distributed between the nodes, and an appropriately huge amount of queries to optimize the search capabilities.
To efficiently perform these simulations, it is desirable to run them on multiple computer systems simultaneously. It is therefore desirable that the simulator can be run on host systems with different operating systems and window managers.

Comparability and Data persistence Queries in the simulation will generate loads of data which is used to analyze the performance of the tested algorithm. It is desirable that data of different simulation runs is stored so that it is available for direct comparisons between simulations which have been performed with similar parameters. The impact of changing simulation parameters can then easily be recognized and the parameters can be repeatedly tweaked to reach an optimal performance of the *SemAnt* algorithm.

5.2 System architecture

In this section the structure of the *SimulAnt* system is presented. In *SimulAnt*, five main building blocks can be identified. These are shown in 5.1. The core system, which is responsible for performing the simulation, consists of the `Network`, `Message`, `Repository` and `Storage` packages. The `UI` package provides a graphical user interface which is used to conveniently control simulation parameters and evaluate the simulation results.

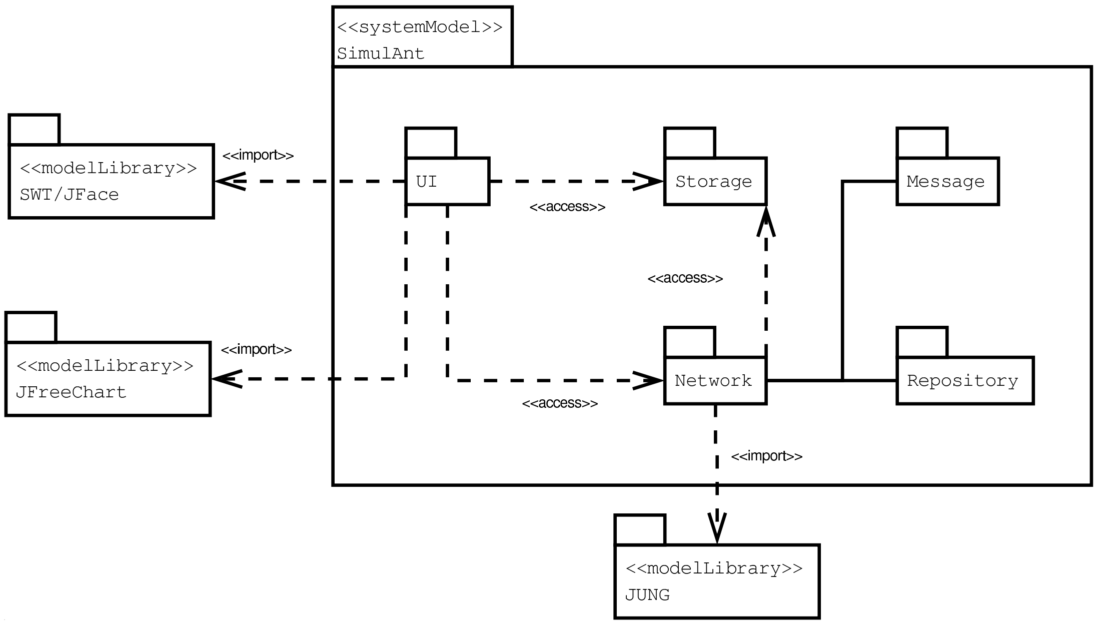


Figure 5.1: System Packages

5.2.1 Message package

Messages that are passed in the *SimulAnt* environment are represented by `Ants`. This package contains the classes that are employed for realizing the messaging mechanism, and to gather data which is used to optimize the network. The components of the `Message` package are shown in Figure 5.2.

Query This class is used to specify the concepts which are the criteria by which documents are searched. It is also used as a container in which the documents that have been found are transported back to the node where the query originates. Each `Query` object has a unique identifier

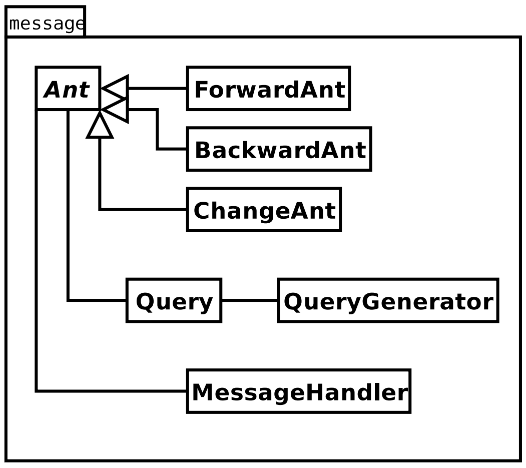


Figure 5.2: Message Package

and a timestamp of its creation. It holds several fields where statistical data, like the number of messages that have been generated to satisfy the query, and the amount of documents that have been found, are stored.
A single `Query` object may simultaneously be transported between nodes by more than one `Ant`. Upon termination of the query – its lifetime has been exceeded, or all `Ant` instances that were associated with this `Query` have either returned or died – the data which has been accumulated in the `Query` is stored for later analysis.

QueryGenerator The `QueryGenerator`'s role is to create new queries at random. At each clocktick the `QueryGenerator` randomly chooses nodes which will serve as the origin of a query. The amount of nodes which are selected can be controlled by a simulation parameter. When a node is chosen to be the origin of a query, the concepts which are the search criteria for the query are randomly selected from a list of all concepts, which are actually used to classify documents in the network.

Ant `Ant` is the base abstract class which is used for passing information (or queries) between network nodes. Each `Ant` has a sender- and receiver node and can hold a `Query` object. `Ant`s store the backtrace of nodes that have already been included in their search path. They also store the information how far they have travelled in terms of network links, and path costs.

ForwardAnt This class, which is derived from the `Ant` class, models the ants which are used to propagate a query to search for documents. Every time a `Query` is forwarded to another node, a new `ForwardAnt` is created. The data that has so far been gathered by previous `ForwardAnt`s is passed to a new instance. `ForwardAnt`s have the additional function to add nodes to the backtrace and thus update the distance information of the base `Ant` class.

BackwardAnt The `BackwardAnt` is another class derived from `Ant`. When documents matching a query have been discovered by a `ForwardAnt`, a `BackwardAnt` is created to bring the documents back to the query's originating node. The `BackwardAnt` adds the backtrack functionality to the `Ant` base class, which will – when repeatedly invoked – bring the ant back to the queries originating node.

MessageHandler The `MessageHandler`, which is implemented as a singleton [23], is responsible for the delivery of `Ant` instances to the network nodes. In the simulation `Ant`s are not actually sent via paths but are given to the `MessageHandler`, which will pass them to the receiving nodes.
Whenever an ant A travels from its current S node to the next node D in its path, a delivery time is calculated. The delivery time depends on the cost of taking the path between S and D. It is the time at which A will – in terms of clock ticks – arrive at D. A is placed in the `MessageHandler`'s queue, which is prioritized by `Ant`s' delivery times. At each clock-tick, the `MessageHandler` is notified and checks the queue whether any Ants are due for delivery. Ants whose delivery time matches the current clock-tick are passed to their destination nodes. The path from which the ant was incoming is retrieved by the node from the `Ant` object itself.

5.2.2 Network package

As its name implies, the `Network` package deals with all aspects of the simulation that are related to network graph manipulation. It provides functionalities for controlling and designing the network environments in which the *SemAnt* algorithm is tested. The `Network` package is backed by JUNG – a network/graph framework – which will be presented in Section 6.1.1. The classes contained in the `Network` package are shown in Figure 5.3.

Network This class is the central hub of all network activity. It is implemented using the singleton design pattern. It provides methods for creating network graphs by either generating random networks (small-world or powerlaw), or by adding and removing nodes and links manually. `Network`s can be stored and retrieved so that the same network can be reused in different simulation runs.

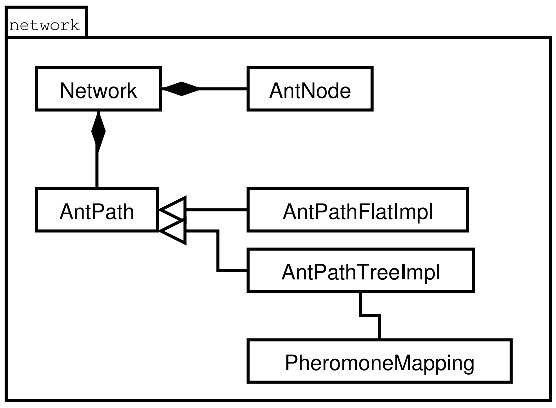


Figure 5.3: Network Package

Additionally to network management, the class provides functions to perform actions that apply to the whole network, like evaporation of pheromones, or storage of `Query` instances which have finished their lifecycle.

AntNode The `AntNode`s hold the repositories where documents are stored. They are also the source of queries and the points between which `Ant`s are travelling.
In *SemAnt's* description (see Section 4.2) it is said that the nodes hold the pheromone tables. This is due to the fact that network paths cannot usually hold any data. In the case of this simulation environment, data may be attached to the paths. This resembles the natural behaviour of ants more closely. They don't drop their pheromones in anthills but disseminate them while travelling along a path. The actual routing information is therefore located in the `AntPath` class.
The most important method of the `AntNode` (and *SimulAnt*) is the method `receiveAnt(Ant ant)`. It is responsible for retrieving documents, maintaining the routing information, and selecting paths for `ForwardAnt`s. A flowchart diagram illustrating this method is presented in Figure 5.4.

AntPath The `AntPath` represents a connection between two `AntNode`s. Each path has a field for the costs that are incurred by choosing the path, and data structures to store the pheromones that are dropped by `BackwardAnt`s. Currently, there is a flat and a hierarchical implementation of the `AntPath`. The hierarchical implementation stores pheromones for the concepts in a tree-structure, while the flat implementation is storing them in a table, where the keys are a string-representations of the concepts. The hierarchical implementation better

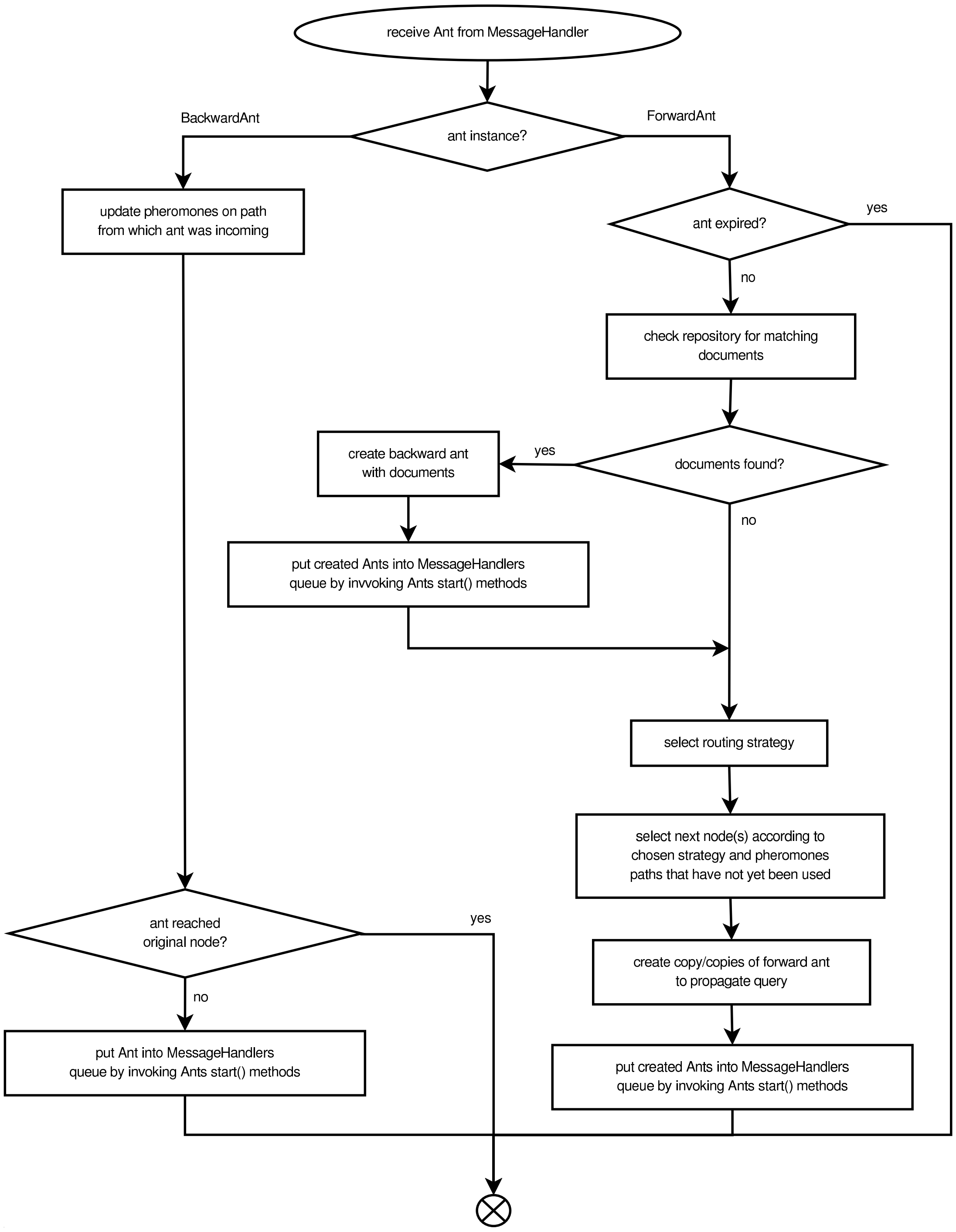

receive Ant from MessageHandler
ant instance?
BackwardAnt
ForwardAnt
update pheromones on path from which ant was incoming
ant expired?
yes
no
check repository for matching documents
documents found?
yes
create backward ant with documents
no
put created Ants into MessageHandlers queue by invvoking Ants start() methods
select routing strategy
select next node(s) according to chosen strategy and pheromones paths that have not yet been used
create copy/copies of forward ant to propagate query
put created Ants into MessageHandlers queue by invoking Ants start() methods
ant reached original node?
yes
no
put Ant into MessageHandlers queue by invoking Ants start() methods

Figure 5.4: `receiveAnt(Ant ant)` method

represents the structure of the concepts, but accessing the pheromones is more complicated and time consuming than data access in the flat implementation.
AntPaths are directed and for each path between two nodes, the existence of a path in the opposite direction is required. This requirement has two reasons:

- `BackwardAnt`s backtrack the paths that have been taken by the `ForwardAnt`s from which they were spawned. If paths between nodes were available only in a single direction, the backtracking would be impossible.
- The pheromone tables are stored in the paths rather than in the nodes. If the tables were stored in the nodes, the direction of paths would not be needed. The direction would be implicated by the pheromone tables always storing pheromones for paths leading away from the node.

When a `BackwardAnt` arrives at an `AntNode` and the pheromone amounts are updated, the pheromones on the path which is opposed to the path from which the `BackwardAnt` came (the path originally taken by the `ForwardAnt`) are increased.

5.2.3 Repository package

The `Repository` package is responsible for management of the documents which are distributed between the networks nodes. It contains the following three classes:

Document The `Document`s are the resources for which the ants are searching in the network. `Document`s are identified by their title. They hold a list of keywords which are used as search criteria. The documents' content is irrelevant for the purposes of the simulation and therefore omitted.

DocumentRepository `DocumentRepository` objects are located at each network node. `Document`s can be added to or removed from a repository. The documents are stored in a hashtable where their hash values depend on their associated concepts.

DocumentGenerator When a new network graph is generated, it needs to be populated with document instances. This process is the responsibility of the `DocumentGenerator`. Currently there is one implementation of the generator which produces an equal number of documents for each concept. The documents are distributed between the nodes following a crude approximation of a power-law distribution. The assumption is that in the network there are some nodes which show great interest in a specific topic, some nodes that are moderately interested in the topic, and nodes which have for some reason requested a document and hold it in their repository. Documents are distributed as follows:

1. Each concept is assigned primary (high interest) and secondary (moderate interest) nodes.
2. Each node may be primary node for only one concept but may be a secondary node for many concepts.
3. Each document generated is for a concept is assigned to the concepts primary node with a probability of 60 percent. With a probability of 20 percent, the document is assigned to the concepts secondary node. The remaining documents are assigned to any node in the network.

5.2.4 Storage package

The purpose of the `Storage` package is to provide access to storage systems like databases or filesystems, so that simulation properties and results which need to be kept or reused can be stored and reloaded. Currently, the `Storage` package is made up of a single class:

DBHandler Storage of data is implemented with a database (see Section 5.3). The `DBHandler` is a singleton class which provides methods for accessing a database instance. Methods for the following operations are available:

- storing and loading of networks
- storing and loading of concepts for document classification
- storing and loading of documents
- storing of simulation parameters
- storing of query statistics
- loading and preprocessing of statistical query data

5.2.5 UI package

The `UI` package implements the graphical frontend of the simulator. Its composition is shown in Figure 5.5. Its components allow creation of new simulations and network graphs, checking the progress of the simulation, issuing commands to control a running simulation, and viewing the simulation results.

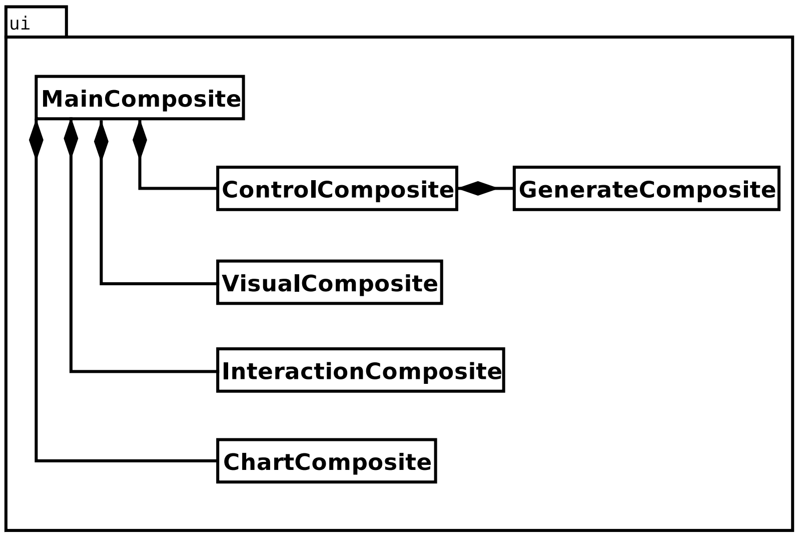


Figure 5.5: UI Package

`Composite`s are user interface elements which act as containers and are capable of containing other user interface elements. The following `Composite`s are associated with pages of the user interface:

ControlComposite The `ControlComposite` is used to set up and start new simulations. In the control composite the parameters for the *SemAnt* algorithm are specified. The values of the parameters are retained between simulation runs. When different simulations with slightly varying parameters are run, not all of the parameters need to be specified again, but only the parameters which need to be changed can be updated.
Creation of new network graphs is done via a dialog where the type and parameters of the new network can be specified.
A screenshot of the form created by the ControlComposite is shown in Figure 5.6.

InteractionComposite In the `InteractionComposite`, commands which affect the simulation during runtime can be issued. Commands which will be executed are presented in a list. They each have a time at which clocktick they will be processed. Commands include:

- halting a simulation, so that current pheromone levels, document repositories contents, and network links can be checked
- adding and removing nodes from the network graph
- adding and removing documents from a nodes repository

Sets of commands can be stored and reloaded so that they can be reused in subsequent simulation runs.
A screenshot of the command interface is shown in Figure 5.7.

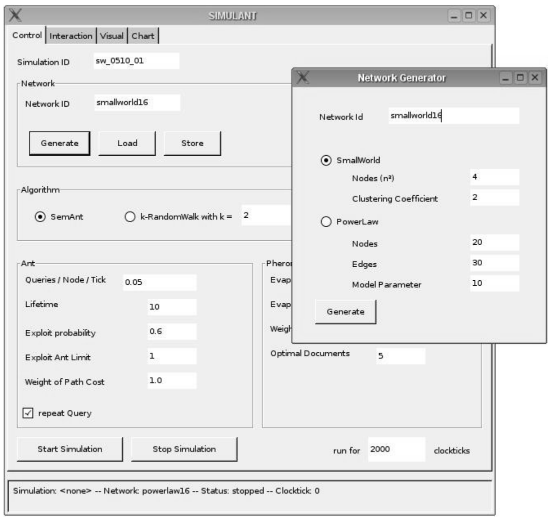
VisualComposite The `VisualComposite` allows to check the state of the network. It shows a graphical representation of the network graph (this is probably useful only for very small networks). Nodes can be selected by clicking on them in the graph or choosing them from a list of all nodes in the network. For selected nodes information about its links, pheromones on the links, and documents in the nodes repository can be displayed.
A screenshot is shown in Figure 5.8.

ChartComposite The `ChartComposite` is used to display a graphical analysis of simulation results. By selecting the identifiers of finished simulations and timing parameters, line charts of the network optimization performed in a simulation are generated. It is possible to select multiple simulations to enable a direct comparison between simulations with different parameters.
A screenshot of the charting window is presented in Figure 5.9.

5.2.6 Utility Classes

In the following, classes which did not fit in the above packages or are not used at runtime of the simulation will be described:

Simulation The `Simulation` singleton class implements the *main()* method of *SimulAnt*. It is responsible for instantiation of the *Network* and user interface. `Simulation` manages the current state of the simulation (running/ paused/ stopped) and is responsible for loading and storing the simulation parameters. A sequence diagram showing the events triggered by the `Simulation` class is presented in Figure 5.10.



Simulation ID sw_0510_01

Network ID smallworld16

(◉) SemAnt ( ) k-RandomWalk with k = 2

Queries / Node / Tick 0.05

Lifetime 10

Exploit probability 0.6

Exploit Ant Limit 1

Weight of Path Cost 1.0

☑ repeat Query

Optimal Documents 5

run for 2000 clockticks

Network Id smallworld16

(◉) SmallWorld

Nodes (n²) 4

Clustering Coefficient 2

( ) PowerLaw

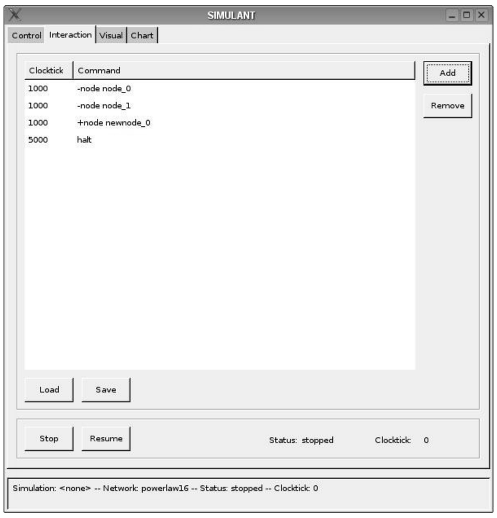
Nodes 20

Edges 30

Model Parameter 10

Simulation: <none> -- Network: powerlaw16 -- Status: stopped -- Clocktick: 0

Figure 5.6: Screenshot: Control

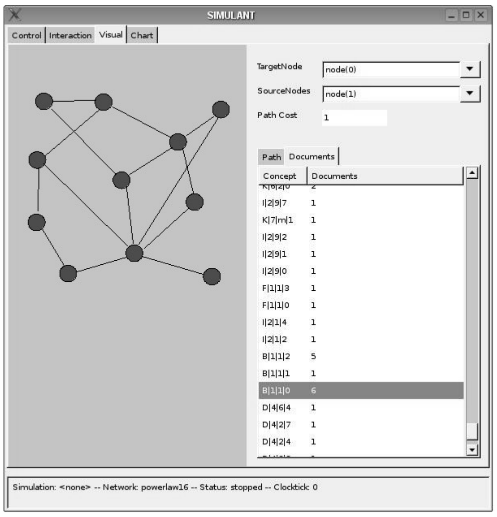


Clocktick	Command
1000	-node node_0
1000	-node node_1
1000	+node newnode_0
5000	halt

Status: stopped Clocktick: 0

Simulation: <none> -- Network: powerlaw16 -- Status: stopped -- Clocktick: 0

Figure 5.7: Screenshot: Interaction



TargetNode node(0)

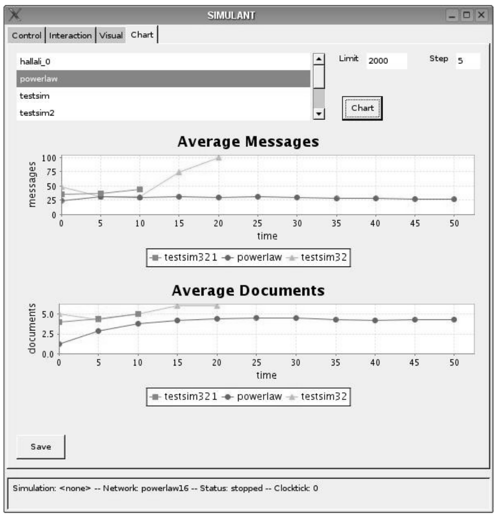
SourceNodes node(1)

Path Cost 1

Concept	Documents
I|2|9|7	1
K|7|m|1	1
I|2|9|2	1
I|2|9|1	1
I|2|9|0	1
F|1|1|3	1
F|1|1|0	1
I|2|1|4	1
I|2|1|2	1
B|1|1|2	5
B|1|1|1	1
B|1|1|0	6
D|4|6|4	1
D|4|2|7	1
D|4|2|4	1

Simulation: <none> -- Network: powerlaw16 -- Status: stopped -- Clocktick: 0

Figure 5.8: Screenshot: Visual



Limit 2000 Step 5

Simulation: <none> -- Network: powerlaw16 -- Status: stopped -- Clocktick: 0

Figure 5.9: Screenshot: Chart

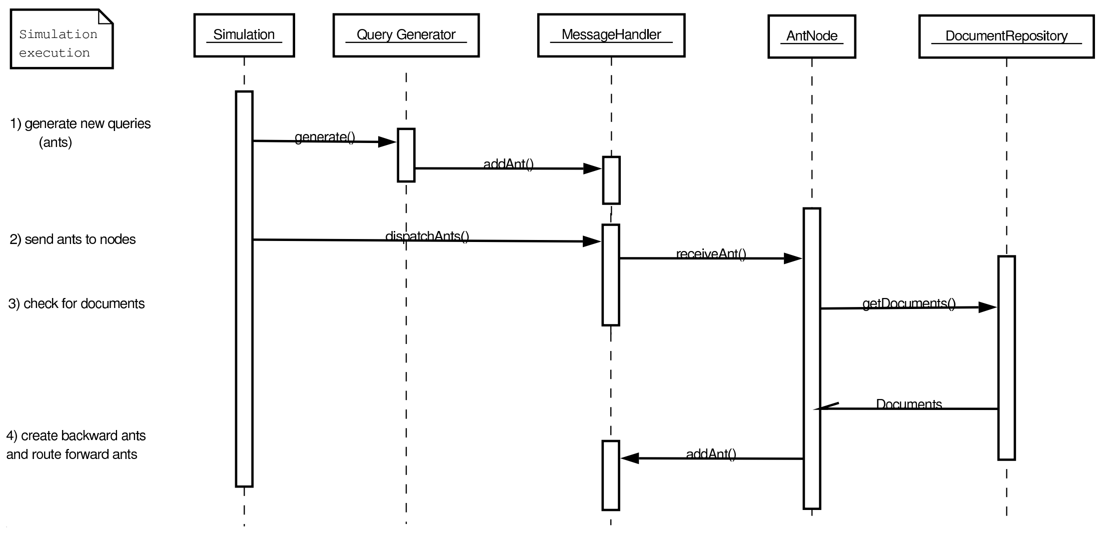


Figure 5.10: Simulation sequence diagram

ACM2DB The documents which are used for the simulation are associated with concepts taken from the ACM classification scheme. ACMs classification tree is available as an RDF schema [66]. The ACM classification needs to be imported into *SimulAnts* data structures where a flat representation of the taxonomies is implemented. The `ACM2DB` class is used to parse the RDF schema, extract the keywords and insert them into *SimulAnts* data storage.

Chart2SWT The charts displaying the simulation results are generated in a Java Swing context by the `jfreechart` (see Section 6.1.3) library. The `Chart2SWT` class provides a static method to convert the Swing graphics into the SWT picture format, which is used by *SimulAnt*.

5.3 Data Storage

During the simulations, a huge amount of data about the query routes and routing performance will be accumulated. This data has to be stored to be able to evaluate the performance of the algorithm and to compare the impact of different model parameters. This data could either be stored directly in a file, for example as comma separated values, or in a database.

Writing the data to a file would be easier to implement but has the drawback that evaluation would have to be done by employing spreadsheet software. The usage of spreadsheets when dealing with massive amounts of data is awkward at its best and useless at its worst. The shortcomings of spreadsheet software are that they only allow a certain amount of lines to be used, thus preventing the evaluation of huge amounts of data. Furthermore, data cannot be easily manipulated. One would have to resort to time consuming import/export and copy/paste functions to structure the data to obtain the desired results.

Using a database to store the simulation data takes a little more design and implementation effort but it's well worth the cost. The advantages of using a database (and the accompanying query language) are the following:

- Graph data and model parameters can be stored and reused in further simulations without having to resort to multiple configuration files.
- Model data and result data can be stored together without having to invent a data structure for storage (XML...).
- Results of different simulations can be directly compared
- Scaling of the results (i.e., stretching of the time-axis) is easily possible.

5.3.1 Choice of Database

Once it was clear that a database will be used for storage a decision about the database product had to be made. I decided on using the MySQL 4.1.14 database [45] for the following reasons:

- MySQL is distributed under GNU GPL and thus it is free to use.
- Simulations will be done on different operating systems and MySQL runs on Linux as well as on Windows systems.
- Installation and administration of MySQL is *very* simple compared to products such as Oracle or DB/2.
- During the simulation all access to the database will be write operations. Read operations involving joins over multiple tables and huge data amounts will not be necessary or are not time critical since they will be performed after the simulation has finished. Thus, an industrial strength database which is optimized for such conditions is necessary.

5.3.2 Data Model

The database schema provides for storing the simulations network graph, the network parameters of each simulation, and statistical data generated by sending and receiving query messages. The model is displayed as an EER-Diagram in Figure 5.11. The table definitions are given in Tables 5.1 to 5.8. The usage of each table is outlined in the following:

Network The 'network' table is used for storing the name of the network. By referencing the network by its name it may be used in simulations with different model parameters so that the impact of the changed parameters can be evaluated

Node The 'node' table is used for storing all nodes that are initially available in a network.

Path The 'path' table stores the paths between network nodes and the cost associated with travelling the path.

Taxonomy The 'taxonomy' table stores the identifiers for the taxonomies.

Concept The 'concept' table holds all keywords associated with a taxonomy. The keyword table is not structured hierarchically. Instead, a flat representation of the taxonomies hierarchy is used. This is poor database design, but emphasis was laid on execution speed of the simulation and a flat representation of the concepts has proven to be much faster than a tree based approach.

Document The 'document' table holds all documents that are initially available in a network. Documents are assigned to nodes where they can be located, and concepts which can be used to query for them.

Simulation The 'simulation' table is used to store the parameters of a single simulation run.

Query The 'query' table stores data about each query sent during a simulation run. The query's starting time as well as the number of messages it generates, and how many of those messages actually contributed to returning documents are stored. After finishing a simulation, this data may be used for analyzing the behaviour of the algorithm in a certain network with a certain set of model parameters.

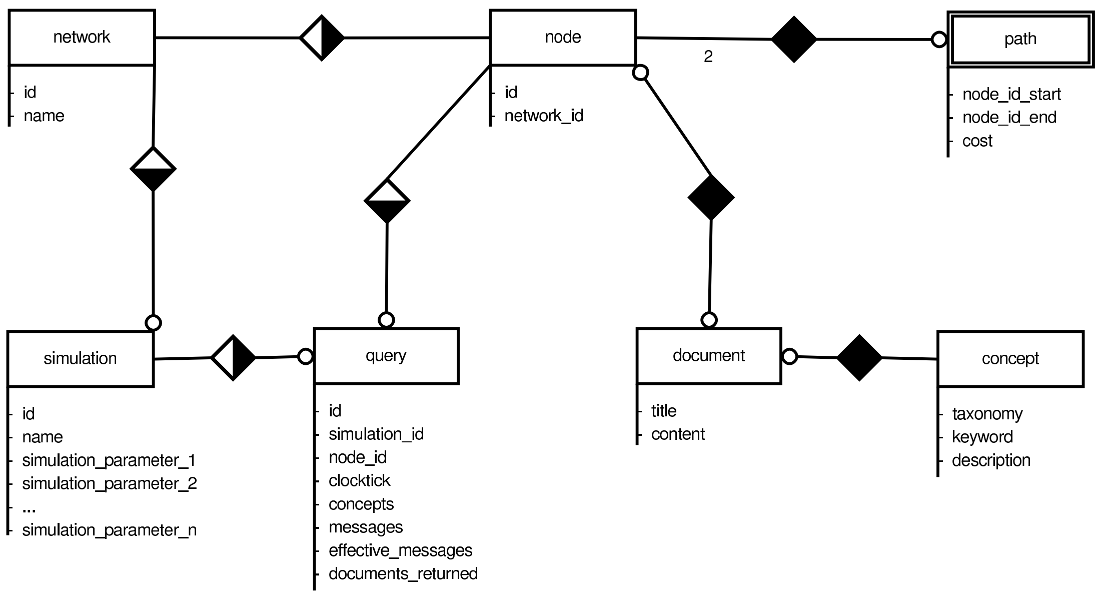


Figure 5.11: Database EER-Model

column	**type**	**null**	**key**	**comment**
id	int	N	PK	
name	varchar(32)	N		

Table 5.1: Network

column	**type**	**null**	**key**	**comment**
id	int	N	PK	
network_id	int	N	FK	references network.id

Table 5.2: Node

column	**type**	**null**	**key**	**comment**
node_id_start	int	N	PK, FK	references node.id
node_id_end	int	N	PK, FK	references node.id
cost	double	Y		default 1.0

Table 5.3: Path

column	**type**	**null**	**key**	**comment**
id	int	N	PK	
name	varchar(32)	N		

Table 5.4: Taxonomy

column	**type**	**null**	**key**	**comment**
id	int	N	PK	
taxonomy_id	int	N	FK	references taxonomy.id
keyword	varchar(256)	N		flat representation, separated by '—'
description	varchar(256)	Y		

Table 5.5: Concept

column	**type**	**null**	**key**	**comment**
id	int	N	PK	
title	varchar(256)	N		
content	varchar(256)	Y		

Table 5.6: Document

column	**type**	**null**	**key**	**comment**
id	int	N	PK	
network_id	int	N	FK	references network.id
name	varchar(32)	N	U	
algorithm	int	N		0: SemAnt 1: k-Random
optimal_documents	int	N		semant parameter
path_cost_weight	float	N		semant parameter
document_weight	float	N		semant parameter
exploit_probability	float	N		semant parameter
exploit_ant_limit	int	N		semant parameter
use_hierarchy	boolean	N		semant parameter
ant_lifetime	int	N		semant parameter
evaporation_rate	float	N		semant parameter
evaporation_ticks	int	N		semant parameter
answer_repeat	boolean	N		semant parameter
query_probability	float	N		semant parameter
k_random	int	N		k-random parameter

Table 5.7: Simulation

column	type	null	key	comment
simulation_id	int	N	PK, FK	references simulation.id
id	int	N	PK	
node_id	int	N	FK	references node.id, node that created query
clocktick	int	N		start time
messages	int	N		total amount of messages
effective_messages	int	N		messages useful for document retrieval
documents_returned	int	N		amount of retrieved documents

Table 5.8: Query

5.4 Related Work

Before deciding upon creating a new framework for simulating the *SemAnt* algorithm, a few other simulation platforms were evaluated to find out whether they can be reused or adapted to suit the needs of the *SemAnt* simulation. In this section two such simulators are described. In each of those two some shortcomings were found that made them unsuitable for testing *SemAnts* performance.

5.4.1 Anthill

The Anthill Project [6], which has been developed at the University of Bologna, aims at providing researchers with a framework to implement and test ant-based peer-to-peer algorithms. Its implementation allows for testing algorithms in a real IP network, or in a simulated network on a single host.
AntHills main building blocks are *Nests* and *Ants*. Similar to *SimulAnt's* `AntNodes`, the *Nests* represent nodes in a peer-to-peer network and contain documents or offer computing functionalities. The interface of a *Nest* allows for:

- addition of documents
- posing requests

- adding or removing connections to other nests
- getting connections to neighbouring nests

The components of a *Nest* are the *document storage*, an *ant manager* which is used to schedule computations performed by ants, and a *gateway* which manages the communication between *Nests* by sending and recieving ants.

Ants which are used to gather information and optimize the network need to implement a single `run()` method. The ant's behaviour is modeled in this method by the user of the AntHill Framework. The `run()` method is invoked by the ant manager each time an *Ant* visits a *Nest*. The following operations are allowed by the ant manager:

- move the ant to another nest
- query and update the document storage
- add new nests as neighbours
- get the set of neighbours of the nest
- retrieve and update of pheromone informations
- notify the nest about a response to a query

AntHill contains an evaluation framework which collects data about the algorithms behaviour. This data includes the number of requests that have been generated, the number of requests which have been completed satisfactorily, the number of ant movements and the amount of documents which have been copied between nests.

An evaluation of AntHill showed that it was unsuitable as a base on which to implement *SimulAnt* because out-of-memory errors occurred when algorithms were run for a few thousand iterations.

5.4.2 Simulator for INGA

The INGA (Interest-based Node Group Algorithm) algorithm devised by C. Tempich et al. [36] is applied for semantic search in a peer-to-peer environment. It works by analyzing successfully answered queries at each node the query passes through, and creating network shortcuts to nodes which have

been helpful in procuring the desired documents.
When building the routing indices by analyzing the queries, the nodes which helped finding the documents may assume four different roles:

- *Content Providers* are nodes which have successfully answered a query or semantically similar query.
- *Recommenders* are peers which have posed a similar query in the past, and thus are assumed to have knowledge how to find the desired documents.
- *Bootstrapping Peers* are peers which have established a network to other peers and cover a variety of topics. These peers make up a bootstrapping network.
- *Default Peers* are random peers which do not meet any of the above criteria.

Routing queries is done by forwarding them, depending on availability, to content providers, recommenders or bootstrapping peers. If none of the above are present, the query is sent to a default peer.

Each INGA peer is composed of the following components:

- A *network component* is used to provide basic network functionalities and manage peer identification.
- The *content database* stores the nodes resources and may be queried to retrieve them.
- The *shortcut management component* manages the routing indices by extracting information from the query. Routing indices are built only for topics in which the node itself is interested.
- A *routing logic component* is responsible for selecting peers to which the own queries, or queries forwarded by other peers, are relayed. Peers are selected depending on the information which the local peer has already gathered about the specific or similar queries.

The simulation engine for the INGA algorithm has been evaluated. Like *SimulAnt*, it is also built on the JUNG (see Section 6.1.1) framework but does not exploit JUNGs feature of user data annotation. Therefore, the mapping of peer information to network nodes is rather awkward. A further design flaw, which made it unsuitable for adaptation to *SimulAnts* needs, is that the timing information for the algorithm was derived from the system clock and therefore simulation results are not reproducible on different host computers, or, in extreme cases, even the same host.

6 Implementation

In this chapter, the components and tools which were used to create the *SimulAnt*-environment are briefly discussed. Section 6.1 deals with Java libraries that have been utilized to create *SimulAnt*. In Section 6.2, the main software applications which were used to build *SimulAnt* will be presented.

6.1 Components

In this section the software libraries which constitute the main building blocks of the simulation environment will be presented. These components are the JUNG framework for network-graph manipulation (Section 6.1.1), the Java-SWT for creating graphical interfaces (Section 6.1.2), and JFreeChart which has been employed to obtain visual representations of simulation results (Section 6.1.3).

6.1.1 JUNG

The **J**ava **U**niversal **N**etwork/**G**raph (JUNG) framework [49], which was first released in August 2003, is an open source library that was created with the intent of providing a common framework for graph and network manipulation, visualization and analysis.

Several different types of graphs or networks can be created by using very basic manipulation methods. Graph types that are supported include directed and undirected graphs, multigraphs and hypergraphs. Graphs can be edited by calling the appropriate *add* or *remove* methods for edges and vertices on the graph instances. Vertices offer methods to query for their edges, neighbor vertices or – in the case of directed graphs – their predecessors or successors. Edges provide methods to access the vertices they are connected to.

Network entities may be annotated with user data. User data can be any Java object. The data is attached to the network entity as key-value pairs.
In addition, JUNG also offers functionalities to generate different types of networks: random networks, smallworld networks, powerlaw networks, and others.
JUNG has a visual component with which Swing images of the current network can be displayed. Nodes and edges in the image can be annotated with user data. The visual component supports selection and dragging of nodes. In the case of *SimulAnt*, this feature has been found to be mostly useless though. The Swing style image needs to be converted to fit into the SWT framework, which is a rather time consuming task, and since *SimulAnt* is supposed to run with networks with thousands of nodes the visualization would be too crowded to be of any real use.

6.1.2 SWT / JFace

When programming graphical user interfaces in Java there are two different technologies which might be used: AWT/Swing [62, 63] which are portable Java APIs from the Java Foundation Classes and SWT/JFace [65, 33] which are the graphical components on which the Eclipse UI is built. There is some tension between UI programmers whether AWT (Abstract Widget Toolkit) or SWT (Standard Widget Toolkit) is better. Since *SimulAnt* has been built using the Eclipse Framework the decision was to use Eclipses graphical components as well so that *SimulAnt* could be easily embedded into the platform as plug-in.

6.1.2.1 SWT

The AWT technology is a portable API that uses operating sytem widgets only if they are supported by all operating systems or window managers. Widgets which are not included in this least common denominator are emulated. AWT therefore needs lots of memory and has a comparatively high response time.
IBM found the AWT/Swing technologies to be too slow to build the Eclipse platform with it and therfore, SWT has been developed [28]. SWT is using different native libraries for each supported operating system and utilises native widgets whereever possible. Widgets which are not supported by the operating system on which the SWT application is running are emulated. For example, the Tree widget is native in the Windows OS but is emulated in the Motif GUI toolkit. At the cost of portability, which is an arguable case for some Java

applications anyway, response time and embedding in the operating systems look-and-feel is superior to that of AWT based products.

The three building blocks of any SWT based UI are the *Shell*, *Display* and *Widget*. The Shell is the operating systems instance of a window. Styles can be set for a Shell so that it can take the appearance of different commonly used windows: empty windows, dialogs, modal pop-ups, etc.
The Display is used to relay actions between the thread running the user interface and threads that are realizing the core functionality of the system.
Widgets are the user interface elements like buttons, labels, and text areas which are placed on the Shell instance.

The correlation between Shell, Display and Widgets is shown from three different angles in Figure 6.1.

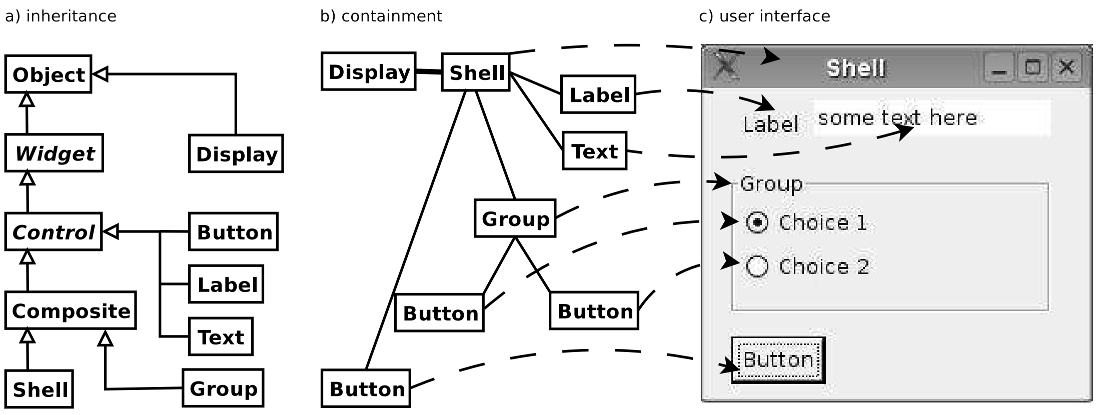


Figure 6.1: SWT Components

6.1.2.2 JFace

SWT provides the programmer with a raw set of widgets only. It is lacking support for higher level interactions or model based approaches to GUI building like those that can be found in the Swing API. To support easy integration of the user interface with the application-model of the software, the JFace API has been developed. JFace is built on top of SWT but doesn't hide SWT functionality from the programmer. Each JFace component offers access to the basic SWT widgets from which it has been constructed.
The aim of the Jface API is to free the programmer from the more tedious and repetitive tasks of building a user interface and allow him or her to put more

focus on designing the softwares interaction with the end-user.
Some concepts that are realized by the JFace API are listed below:

- *Windows* allow easy creation
- *Viewers* can be used to populate tree and list-like widgets and syncronize them with the data structures that hold their raw content.
- *Actions* are used to define the interface's behaviour and to link functionality to different places (like toolbars and menus) in the user interface.
- *Registries* are employed to manage image and font handling.
- *Wizards* and *Dialogs* support the programmer by allowing creation of more complex user interactions.

6.1.3 JFreeChart

To get a visual representation of the simulation results, some charting capability had to be integrated into *SimulAnt*. Several libraries providing charting functionality are freely available. Charting libraries released under LGPL include the JFreeChart project, JOpenChart, the JCCK (Java Chart Construction Kit), JChart2D and others.

JFreeChart [48] is a very comprehensive charting tool. It supports a lot of different chart types including some that are not found in other charting libraries. Charts supported by JFreeChart are:

- line and area charts
- 2D and 3D pie charts
- bar charts
- scatter plots and bubble charts
- time series, high/low/open/close charts and candle stick charts
- combination charts
- Pareto charts
- Gantt charts

Charts created with JFreeChart can easily be integrated into Java-Swing environments. To display them in an SWT context, the image data can be copied from AWT to SWT using the source code shown in Figure 6.2. Charts may not only be displayed inside Java user interfaces but also be exported to different graphics formats. JFreeCharts directly supports PNG and JPEG graphics. PDF and SVG graphics may be created by including the appropriate additional libraries (iText or Batik).

```
// Color adjustment
Color swtBackground = parent.getBackground();
java.awt.Color awtBackground = new java.awt.Color(swtBackground.getRed(),
                                          swtBackground.getGreen(),
                                          swtBackground.getBlue());
chart.setBackgroundPaint(awtBackground);
// Draw the chart in an AWT buffered image
BufferedImage bufferedImage = chart.createBufferedImage(width, height, null);
// Get the data buffer of the image
DataBuffer buffer = bufferedImage.getRaster().getDataBuffer();
DataBufferInt intBuffer = (DataBufferInt) buffer;
// Copy the data from the AWT buffer to a SWT buffer
PaletteData paletteData = new PaletteData(0x00FF0000, 0x0000FF00, 0x000000FF);
ImageData imageData = new ImageData(width, height, 32, paletteData);
for (int bank = 0; bank < intBuffer.getNumBanks(); bank++) {
    int[] bankData = intBuffer.getData(bank);
    imageData.setPixels(0, bank, bankData.length, bankData, 0);
}
// Create an SWT image
Image swtImage = new Image(parent.getDisplay(), imageData);
```

Figure 6.2: AWT to SWT image conversion

According to Walter Goh [24], the use of JFreeGraph may incur some performance issues. In most cases, bad performance seems to be caused by a bug in Java or wrong application of the API. Since (1) the charting is mainly done after the simulation run has finished and (2) no realtime charting is needed, these performance problems do not apply to *SimulAnt*.

6.2 Production environment

A brief description of the tools and applications used to create the *SimulAnt* system will be given in this section. These software applications include the

Eclipse project (Section 6.2.1), the Jigloo Eclipse plug-in which supports the creation of graphical user interfaces (Section 6.2.2) and the Aqua Datastudio (Section 6.2.3), which is a comfortable frontend for accessing and manipulating databases.

6.2.1 Eclipse Platform

The Eclipse [64] platform serves two purposes. First, it is a framework designed for the purpose of building integrated development environments (IDEs) and other software applications. Second, it is a fully featured Java development environment in its own right.

6.2.1.1 Eclipse as Framework

When Eclipse is used as a framework to develop new applications, it can be seen as an integration point where programs or parts of programs are able to interact. Eclipse has the functionality to discover and integrate programs which are called plug-ins. By discovering and loading these plug-ins at startup, the platform can be configured to offer such functionality as the user intends it to have. This functionality is not restricted to programming and software development only. One could imagine plug-ins that facilitate database access, typesetting, data modeling, etc. This thesis, for example, is written using Eclipse and TexLipse [58], a plug-in which supports the creation of LaTeXdocuments.

Plug-ins are coded in Java and adhere to the OSGi specification [50]. To integrate with the platform, they utilize extension points provided by Eclipse or other plug-ins. By using these extension points, a plug-in can add new functionality to the existing one. Each plug-in may in its manifest file also declare points where plug-ins written by other parties may insert functionality. Bigger applications coded to run in the Eclipse environment (like the Eclipse platform itself) are typically distributed as multiple interacting plug-ins.

A very general overview of the platform – which also provides its own APIs known as Eclipse Software Development Kit (SDK) – is shown in Figure 6.3. A short description of each of these components will be given in the following paragraphs:

Platform Runtime The Platform Runtime, like all other components of Eclipse, is built as a plug-in. During the startup of the Eclipse frame-

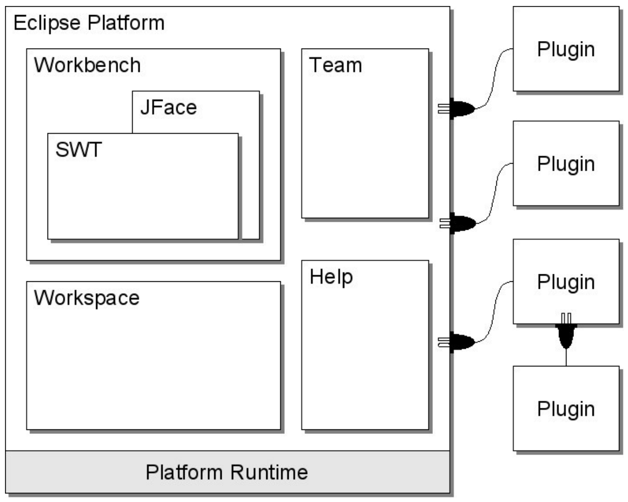


Figure 6.3: Eclipse Platform

work, it is the Platform Runtime's responsibility to find all available other plug-ins and check their manifests. With the information gathered from the plug-in manifests, the Platform builds a runtime plug-in registry by matching plug-in extension points with extension declarations. Plug-ins themselves are loaded only when their functionality is actually needed by the user, thus reducing the memory load if several large plug-ins are installed but are not used concurrently.

Workspace The Workspace is where user's projects are stored. The projects may be located anywhere in the filesystem but are subdirectories of a designated workspace directory by default. Files in a project are referred to as *resources*. The Workspace provides for resource annotation which may be used to tag resources with todo-items, breakpoints, bookmarks and others. A history of resource changes is kept, so that previous versions of resources may be used for rollback, or to recover resources that have accidentally been deleted. Upon modification of resources or resource sets, events are generated and propagated to all plug-ins registering with these events. The resource change events can then be used to keep user interface elements up do date or detect whether different tools are currently working on the same set of resources.

Workbench The Workbench is Eclipse's user interface. It displays a collection of Viewers and Editors. Editors are used to open, edit and save objects, whereas viewers display information about the objects that are manipulated by the user. Unlike other editor applications, the Eclipse editors may be tightly coupled with the workbench and add to the functionality of the workbench menus and toolbars if they are active. Different editors and views may be needed to perform certain tasks. The visibility and arrangement of these user interface elements may be stored in Perspectives. The user can switch between different Perspectives to completely change the layout of the UI. The workbench is built upon the SWT and JFace APIs which have already been presented in Section 6.1.2.

Team Projects in the workspace may be associated with version and configuration control repositories such as CVS [51] or Subversion [13]. Different version control products have different workflows. Rather than forcing its own views of version control on team repository providers, Eclipse offers them a set of basic hooks so that they may take appropriate actions if certain resources are modified. These hooks provide support for both optimistic and pessimistic versioning models. The Eclipse platform itself supports team versioning with CVS via *pserver*, *ssh* and *extssh* connections.

Help Plug-ins are usually distributed with their manual pages in a subdirectory of the plug-ins bundle. Larger projects may include their documentation as a separate plug-in. When creating documentation for a project, one has to supply both the raw content, in form of HTML pages, and the navigation structure as XML. Content and structure are kept separate so that existing online documentations can easily be extended at defined insertion points. The Eclipse platform provides its own content server to resolve links between documentation pages supplied by different plug-ins and display all available documentations as a tree structure of online-books with their respective topic subtrees.

Plug-in The plug-ins are software components which use and offer extension points to add to the functionality of the Eclipse Platform.

6.2.1.2 Eclipse as Java Development Environment

The Software Development Kit consists not only of APIs to create new applications on the Eclipse Platform but also includes Java Development Tooling (JDT). JDT is a series of plug-ins which aim at supporting software development in Java in general. The JDT implementation is divided in two groups of plug-ins: User interface and core functionality. This division makes it possible that core functionality can be used by other plug-ins without having to worry about the interface.

In the following, a list of key features of the JDT which greatly aid programming in Java is presented.

Editor The sourcecode editor offers functionality which allows quick, clean and transparent creation of Java code:

- highlighting of syntax, keywords and javadoc comments
- code completion
- code folding
- annotation of bookmarks, changed lines of code, compiler errors, open tasks and others in a sidebar
- outline of the resource in the current editor (updated while the resource is edited)
- formatting of code according to templates
- automatic generation of constructors, getter/setter- methods, method implementations required by abstract classes (including interfaces)
- comparison of current resource version with previous versions

Navigation Several search mechanisms and viewers allow the programmer to quickly find and access packages, classes, methods and fields:

- view of the project as filesystem or Java-packages (flat or hierarchic)
- view of type hierarchies and call hierarchies
- searching for references and declarations of types, fields and methods
- searching for read/write access to fields and variables
- finding the last edit position

Refactoring Several JDT features support quick and safe restructuring of the software projects. When refactoring sourcecode, JDT will ensure that classes referencing or referenced by the changed code are also updated to reflect these changes and retain a correct program source.

- renaming and moving of members
- pulling up or pushing down members in the class hierarchy
- changing method signatures
- inlining of methods, fields and local variables
- interface extraction

Compilation Each Java project is assigned a project builder by JDT. The project builder can be configured to compile the project on demand, or whenever a source file has been changed. With the initial compilation

process, a dependency graph between the sources is created in memory. The builder supports incremental compilation in such a way that, after a full compilation of the project, only those sources which are dependent on edited files will be compiled.
If the compiler encounters errors in the sourcecode, the code is annotated with error messages and they are also displayed in a special "Problems"-view.
Of course it is also possible to integrate Apache Ant [3] into the Eclipse platform and use Ant-buildfiles to compile and package the project.

Debugging A well implemented logging mechanism is the key to successful tracking of programming errors, but sometimes a runtime debugging tool will help to find bugs that could not be located with logging output.

- view of threads and execution stacks
- stepping through breakpoints in the sourcecode, breakpoints may be conditional
- viewing and modifying fields and variables
- dynamic class reloading if possible in the Java VM

6.2.2 Jigloo

Since building graphical user interfaces requires large amounts of standardized code, it is virtually impossible to program a decent UI without software support. Jigloo [12] is a plug-in for the Eclipse framework which assists the creation of graphical user interfaces. Jigloo supports creation of both AWT/Swing and SWT/JFace style UIs and also converts UIs between those two.

Building the user interfaces is done in two editors: the sourcecode editor and the form editor. Inside the form editor the interface can be created by dragging, dropping and correctly placing UI-elements. Two-way editing is supported by Jigloo so that any changes made in the form editor will instantly be updated in the appropriate source code sections and vice versa.The style of the produced code can be controlled by the programmer at a very fine grained level. Different styles by other GUI-building applications will be recognized by Jigloo and may be used as code templates.

Two key aspects which have to be considered when building user interfaces are the layout and handling of the interface. Jigloo supports a wide array of component layouts including JGoodies FormLayout, Clearthoughts TableLayout, AWT GridBag and SWT Form and Absolute layouts.

6.2.3 Aqua Datastudio

Accessing databases with the associated command line interfaces is a cumbersome task which can be greatly aided by more advanced database manipulation tools. Since most Eclipse plug-ins for database support that have been considered for building *SimulAnt* are either commercial software or seemed to be too simple or buggy, the tool of choice has become AquaFold's Datastudio [4].
Datastudio offers support for most common databases. It visually assists creation of databases and schema.
When typing SQL statements, a lot of queries can be produced with a few keystrokes. Templates for producing insert-, update-, and select-statements can be invoked by pressing hotkeys. As soon as the tables have been selected for those statements, the column names can be automatically inserted into selection and insert-statements.
Aqua Datastudio offers a script-generator to create scripts from an existing database structure. The content and style of the scripts may be controlled at a very fine grained level.

7 Evaluation of the algorithm

This chapter presents some of the insights gained by evaluating the results of several simulation runs of *SemAnt*. In the first part of Section 7.1, the efficiency of *SemAnt* is compared to the efficiency of a *k-random-walk* algorithm. The second part of this section shows how using the hierarchic classification of the documents can affect the search process. Section 7.2 deals with the distribution of pheromones across the network, and the effects the distribution has on the behaviour of the ants.

7.1 Efficiency

The following experiments are conducted using a Kleinberg-small-world network (see Section 2.2.1) with 1024 nodes and a clustering coefficient of 2. The path cost of all links in the network is set to 1.0. The documents distributed between the nodes of the network are classified using the ACM Computing Classification System [5]. Each document is associated with one leaf concept of the ACM System. Each topic from the system is assigned an equal number of documents. Each node holds roughly the same amount of documents. In total 30940 documents are distributed throughout the network. The distribution of the documents between the nodes is governed by a parameter P_{expert}. P_{expert} determines the amount of similar documents stored at each node. Two documents are similar if they are instances of a sub-concept of the same third-level concept of the ACM system. The remaining documents are instances of random concepts. $P_{expert} = 100\%$ means that all documents stored at a node are similar, while with a setting of $P_{expert} = 0\%$ the documents are distributed between the nodes totally at random.
A global clock is used to synchronize the ants movements. At each tick of the clock, each node in the network has a 10 percent chance to create a new search query.

The parameter settings which are used for the *SemAnt* algorithm are shown in Table 7.1.

ρ	evaporation rate	0.07
T_{max}	forward ant lifetime	25
ω_e	probability of exploiting vs. exploring	0.85
D_{max}	maximum number of documents	10
ω_d	weight of document quantity vs. link costs	0.5
β	weight of link costs	1

Table 7.1: *SemAnt* parameters

The following metrics are used for the experiments:

- *Hit rate* is defined as the number of documents that are retrieved for each query within a given period of time.
- *Resource usage* is defined as the number of messages that are sent for each query within a given period of time. The resource usage includes both forward and backward ants.
- *Efficiency* is the ratio of resource usage to hit rate. By dividing the number of sent messages by the amount of documents found the average number of links travelled to find one document is obtained.

7.1.1 Reference algorithm

Lacking any implementations of other informed algorithms in *SimulAnt*, the performance of *SemAnt* is compared to that of an uninformed search algorithm. As a reference algorithm, a random walk strategy [37] is employed. In a random walk algorithm, a message is forwarded at each noda along a randomly chosen outgoing link until the destination node of the message has been reached. Compared to broadcasting strategies for resource location, a random walk approach is likely to find a lower amount of documents, but at the same time to consume considerably less network resources. The response time of the random walk algorithm is higher than that of a broadcast, but it can be reduced by simultaneously sending multiple walkers.
Pseudo-code for a random walk with k walkers is shown in Figure 7.1.

In the case of *SimulAnt* the k-random-walk is implemented by starting k ants from the source node and routing them to a random neighbour of the current node. Since the destination node (the node which stores the most documents

```
void kRandomWalk(Graph g, int k, Node startNode, Node destinationNode) {
    Node n = startNode;
      while (!n.equals(destinationNode)) {
         for (int i = 0; i < k; i++) {
            List neighbours = n.getNeighbours();
            if (n.size() != 0) {
            n = neighbours.get((int)Math.random()*nodeList.size());
         }
      }
   }
}
```

Figure 7.1: k-random-walk algorithm

for a query) is unknown, the walkers continue until their allotted lifetime has expired.

An advantage of using a random walk algorithm to compare the simulation results is, that the number of messages used and the number of documents returned by the random walk can be set by tweaking the k and lifetime parameters. Hence it is possible to send roughly the same amount of messages ai in the algorithm which is tested (i.e., *SemAnt*), and to compare the number of documents retrieved.

7.1.2 SemAnt vs. random-walk

In this experiment, the performance of *SemAnt* is compared to the efficiency of the k-random-walk algorithm which was presented in the previous section. Since it is easily possible to tune the random-walker's resource usage and document hit rate, the algorithm is set up, in such a way that it uses up approximately the same network resources as *SemAnt* does when starting. The settings for the random-walk are 2 walkers ($k = 2$) with a lifetime of 25 clock ticks. Both the random-walk and *SemAnt* are run for 10.000 clock ticks. When searching for documents, only the pheromone trails which exactly match the search query are considered. The document distribution parameter P_{expert} was set to 60%, exactly as described in Section 5.2.3.
Charts depicting a resource usage and hit rate comparison of the two algorithms are shown in Figures 7.2 and 7.3.
The resource usage and hit rate of the random-walk algorithm remains constant at about 59.27 messages (including messages which return documents

to the source of the query) and a yield of 1.49 documents per query. *SemAnt* starts out with 60.44 messages (including forward and backward ants) and a hit rate of 1.82 documents per query. The optimization performed by *SemAnt* increases these values to an average of 55.13 messages and 3.8 documents after 1000 clockticks. After about 2000 clock ticks *SemAnt* reaches its peak performance, and returns an average of 3.95 documents while sending 53.02 messages.
A detailed description of the experiment can be found in [41].

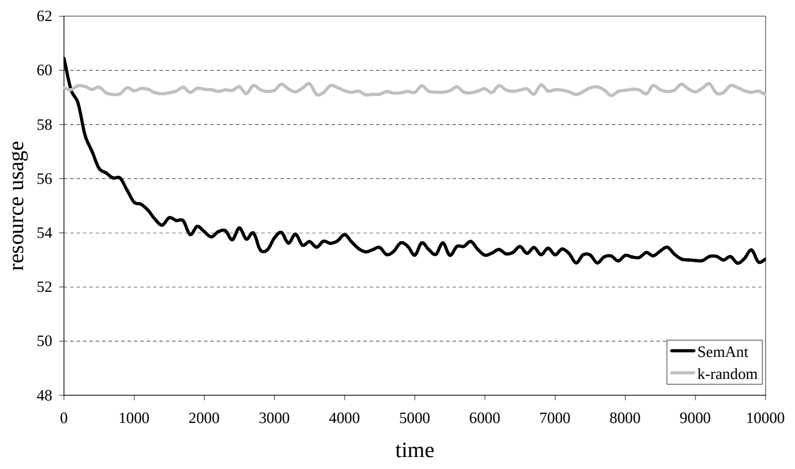


Figure 7.2: *SemAnt* vs. random-walk – Resource usage

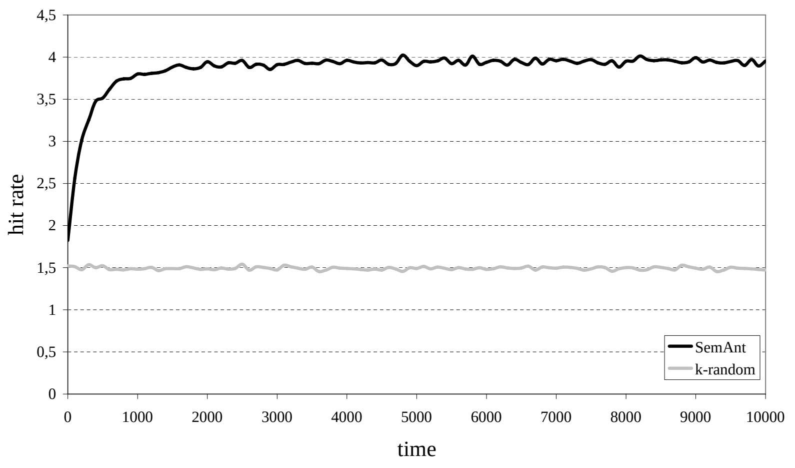


Figure 7.3: *SemAnt* vs. random-walk – Hit rate

7.1.3 Exploiting the hierarchic classification of documents

This experiment shows (1) the effect of including super-concepts in the search process, and (2) how the distribution of documents in the network affects *SemAnts'* performance. Six scenarios with different document distributions ($P_{expert} = [100\%, 80\%, 60\%, 40\%, 20\%, 0\%]$) were created. For each of six scenarios two simulation runs with a duration of 5.000 clock ticks were started. In the first simulation run, the documents were searched by following pheromone trails which exactly matched the queries' keywords. In the second run, the super-concepts of the keywords were integrated into the search process as described in Section 4.3. Figure 7.5 depicts the results of the first simulation run (not using the hierarchy of keywords). The *efficiency* is a measure of messages sent per found document. Lower values indicate a better performance of the algorithm. As expected, the algorithm's performance increases with the number of similar documents that are located at the same node. In Figure 7.4 the

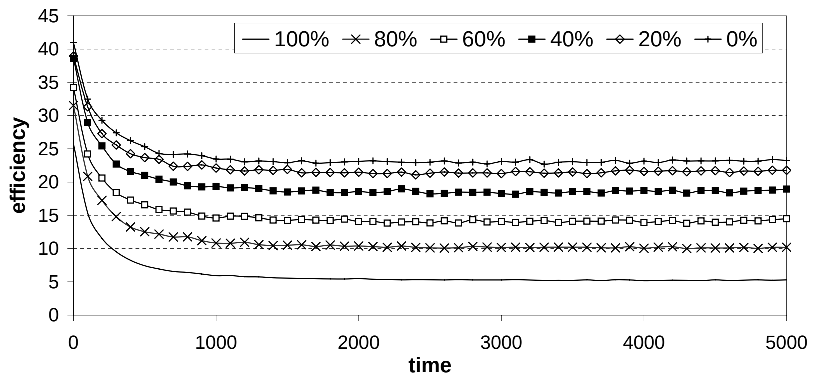


Figure 7.4: Efficiencies not using super-concepts

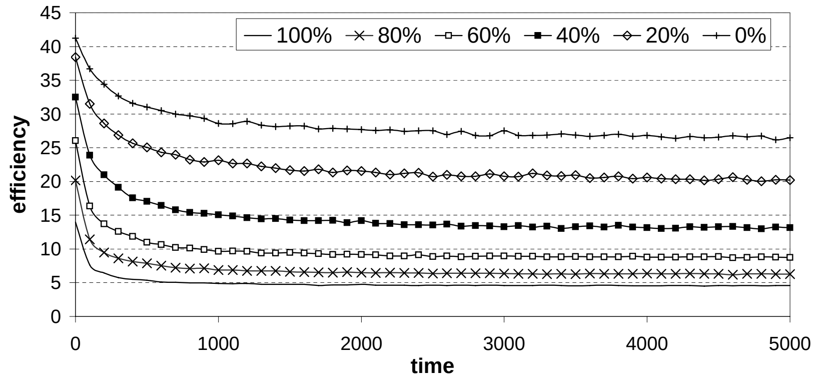


Figure 7.5: Efficiencies using super-concepts

results of the second simulation run are displayed. It can be seen that the algorithm's performance is worse if all documents are randomly distributed across the network and the super-concepts are considered. For situations where some patterns in the document distribution can be found, the algorithm's performance increases and the optimization is done faster than without using the trails of the super-concepts. The greatest increase of performance is found where P_{expert} is set to 60%. In this case the inclusion of super-concepts in the search process raises the efficiency of *SemAnt* by 39.5 percent.
A detailed description of this experiment can be found in [42].

7.2 Pheromone distribution

In this section the distribution of pheromones across the network is explained. Contrary to the previous experiments, only one of the network's nodes holds documents for a specific search key. All of the queries that are generated during the simulation are using this concept as search key, and are therefore looking for the one node owning the documents. The network has a diameter of 9 and ants have a lifetime of 10, thus each ant has a chance to find the documents before it expires. The simulation was run for 100 clock ticks.
Table 7.2 shows the average pheromone amounts of pheromones on all paths of the nodes, depending on the shortest path distance between each node and the node holding the documents. It can be seen that the closer an ant gets to the documents, the higher the pheromone concentration leading to the documents gets.

Distance	**Pheromone concentration**
9	0.0000349
8	0.6604124
7	2.7342447
6	7.0967345
5	18.155665
4	34.864378
3	76.079236
2	184.44743
1	530.63213
0	3205.7666

Table 7.2: Pheromone concentration

7.2.1 Circling the nodes

As shown in the previous section, the concentration of pheromones drastically increases with the proximity of documents matching the pheromone trail. When they have found the documents, forward ants spawn a backward ant, which returns the documents to the querying node. The forward ants then continue their path and try to find some more documents. Since the pheromone trails leading to the node they have found are very strong, the forward ants try to reach this node again. Ants are allowed to visit each node only once, therefore forward ants try to stay close to the node for the rest of their lifetime. It is unlikely that they will find another significant source of documents.
This behaviour probably depends on the clustering coefficient of the network. If the network is not clustered, ants may find other document sources, since there is a low probability that they find a circle back to the first node where they located a document source.

8 Conclusion and Future Work

The final chapter sums up the insights gained by the work on the *SimulAnt* system in Section 8.1. Some ideas how the *SemAnt* algorithm and the simulation environment may be improved are presented in Section 8.2.

8.1 Conclusion

Ant algorithms are a class of algorithms which are inspired by the food gathering behaviour of real ant colonies (Chapter 3). The trails that are built by ants are a result of an indirect communication system used by the ants. This indirect communication, called stigmergy, employs the dropping of volatile chemical substances (pheromones) to mark the environment. On their way to the food source and back to the nest the ants drop pheromones their path, which attract other ants and cause them to follow that path. Due to the differential path length, shorter paths get marked with greater amounts of pheromones and thus become more attractive to be followed. The stigmergic communication and the resulting trail laying and trail following behaviour of real ants can be adapted to solve optimization problems from the domain of computer science. Several optimization problems, including the travelling salesman problem, map coloring, and query routing have been tackled by ant algorithms with great success.
The *SemAnt* algorithm (Chapter 4) is an ant algorithm designed to efficiently route queries through peer-to-peer networks. Queries, in *SemAnts'* case, are used to search for documents which are located in the nodes of the network. The documents are annotated with keywords from hierarchical classification schemes, so that the semantic relations between the keywords may be exploited when routing the queries through the network.

As the practical part of this diploma thesis a simulation environment, *SimulAnt*, used to test the applicability of *SemAnt* to query routing in peer-to-peer networks has been designed and implemented (Chapters 5 and 6). *SimulAnt* can be controlled via a graphical user interface, and is capable of simulating ant-based algorithms in networks with several thousands of nodes and hundred-thousands of ants. The performance of the algorithm can be checked during and after the simulation run by plotting charts of metrics such as the average number of messages sent, or average number of documents found. Simulation data (Chapter 7) generated with the *SimulAnt* system show that *SemAnt* is well suited to solve the query routing problem at hand. According to the simulation results, the *SemAnt* algorithm is much more efficient than the *k-random-walk* algorithm, which was chosen as reference algorithm for performance comparisons.

8.2 Future Work

Several issues concerning the algorithm's performance and the usability of the simulation environment, which are up to now not resolved, have arisen. In the following, some propositions about future work, which may enhance the capabilities of the *SemAnt* algorithm or the *SimulAnt* platform, are given.

Optimization of network and algorithm parameters Currently the *SemAnt* algorithm has only been tested in networks with a constant path length, therefore neglecting parameters like the weighting of documents and path length when distributing pheromones. When performing tests with networks with varying path lengths, the impact of those parameters on the search behaviour can be determined. In addition, the effect of changing the pheromone evaporation rate has not yet been explored.

Upper bound for retrieved documents As stated in Section 7.2, ants which have found a node which has many documents matching its search query will, after sending a backward-ant, circle the node for the rest of their lifetime because of the high pheromone concentration on its neighbours. This behaviour leads to a great number of messages, which yield few to no documents. To prevent ants from circling nodes, an upper bound on the documents which will be fetched by ants could be introduced. If an ant hits a node which has enough documents to satisfy the upper bound, the ant would spawn a backward-ant, and then expire.

Decisions at paths with close to equal pheromone levels Related to the upper bound for fetched documents are the ants' decisions at paths where two trails are near to equal in their pheromone concentration. If two nodes holding large amounts of documents exist, and an ant comes close to them, it would, when using the exploitation strategy, find only one of those nodes. It is thinkable that between those two nodes there exists a node from which paths lead to both of those nodes, and that the

pheromone concentrations on these paths are nearly the same. In this case, another forward-ant could be spawned, and the two ants could exploit both paths. A similar behaviour could be achieved if ants which are confronted with such a situation would choose their exploration behaviour by default.
Whether this spawning of multiple ants actually entails a higher number of found documents, or just sends more ants to the same node on different paths, is probably depending on the structure of the network and has yet to be tested.

Adaptation to other algorithms *SimulAnt* has been designed to support the execution of the *SemAnt* algorithm. While *SemAnt* is a query routing algorithm, most other ant-based algorithms work on the same principles, which are based on the ACO-meta-heuristic (see Section 3.2). If the need arises to use *SimulAnt* to simulate algorithms similar to *SemAnt*, some refactoring work could be done. The `simulant.network.AntNode` class, and especially its `receiveAnt` method, are responsible for the ants' behaviour. By making the `receiveAnt` method abstract and implementing it in classes derived from `AntNode`, implementations of various ant-based algorithms would be possible. The different algorithms could then be chosen by loading the appropriate subclass of `AntNode`. In addition, by doing so the *k-random-walk* reference algorithm, which is currently implemented in the same method as the *SemAnt* behaviour, could be moved to its own class, allowing for a more modular and clear source code.

Bibliography

[1] K. Aberer and M. Hauswirth. An Overview on Peer-to-Peer Information Systems. *Proceedings of Workshop on Distributed Data and Structures (WDAS-2002)*, 2002.

[2] L. A. Adamic. Zipf, power-laws, and pareto-a ranking tutorial. *Xerox Palo Alto Research Center, Palo Alto, CA, http://ginger. hpl. hp. com/shl/papers/ranking/ranking. html*, 2002.

[3] Apache Software Foundation. Apache Ant. `http://ant.apache.org/`.

[4] AquaFold, Inc. Aqua Datastudio. `http://www.aquafold.com/`.

[5] Association for Computing Machinery. ACM Computing Classification System (ACM CCS). `http://www.acm.org/class/1998/ccs98-intro.html`, 1998.

[6] O. Babaoglu, H. Meling, and A. Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 02)*. IEEE, July 2002.

[7] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[8] A. Bauer, B. Bullnheimer, R. F. Hartl, and C. Strauss. An Ant Colony Optimization Approach for the Single Machine Total Tardiness Problem. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1445–1450, Mayflower Hotel, Washington D.C., USA, July 1999. IEEE Press.

[9] B. Bullnheimer, R. Hartl, and C. Strauss. Applying the Ant System to the Vehicle Routing Problem. In *Second Metaheuristic International Conference (MIC'97)*, Sophia-Antipolis, France, July 1997.

[10] G. D. Caro and M. Dorigo. AntNet: Distributed Stigmergy Control for Communications Networks. *Journal of Artificial Intelligence Research (JAIR)*, 9:317–365, July 1998.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Workshop on Design Issues in Anonymity and Unobservability*, 320, 2000.

[12] Cloud Garden. Jigloo. `http://www.cloudgarden.com/jigloo/`.

[13] CollabNet, Inc. Subversion. `http://subversion.tigris.org/`.

[14] F. Comellas and J. Ozon. An ant algorithm for the graph colouring problem. In *ANTS'98 – From Ant Colonies to Artificial Ants: First international workshop on ant colony optimization*, Brussels, Belgium, October 1998.

[15] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the Operations Research Society*, 48:295–305, 1997.

[16] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 02)*. IEEE, July 2002.

[17] M. Dorigo and G. Di Caro. The ant colony optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London, 1999.

[18] M. Dorigo and L. M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1:53–66, 1997.

[19] M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[20] L. Gambardella and M. Dorigo. HAS-SOP: Hybrid Ant System for the Sequential Ordering Problem. Technical report, Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale, 1997.

[21] L. Gambardella, E. Taillard, and G. Agazzi. *New Ideas in Optimization*, chapter MACS-VRPTW: A Multiple Ant Colony System for Vehicle Routing Problems with Time Windows, pages 63–76. McGraw-Hill, London, UK, 1999.

[22] L. Gambardella, E. Taillard, and M. Dorigo. Ant colonies for the QAP. *Journal of the Operational Research Society*, 50(2):167–176, 1999.

[23] D. Geary. Simply Singleton. `http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns-p3.html`, April 2003.

[24] W. Goh. Review: JFreeChart. `http://software.newsforge.com/software/06/01/04/1752256.shtml`, January 2006.

[25] S. Goss, S. Aron, J. Deneubourg, and J. Pasteels. Self-organized shortcuts in the Argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989.

[26] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM 2000*, pages 1371–1380, Tel Aviv, Israel, March 2000. IEEE.

[27] P. Grassé. La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6(1):41–80, 1959.

[28] K. Güclü. SWT Programming with Eclipse. `http://www.developer.com/java/other/article.php/10936_3330861_1`.

[29] M. Guntsch and M. Middendorf. Pheromone Modification Strategies for Ant Algorithms Applied to Dynamic TSP. In *Applications of Evolutionary Computing*, Lecture Notes in Computer Science, pages 213–222. Springer, 2001.

[30] M. Hauswirth and S. Dustdar. Peer-to-Peer: Grundlagen und Architektur. *Datenbank-Spektrum*, 13:5–13, 2005.

[31] F. Heylighen. Principia Cybernetica Web. `http://pespmc1.vub.ac.be/SELFORG.html`.

[32] A. Howe. Napster and Gnutella: a Comparison of two Popular Peer-to-Peer Protocols, 2000. Universidade de Victoria.

[33] Internation Business Machines Corp. JFace. `http://wiki.eclipse.org/index.php/JFace`.

[34] J. M. Kleinberg. Navigation in a small world. *Nature*, 406:845, August 2000.

[35] G. Leguizamon and Z. Michalewicz. A new version of ant system for subset problems. *Proceedings of the 1999 Congress on Evolutionary Computation*, 2, 1999.

[36] A. Löser, C. Tempich, B. Quilitz, W. Balke, S. Staab, and W. Nejdl. Searching Dynamic Communities with Personal Indexes. *Proceedings of the 4th International Semantic Web Conference*, 2005.

[37] Q. Lv, P. Cao, E. Cohen, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 16th ACM Conference on Supercomputing*, pages 84–95, June 2002.

[38] V. Maniezzo and A. Carbonaro. An ANTS heuristic for the frequency assignment problem. *Future Generation Computer Systems*, 16(8):927–935, 2000.

[39] V. Maniezzo and A. Colorni. The Ant System Applied to the Quadratic Assignment Problem. *Knowledge and Data Engineering*, 11(5):769–778, 1999.

[40] E. Michlmayr, S. Graf, W. Siberski, and W. Nejdl. Query Routing with Ants. In *Proceedings of the 1st Workshop on Ontologies in P2P Communities, 2nd European Semantic Web Conference 2005 (ESWC2005)*, May 2005.

[41] E. Michlmayr, A. Pany, and S. Graf. Applying Ant-based Multi-Agent Systems to Query Routing in Distributed Environments. In *3rd IEEE Conference On Intelligent Systems*, London, UK, September 2006. IEEE.

[42] E. Michlmayr, A. Pany, and G. Kappel. Using Taxonomies for Content-based Routing with Ants. In *Proceedings of the Workshop on Innovations in Web Infrastructure, 15th International World Wide Web Conference (WWW2006)*, Edinburgh, UK, May 2006.

[43] S. Milgram. The small world problem. *Psychology Today*, 2(1):60–67, 1967.

[44] R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[45] MySQL AB. MySQL open source database. `http://www.mysql.com`.

[46] Netscape Communication Corp. dmoz Open Directory Project. `http://dmoz.org/`.

[47] M. Newman, C. Moore, and D. Watts. Mean-Field Solution of the Small-World Network Model. *Phys. Rev. Lett.*, 84(14):3201–3204, April 2000.

[48] Object Refinery Ltd. JFreeChart. `http://jung.sourceforge.net/`.

[49] J. O'Madadhain, D. Fisher, and T. Nelson. Java Universal Network/-Graph Framework. `http://jung.sourceforge.net/`.

[50] OSGi Alliance. OSGi. `http://www.osgi.org`, January 2000.

[51] D. R. Price, Ximbiot, and Free Software Foundation, Inc. CVS – Concurrent Versions System. `http://www.nongnu.org/cvs/`.

[52] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[53] M. Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. *p2p*, 00:0099, 2001.

[54] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[55] R. Schoonderwoerd, O. E. Holland, J. L. Bruten, and L. J. M. Rothkrantz. Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, 2:169–207, 1996.

[56] C. Shirky. What is p2p... and what isn't? `http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html`, November 2000.

[57] K. M. Sim and W. H. Sun. Ant Colony Optimization for Routing and Load-Balancing: Survey and New Directions. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 33(5):560–572, 2003.

[58] SoberIT Lab, Technical University Helsinki. TexLipse. `http://texlipse.sourceforge.net`.

[59] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. In *Proceedings of IEEE INFOCOM 2003*, April 2003.

[60] T. Stützle and H. Hoos. MAX–MIN Ant System and local search for the traveling salesmanproblem. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation*, pages 309–314, Indianapolis, IN, USA, April 1997.

[61] T. Stützle and H. Hoos. *Advances and Trends in Local Search Paradigms for Optimization*, chapter MAX–MIN Ant system and local search for combinatorial optimization problems. Springer, 1998.

[62] Sun Microsystems, Inc. Abstract Window Toolkit (AWT). `http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html`.

[63] Sun Microsystems, Inc. Java Foundation Classes (JFC/Swing). `http://java.sun.com/products/jfc/index.jsp`.

[64] The Eclipse Foundation. Eclipse. `http://www.eclipse.org`.

[65] The Eclipse Foundation. SWT: The Standard Widget Toolkit. `http://www.eclipse.org/swt/`.

[66] W3C. RDF Vocabulary Description Language. `http://www.w3.org/TR/rdf-schema/`.

[67] D. J. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

[68] B. Yang and H. Garcia-Molina. Designing a Super-Peer Network. In *ICDE*, pages 49–, 2003.

[69] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

Disclaimer

No animals, especially ants (real or simulated), were in any way mistreated or harmed in the production of this document.