Ant Routing System

–

a routing algorithm based on ant algorithms applied to a simulated network

Written by: Mikkel Bundgaard, Troels C. Damgaard,
Federico Decara og Jacob W. Winther.

Supervisor: Kåre Jelling Christoffersen

Spring 2002 IT University of Copenhagen – Internet technology

Abstract

In this report a routing algorithm Ant Routing System (ARS) is presented. ARS is based on a general-purpose metaheuristic named *Ant Colony Optimization*, which is a framework for building ant-inspired algorithms. ARS is applied as the routing algorithm in a simulated package-switched point-to-point network.
It is investigated whether ARS is able to obtain an increase in throughput while avoiding package loss in a heavily loaded network, when packages are sent between two distinct routers. To this end, it is investigated how prioritizing different heuristics effect the quality of the routing performed.
It is concluded that ARS behaves differently depending on the relative priority of positive feedback, negative feedback and local heuristics, and that it is possible to adjust the parameters to achieve distribution of traffic over several paths when the network is heavily loaded, resulting in a higher throughput and lower package loss.

Keywords: Ant algorithms, Ant Colony Optimization, ACO, Ant System, routing, metaheuristics, network model, swarm intelligence.

Index

INTRODUCTION **1**
MOTIVATION 1
PROBLEM DOMAIN 1
PROBLEM DEFINITION 2
PROCESS 2
TARGET READERSHIP 3
STRUCTURE OF THE REPORT 3
WHO WE ARE 4
ACKNOWLEDGEMENTS 4
THEORY **5**
GRAPH THEORY 5
ANT ALGORITHMS 6
COLLECTIVE BEHAVIOR IN SOCIAL INSECTS 7
ARTIFICIAL ANTS 9
DOUBLE BRIDGE EXPERIMENT 11
THE ANT COLONY OPTIMIZATION METAHEURISTIC 14
METAHEURISTICS 19
ANT SYSTEM 20
NETWORK MODEL **25**
FORMAL DEFINITION 25
ROUTING 25
FORMAL DEFINITION CONTINUED 27
ROUTING QUALITY IN THE MODEL 29
NETWORKS AND DYNAMIC GRAPHS 30
SUMMARY 31
EVENT DRIVEN NETWORK SIMULATION **32**
THE CONCEPT OF EVENT LOCATION 32
SIMULATION EVENTS 32
ANT PACKAGES IN THE SIMULATED NETWORK 33
SUMMARY 34
ANT ROUTING SYSTEM - AN ACO NETWORK ROUTING ALGORITHM **35**
MOTIVATION 35
AN OVERVIEW OF ANT ROUTING SYSTEM 35
REQUIREMENTS OF THE NETWORK 36
MECHANISMS IN THE ALGORITHM 37
SUMMARY 41
DESCRIPTION OF SELECTED EMPIRICAL OBSERVATIONS **42**
COMPARISON WITH A “BENCHMARK” ALGORITHM 42
NETWORK TOPOLOGIES 43
DEFINITIONS 45
THESES 45
RESULTS OF EXPERIMENT 1 – TEST OF BASIC CAPABILITIES IN A 10X10 GRID NETWORK 47
EXPERIMENT 2 – DOUBLE BRIDGE NETWORK WITH HEAVY NETWORK LOAD 51
EXPERIMENT 3 –10X10 GRID NETWORK WITH HEAVY NETWORK LOAD 54
INITIAL CONVERGENCE 57
SUMMARY 59
CONCLUSION **60**
IDEAS FOR FURTHER STUDIES AND IMPLEMENTATIONS **61**
ANT ALGORITHMS AND INDUCTIVE MACHINE LEARNING ALGORITHMS **62**
INDUCTIVE MACHINE LEARNING ALGORITHM THEORY APPLIED ON ANT ALGORITHMS 62
INDUCTIVE MACHINE LEARNING ALGORITHMS COMBINED WITH ANT ALGORITHMS 64
BIBLIOGRAPHY **67**

Introduction

The goal of this project is to investigate properties of a certain type of swarm intelligence commonly referred to as *ant algorithms*. We will study the use of an ant algorithm operating upon a dynamic problem domain, that is, a domain that changes as a function over time.

Specifically, we will discuss and use an *ant-inspired graph-based general-purpose algorithm metaheuristic* named *Ant Colony Optimization* as the basis of an implementation of a routing algorithm in a package-switched point-to-point network (such as the Internet). Ant-inspired algorithms have the capability of finding short paths in graphs, and show an inherent adaptability that could be utilized to solve dynamic problems such as routing in a network.

Motivation

Swarm intelligence is an area of research that over the last decade has experienced a boom in interest. Inspired by the seemingly intelligent behavior of swarms of primitive animals, swarm intelligence has proven to be a promising field of research in many different areas.

A swarm of ants in the search for food shows the remarkable capability of finding shortest paths between a found food source and the anthill. Even though any single ant could be said to possess the capability of finding a short path from the anthill to a nearby food source, the probability of this occurring is very small, since an ant is not a very smart animal. The amazing thing is that when many ants cooperate on finding food, using pheromone trails as a simple indirect form of communication, the swarm of ants seems to be able to find a shortest path effectively. Another feature is their ability to adapt to a changing environment. If an obstacle is placed on the path from the food source back to the anthill, ants are capable of finding the shortest path around the obstacle – and possibly find food sources closer to the anthill.

The *emergent* intelligence that a swarm of ants seems to possess is enticing from the perspective of computer science, because of the possibility to simulate and potentially exploit this behavior to solve algorithmic problems.

By observing and modeling the processes that occur in natural ants when working in their natural domain, it is possible to use this as inspiration to create a ‘colony’ of artificial ants, working in an environment represented by a graph and potentially experience a similar emergent intelligence. In fact, researchers in the field of biology have developed good theories on how ants communicate as a group and attempts to adapt these theories to a simulated domain in a computer have verified that it is possible to obtain emergent intelligent behavior from colonies of artificial ants [Bonabeau et al., 1999].

Problem domain

Network routing is one of the possible uses for the abilities of artificial ant colonies, which potentially could utilize their adaptive behavior.

The quality of a routing algorithm could be measured by how fast the algorithm is able to transfer packages through the network while minimizing package loss. At an abstract level, a network can be viewed as a graph, with routers represented by vertices and wires represented by edges. If an ant-inspired algorithm is capable of finding a short path in a graph, can it also be that their adaptable nature allows them to find good solutions in a graph when costs on edges are time-varying stochastic variables[1]? This is the question that we will try to answer in this report.

We propose that one of the problems with existing routing algorithms in use is their deterministic and static nature, given the highly dynamic nature of a network. In principle, a router calculates a single shortest path to other routers in the network on the basis of latency values that are updated with intervals. All packages are

[1] It is important to note that edge cost varies *determinedly* as a function of package load and a number of predetermined properties (queue length etc. etc.) in the system. The stochastic factor is introduced, as it is not presumed that sending -frequency and -sources is known a priori.

sent deterministically over the shortest paths to their destination. In the intervals between updates, the routing algorithms thus route packages based on static information.

This approach present the problem that a network topology might have two distinct paths with almost equal length between a pair of fast routers, with a number of slower routers on the intermediate paths. Distribution over both paths could ensure the maximum throughput with a minimum of increased latency.

Another problem is that routing a package has a feedback mechanism: Routing a package to a router increments the latency to the receiver because the receiver must process the sent package, thereby making the latency values on which the shortest path was calculated invalid. In the case where two paths exist with almost equal length the latency increase from using the lowest latency path would eventually make the other path the fastest.

One could argue that the information used by existing routing algorithms is recalculated at regular intervals, ensuring a relatively updated shortest path, but this could be viewed as fixing a problem when it arises instead of trying to avoid it.

We propose[2] that an algorithm that continuously updates the information used for routing would constitute a solution better suited to this type of problem. The information would be based on the gathered experience with latency variations experienced by packages routed through the network.

The optimal solution would be a routing algorithm that distributes packages over the necessary number of different paths with minimal length seen over a period of time. We speculate that this is exactly what ant-inspired algorithms can do. By continuously searching the domain in which they work, they show the ability of converging towards ‘good’ shortest paths, without losing the ability to adapt to changes in the domain.

We will show that a successfully designed ant-algorithm will be able to mainly use the shortest path when the network is capable of handling the packages, but utilize distribution over several short routes when the network is congested.

Problem definition

Let us continue by specifying the problem we wish to examine in this project.

Can an ant-inspired routing algorithm be used to obtain a distribution of package transfers, which result in improved throughput and reduced package loss in a simulated package-switched point-to-point network?

To be able to answer this question, we have implemented an ant-inspired routing algorithm capable of routing packages in a simulation of a package-switched point-to-point network. We will base the answer upon the capabilities of the ant-inspired routing algorithm in this domain.

The behavior of ant-inspired algorithms can be controlled by prioritizing different heuristics used to construct solutions. To investigate their effects in the context of routing we will answer the following question.

How does an ant-inspired routing algorithm react to *the prioritizing of* different heuristics in the context of routing in a simulated package-switched point-to-point network?

To answer these questions we will run simulations examining the different effects.

Process

The purpose of this report is for us to implement, optimize and analyze ant algorithms. Our study-related goal is also to demonstrate understanding of the ACO metaheuristic, a design template for ant-inspired algorithms, and to make an investigation of the properties of our ant algorithms, both from an analytical and from an empirical point of view.

---

[2] As did Di Caro and Dorigo in [Di Caro & Dorigo, 1998] in 1998.

Our work with this project has mainly been conducted as a series of interdependent and related sub-projects. We find that the learning process of this project have been positively influenced by this method of work.

We have chosen to include only some of the observations and conclusions, which we have made through the progress of this project – as we have tried to make this report coherent and mainly focused on the application of ant algorithms to a specific problem domain.

Target readership

In this report, we will give an introduction to the field of ant algorithms, and we will explain the biologically inspired terminology.

This report is mainly written for readers having knowledge corresponding to students of computer science or above. The reader is expected to have a basic understanding of data structures and algorithmic theory and a reasonable understanding of networks and the problems regarding routing. The description of how the network model is implemented by an event-driven simulation presupposes some prior experience with event-driven simulations, but that chapter is not essential for the central themes in this report.

Furthermore, as the implementations of this project are made in Java, experience with this language would probably ease the reading of the code. However, as we have thoroughly organized and commented the code (quite extensively) according to the code conventions issued by Sun, it is our belief that a reader with an understanding of any high-level imperative language would to be able to understand the implementations.

Structure of the report

The first chapter of this report presents a formal definition of graphs and the theory of ant algorithms, self-organization and the Ant Colony Optimization metaheuristic. The chapter ends with a description of an archetypical ant algorithm, Ant System, which was developed by Dorigo et al. to solve the Traveling Salesman Problem. The keen reader will find a fully documented implementation of the Ant System algorithm in the appendix B [*Class AntSystem*] based on a generalized implementation of the Ant Colony Optimization metaheuristic.

Following the theory chapter, we present a description of our network model. In particular, we define how to calculate the routing time between two routers in this network. This chapter also contains an explanation of the difference between a dynamic graph with edge weights being only stochastic functions of time and our network model. In addition, a description of how to measure the quality of routing in the model by use of package loss and throughput is presented.

The chapter [*Event driven network simulation*, p. 32] contains an explanation of our event driven simulation of a network, which is based on the network model mentioned above. The explanation discusses the basic elements in the simulation, how events and time are handled in the system, but does not go into detail about the actual implementation. Again, we refer the eager reader to the appendix B [*Code*].

Our ant-inspired routing algorithm Ant Routing System (ARS) builds on the theory presented in the earlier chapters. We present the routing algorithm in the chapter [*Ant Routing System - an ACO network routing algorithm*, p. 35]. We discuss the background for implementing an ant-inspired routing algorithm, and describe the properties and mechanisms of ARS in overview and detail. Finally, we discuss the purpose and the motive behind each of the heuristics implemented in ARS.

In the chapter [*Description of selected empirical observations*, p. 42] we present the results obtained by using ARS as the routing algorithm in some simulated networks. These tests are mainly designed to illustrate the properties of ARS, and only secondarily to give hints about the performance of ARS.

The report ends with a conclusion and pointers to further work based on this report, and a short discussion of some interesting observations regarding similarities between ant algorithms and inductive machine learning algorithms.

Who we are

We are four students, all originally from Roskilde Universitetscenter (RUC), and we have around 3 to 4 semesters of computer science behind us (this semester included). The report is written on the second semester level on IT-University of Copenhagen (ITC), where two of the group members are now studying.

We have worked together before a number of times, and have a somewhat 'synergic' work form, meaning that, even though we have our specialties and main areas, we all part take in each others work.

None of us are native English speakers (which should be fairly obvious to the reader) but we have nevertheless chosen to write the report in English. The purpose has been to practice our skills in technical English. We apologize for any inconvenience or suffering this may have caused the reader.

Acknowledgements

We would like to thank our supervisor Kåre Jelling Christoffersen for his supervision and support through the semester. Our thanks also go to our former supervisor Keld Helsgaun. He proposed to us the idea of working with ant algorithms. Lastly we would also like to thank Thomas Hildebrandt for making it possible for us to cooperate across the borders between universities. We believe that free propagation of student activities benefit students and universities alike, and therefore in the end also the society.

Theory

In this chapter we will present theory used in this report. Since the problems that are discussed in this project are graph problems we will initially try to establish some nomenclature to facilitate the further discussions. We then continue to discuss ant algorithms and the ACO metaheuristic that is a template for building ant-inspired algorithms. We then continue to show an exemplary application of the ACO metaheuristic to solve the traveling salesman problem.

Graph theory

Since many of the following discussions assume a basic knowledge of graph theory, we will start by giving a formal definition of a graph and often used terms to ensure consensus on the nomenclature of this project in the context of graph theory.

Definition of a graph

Let (V,E) denote a given graph G, where $V=\{v_0,\ldots v_n\}$ is a finite set whose elements are called vertices and $E=\{e_0,\ldots,e_m\}$ is a proper subset of $V \times V$ whose elements are called edges. Each edge $e \in E$ is a pair (v_i,v_j) where $v_i \neq v_j$ (there is no self-loops in our graphs). If (v_i,v_j) is an ordered pair for any $(v_i,v_j) \in E$, then $G=(V,E)$ is said to be a directed graph. Otherwise it is said to be an undirected graph. Each edge in the graph also have an associated weight given by a weight function $w : E \rightarrow \mathbf{R}$. E.g. $w(v_i,v_j)$ is the weight of the edge (v_i,v_j). A *complete graph* is a graph such that every vertex has an edge to all the other vertices in the graph.

We define the *neighborhood* N_{v_i} of a vertex v_i to be the set of vertices that vertex v_i is connected to by an edge thus the neighborhood N_{v_i} is the set $\{v_j \mid (v_i, v_j) \in E\}$.

Let s and t be two given vertices of G. A path p of length k from s to t in G is a sequence of vertices of the form $p = \langle v'_0, v'_1, v'_2,\ldots, v'_k \rangle$ such that $v'_0 = s$, $v'_k = t$ and $(v_{i-1}, v_i) \in E$ for $i = 1,2,\ldots,k$. The weight of the path $p = \langle v'_0, v'_1, v'_2,\ldots, v'_k \rangle$ is the sum of its constituting edges:

(1) $$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Hamiltonian cycle

The number of distinct Hamiltonian cycles (meaning a cycle visiting every vertex once and returning to the start vertex) in a complete graph is $(|V|-1)!$. By distinct we mean that the cycle $c = \langle v'_0, v'_1, v'_2 \rangle$ in a complete graph containing the vertices (v'_0, v'_1, v'_2), which ends in v'_0 , is the same as the cycle $d = \langle v'_1, v'_2, v'_0 \rangle$, which ends in v'_1, just with another starting vertex along the same cycle. Because there are $|V|!$ possible permutations of the vertices of a graph, there are $(|V|-1)!$ Distinct Hamiltonian cycles in a complete graph (to see this you could define all the cycles from a given start vertex).

Random walk

A random walk on a graph refers to a random path generation process starting in a vertex v_i. The next vertex to add to the path is chosen uniformly at random in the neighborhood of vertex v_i.

Thus the probability of adding vertex v_j to the path is the inverse to $|N_{v_i}|$ if there is an edge between vertex v_i and vertex v_j, and 0 if there does not exists an edge between vertex v_i and vertex v_j. Each of these steps is defined as a *random step*. Formally stated this is:

(2)
$$p(v_i, v_j) = \begin{cases} 1/|N_{v_i}|, & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Restricted random walk

Further, we define the term: '*A restricted random walk*' to be a random walk in which a vertex may only occur once in the created path, thus setting an upper bound on the length of the path to $|V|$.

After a vertex has been added to the path, the probability of adding the vertex again is set to 0.

Thus the probability of adding vertex v_j – in the neighborhood of a vertex v_i – to the path is the inverse to the number of outgoing edges from v_i, which have not already been added to the path.

We name the set of vertices already added to the path the taboo-list, denoted T.

Analogically to the neighborhood definition above, we define the *restricted neighborhood* $J_{v_i}^T = N_{v_i} / T$, which is the neighborhood N_{v_i} restricted by the vertices in the taboo list T.

The probability of adding vertex v_j is 0 if there does not exists an edge between vertex v_i and vertex v_j or if v_j is in the taboo-list T.

A step in which a vertex is added in a restricted random walk is denoted a *restricted random step*. Formally stated this is:

(3)
$$p(v_i, v_j) = \begin{cases} 1/|J_{v_i}^T|, & \text{if } (v_i, v_j) \in E \text{ and } v_j \notin T \\ 0 & \text{otherwise} \end{cases}$$

Ant algorithms

In 1998, the authors of [Di Caro et al., 1999] proposed a metaheuristic called ACO in a technical report[3] with the goal of characterizing the common properties of a number of algorithms for discrete optimization (and approximation to NP-problems) developed through the 90's that were inspired by the foraging behavior of ants. These algorithms are typically called ACO-algorithms or ant algorithms.

We would like to take a brief historical detour to discuss the main work that motivated the field of ant algorithms. The initial work on ACO algorithms was first presented in a technical report [Colorni et al., 1991], and was made on the Traveling Salesman Problem (TSP). In this report three related algorithms (nicknamed *ant-density*, *ant-quantity* and *ant-cycle*) were presented, analyzed and tested on a few classic TSP-graphs. These algorithms were published in Dorigo's doctoral dissertation[4] and later in [Colorni et al., 1996]. The best algorithm (*ant-cycle*) was later simply referred to as the Ant System (AS) algorithm, and work upon the inferior algorithms were abandoned.

AS initially showed promising results, but did not scale well enough to be applicable on large graphs and was therefore not competitive with state-of-the-art algorithms for TSP. In [Dorigo & Stuetzle 2000] the authors concludes that the merit of AS was due to the fact that during the 90's a number of researchers - mainly based in Europe - were inspired by this approach to extend and improve AS; and more interestingly, also to develop algorithms similar in nature to solve a number of problems other than the TSP.

Some of these other application-domains are:

---

[3] And later published in : 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 11-23. Mcgraw Hill, London, UK, 1999.

[4] *"Optimization, Learning and Natural Algorithms"* (in Italian) PhD thesis from Dipartemento di Electronica, Politecnico di Milano, Italy (1992).

Quadratic assignment problem, Scheduling problems, Vehicle routing, Connection-oriented and Connection-less routing, Sequential ordering, Graph coloring, Shortest common super sequence, Generalized assignment and Constraint satisfaction. (For a full list of applications, algorithm names, and authors etc. – see [Dorigo & Stuetzle 2000, p. 24]).

Swarm Intelligence and Ant algorithms

It is worth mentioning that ant-algorithms belong to a larger field of algorithms called Swarm Intelligence algorithms. A book was published in 1999 – '*Swarm Intelligence: From Natural to Artificial Systems'* [Bonabeau et al., 1999] – where swarm intelligence is presented and defined as '*the emergent collective intelligence of groups of simple agents'*. This field is inspired by the sometimes surprising capabilities of collectives of social insects.

In the book, particular 'intelligent' swarm behaviors in different types of insects are presented, algorithms are modeled on this behavior, and examples of applications are presented. The behaviors range from cemetery organization principles studied in a number of ant-species applied to graph partitioning, nest building and self-assembling among wasps utilized for self-assembling in robotics and cooperative transport (again ants) applied to robotics. The largest chapter though, contains an overview of the aforementioned algorithms based on the foraging behavior of ants.

As this report focuses on ant algorithms, we have chosen to give a short presentation of the properties we have found central in the context of ant algorithms and in their inspiration, natural ants.

Collective behavior in social insects

In order to better understand the mechanisms in artificial social insects one must understand the collective behavior in real social insects. In the following, we will give a short description of the primary mechanisms that are determining for the collective behavior of social insects during food foraging, since this is the primary inspiration for the ACO metaheuristic. This will facilitate a later discussion regarding the creation of artificial insects, and also regarding algorithmic problems in a virtual domain.

Swarm algorithms and ant algorithms rely mainly on two concepts adopted from biology: *self-organization* and *positive feedback*.

Self-Organization

The term self-organization is used to describe macroscopic patterns, which originate from interactions and processes at the microscopic level [Bonabeau et al., 1999]. The term can be used to describe a social colony of insects where every single insect functions autonomously. There is no "supervisor" insect, which coordinates the labor of the insects, but still the collective organization of the colony is well-ordered. The conjecture is that complex collective behavior can emerge from the interaction of simple entities.

In [Bonabeau et al., 1999, p. 6] a social insect colony is described as a *"...decentralized problem-solving system, comprised of many relatively simple interacting entities.*" The way the system is made up by many small and simple entities enables it to both adapt to changes in the environment and to keep on functioning even though one (or more) of the entities stops functioning (dies). Self-organization in the context of social insects depends on the following mechanisms:

Positive feedback

Positive feedback refers to a mechanism that allows insects to converge towards good solutions to a problem. In the context of foraging, the problem is finding the shortest path from the anthill to the food source. In certain species, ants deposit pheromone trails while moving between a discovered food source and the anthill. Positive feedback is achieved because ants can smell deposited pheromone and have a natural tendency to follow the trail laid out by other ants. These ants again deposit pheromone on the same path, reinforcing the already existing pheromone trail, which again enhances the probability of other ants discovering the trail, thus yielding an ever-higher positive feedback.

The subtleties of this process will be explained by example in a later section [*Double bridge experiment*, p. 11] but will be set in the context of artificial ants.

Negative feedback

Negative feedback is the mechanism that counteracts the effect of positive feedback. An often-used example of negative feedback is pheromone evaporation. Ants must maintain a trail of pheromone continuously depositing more pheromone, otherwise pheromone will evaporate and consequently the created trail will be 'forgotten'.

This is a good property, since it allows the ants to leave bad solutions. If a path leads to a food source that is exhausted, the ants will not return to the anthill. Instead they continue their exploration, thus leaving the created trail to evaporate in search of other food sources.

Randomness in social insects

Self-organization strongly depends on randomness (random walks and errors in trail-following etc.). In [Bonabeau et al., 1999, p. 10] randomness is referred to as a crucial factor for the discovery of new solutions *"Not only do structures emerge despite randomness, but randomness is often crucial, since it enables the discovery of new solutions"*. An ant gone astray might stumble upon a new large food source closer to the anthill than any other food source previously found.

Properties of self-organization

A system can often be identified by several properties, if it is self-organized [Bonabeau et al., 1999, p. 12]:

- Creation of spatiotemporal structures in an initially homogeneous medium. The area around an anthill can be seen as an example of this. Assume that initially the area is unaffected by the anthill and the environment (medium) is homogeneous. When the ants begin to search for food, they create and maintain pheromone trail (structures) at different places in the area at different times.
- The possible coexistence of several stable states (multistability). This follows from the point above, because the structures are created using random deviations. This means that given an initial setup with equal probabilities for several solutions, any of these solutions can be chosen, because of random events in the beginning of the system.
- The presence of bifurcations (forkings) marks places where the behavior of the self-organized system changes noticeably when some parameters are changed.

Stigmergy

Self-organization primarily describes the macroscopic patterns that can be observed in an ant colony. The concept of stigmergic interactions focuses on the microscopic interactions between the ants of the system. In their book [Bonabeau et al., 1999] the authors uses the term stigmergy to describe how insects interact. Stigmergy is an indirect and asynchronous form of communication in which the insects manipulate the environment to communicate information to the other insects, which then responds to the change. The insects therefore do not have to be at the same place at the same time as the others to communicate with them.

In many ant species colonies, stigmergy refers to the deposition of pheromone by ants while they are moving. Other ants can then smell the deposited pheromone and have a natural tendency to follow the laid trail. This constitutes an asynchronous and indirect communication scheme, where one ant communicates to other ants where it has been, and it is how positive feedback is created. A little pheromone on a path might lead other ants to follow that same trail, depositing even more pheromone, which can lead to a positive feedback effect, if the selected path is good (leading to food) thus recruiting even more ants to follow the path.

How it works

In the above we have described the mechanism that creates self-organization, but the central question is how they participate in ensuring that ants find a path from the anthill to the food source. We will not go into detail about this process here, but we will try to explain the basic idea.

Positive feedback is the primary mechanism that allows colonies of ants to find short paths from the anthill to a food source. Positive feedback is created by stigmergic communication between the ants. Ants start out moving about at random, at some point an ant could find a food source and start to move back to the anthill (some ant species seems to always know the path back to the anthill). Other ants – still moving around at random – smells the trail and find the food source and also return to the anthill, following almost the same path. This increase in pheromone value recruits even more ants that start to follow the path, thus the positive feedback mechanism ensures that almost all ants follow the path, and thus the ants have found a path to the anthill.

To discuss how natural ants find the shortest path if more than one path to the food source exists is beyond the scope of this report, but we will give a simple example that will illustrate this ability in context of artificial ants see the section [*Double bridge experiment*, 11].

Artificial ants

Artificial ants are not intended to model real ants. They have abilities that cannot be attributed to real ants. The intention in ant algorithms is to keep the ants simple but this is not a requirement since they can be assigned any needed level of complexity. For instance, artificial ants have a memory about where they have been.

The mechanisms in biological ants described earlier have artificial counterparts. In the following we will define the presented terms in the context of ant algorithms. Also some additional modeling is required since we need to express exactly how artificial ants react to their environment. In biological ants we can observe some randomness in the behavior of the ants, but in order to model this behavior we need to express the randomness of artificial ants. This is a task of the *stochastic state transition* rule that we will describe below. It states how ants react to pheromone and to other stimuli in their environment. In the following the term ant will refer to artificial ants.

Self-Organization

In the sections below we describe how the different part of self-organization in colonies of social insects can be modeled by artificial ants in order to achieve the same useful properties of self-organization that biological systems has.

Stochastic state transitions

As described in the introduction, we can observe randomness in biological ants, but we need to model this randomness in artificial ants. This is the task of the *stochastic state transition* rule. As the name suggests the stochastic state transition is responsible for the state transition that ants do. We will give a more formal presentation of this process in a later section, but for now we will give an informal description of this process.

Ants construct solutions by moving on a graph. They perform stochastic walks in the graph, consisting of a series of stochastic steps. This stands in contrast to the random walks defined earlier in this report, since the decision of which vertex to add to the path is not uniformly random but subjected to a probability distribution.

If we denote a path as a state, meaning that both a single vertex is a state and a path is a state, the name stochastic state transition is meaningful. It refers to the decision process an ant performs when being in one state and move to the next (adding a vertex to the path).

What has not been pointed out explicitly earlier is that ants must have a termination criterion, which defines when a solution is reached. Stochastic walks in the graph therefore continue until the termination criterion is

reached. In the context of the routing problem this would be to find the destination where the package is to be routed.

Before we continue this discussion we need to introduce the mechanisms analogous for artificial ants as we did for biological ants.

Stigmergy

In the domain of ant algorithms stigmergy refers to a similar communication scheme to what natural ants use. They deposit virtual pheromone on the path they have created as a solution. This can either be on the vertices, on the edges, or on both.

Two types of pheromone depositing exist. Either the ants can construct a whole solution and then backtrack over the constructed solution depositing pheromone. This is called *delayed trail update* and thus happens when the ant has achieved its termination criterion.

Another option for the ants is to deposit pheromone every time they change state, which is referred to as *step by step trail update*. Thus ants deposit trail on the graph every time they add a vertex to the path. The possibility of combining the two methods also exists.

These trails of pheromone are shared among all ants, so that when an ant deposits pheromone at a location, all others can perceive some effects of this change when they move to the same location. When an ant deposits pheromone, it is added to the existing pheromone that exists on the location, thus reinforcing the trail value.

Positive feedback

When good solutions to problems are reinforced more (more pheromone is deposited on the path, that constitutes the solution) than solutions of lesser quality, this is referred to as *positive feedback* in the context of ant algorithms.

In ant algorithms, artificial pheromone trails are deposited by ants in amounts proportionate to the quality of the solution. The evaluation of a solution is performed by a quality function *f*, but this will be explained in greater detail in the section [*Problem representation*, p. 15].

If the solution found is a particular good one positive feedback sometimes results in an autocatalytic effect (a “snow ball” effect), by which more and more ants are recruited to follow the path that constitutes that solution. This can lead to the system converging to a single solution.

The autocatalytic effect is not always desirable, since it may happen when the ants have found a locally optimal solution. The mass recruitment that occurs can leave the ants unable to discover new solutions. This situation is described as a stagnation of the system. It is the job of other mechanisms to ensure that the ants keep looking for better solutions.

Negative feedback

Negative feedback is often used to avoid premature convergence on a suboptimal solution (stagnation). If premature convergence happens, the ants are “satisfied” with the result found so far and have stopped looking for new – and maybe better – solutions.

An example of negative feedback in ant algorithms is artificial pheromone evaporation. Artificial pheromone evaporation is usually implemented as a reduction in pheromone, which reduces the amount of pheromone to a percentage of the original amount. This implies that the amount of pheromone removed is proportionate to the (original) amount of pheromone. This means that good paths need to be continually reinforced by ants in order to ensure that they retain their pheromone value since they evaporate faster.

Stochastic state transitions continued

Let us continue the discussion of the stochastic state transitions. In the above a few relevant points is left unmentioned. Positive feedback implicitly assumes that the stochastic state transition takes pheromone values into account, and uses the relative amount of pheromone on the several possible new states to choose

the next state. The general rule should then be that the state with the most pheromone should be chosen with the highest probability.

Every ant algorithm has a stochastic state transition rule. This rule is used to calculate the stochastic distribution that is used to select the next state. This rule uses local variables at a location such as the pheromone and possibly local heuristic values to calculate the probability distribution that determines what state the ant will move to.

What is relevant to realize is that a stochastic choice is made based on the probability distribution. This means that even though a state has very little probability of being chosen, the possibility still exists.

This possibility of an ant making a ‘wrong’ turn (meaning: choosing a state with low probability) is often decisive because it enables the discovery of new solutions. It is up to the stochastic state transition rule to define the relevance of different local variables meaning how much emphasis should be put on pheromone values or on other local heuristics.

Other methods of forcing randomness upon the algorithm can be to normalize the probabilities for a given choice, so that a lower bound on state transition probabilities is set. An example could be to implement a bound so probabilities are never decreased below a given threshold, say for example 2%.

Summary

In the above we have presented the mechanisms that facilitate the self organizing properties of ant inspired algorithms. Ants use a positive feedback mechanism to converge towards good solutions by depositing pheromone on solutions they have created. This is facilitated by the of the stochastic state transition rule to create a probability distribution that ensures that states with a high pheromone value is given a greater chance of being selected.

Double bridge experiment

To illustrate the basic principle in self-organizing in the foraging behavior of ants and how the shortest path is found by use of positive feedback and a stochastic state transition rule, we have included a simple example called the double bridge experiment[5].

The experiment

Then setup of the experiment is as follows.

A food source, represented by a vertex D, is located a small distance from the anthill represented by the vertex A. There exists two distinct paths from the A to D; We denote the two paths: ACD and ABD with w(ACD) = 1 and w(ABD) = 2. As a graph, the situation could be represented as shown in Figure 1. Labels on the edges are the edge weight/the trail value. We will use this as a basis on determining what state the ants move to.

[5] The example is an adaptation of the example used in [Colorni et al., 1996]

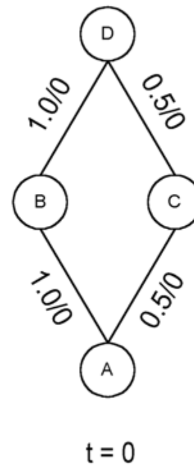

Figure 1: Double bridge experiment as a graph. Edge labels are weight/trail value

Assumptions

Because this experiment is meant to be simple, we make some assumptions to simplify the behavior of the ants.

- The ants use delayed-trail-update in which the ant deposits pheromone – after a complete walk – on all edges in the created path.
- Pheromone never evaporates.
- Ants have only one task: to find food. When an ant finds a food source it backtracks on the walked path depositing pheromone on all the edges and then it dies.
- The amount of trail deposited on an edge (X,Y) by an ant *a* that has constructed a path p_a is calculated by a function called a *trail update rule* defined as:
  newTrailOnEdge(X,Y) = *trailOn*(X,Y) + (1 / $w(p_a)$)
- All ants start at the anthill A.
- All pheromone values are initially 0.
- Ants use restricted walks when looking for food. They can only select between the B or C vertex when starting in vertex A. When this is done the rest of the path is determined.
- The stochastic state transition rule for these ants is very simple. Again since we have restricted walks, the only choice ants can make is between vertex B and C. From any of these two vertices they can only choose vertex D. Since they start in vertex A, they can never choose it.
  From this follows the following probabilities (*p*(W) is the probability of going to vertex W):
  p(A) = 0
  p(B) = *trailOn*(A,B) / (*trailOn*(A,B) + *trailOn*(A,C)),
  if *trailOn*(A,B) = 0 then *p*(B) = 1/2
  p(C) = 1 – *p*(B)
  p(D) = 1

This demonstration of how ants converge to use the shortest path will follow the conventions of the ACO metaheuristic presented in the section [*The Ant Colony Optimization metaheuristic*, p. 14]. In ACO ants are created in generations, and an entire generation of ants creates solutions, and when this is done, all the ants in the generation updates the values of the pheromone trails. Therefore we imagine a generation of ants, where all ants move such that each ant in a generation can only detect the pheromone that the previous generation deposited.

The experiment is run with a generation size of $a = 60$ ants, and it will run for $t = 4$ generations. Each generation is depicted on Figure 2. The label on the edges is *weight/trail value*. In the bottom of each graph is denoted the generation and on both sides of the vertex A is depicted the percentage of the generation of ants that chose the given branch. The number to the left of vertex A denotes the proportion that used the path ABD, and the number to the right of A denotes the proportion that chose ACD.

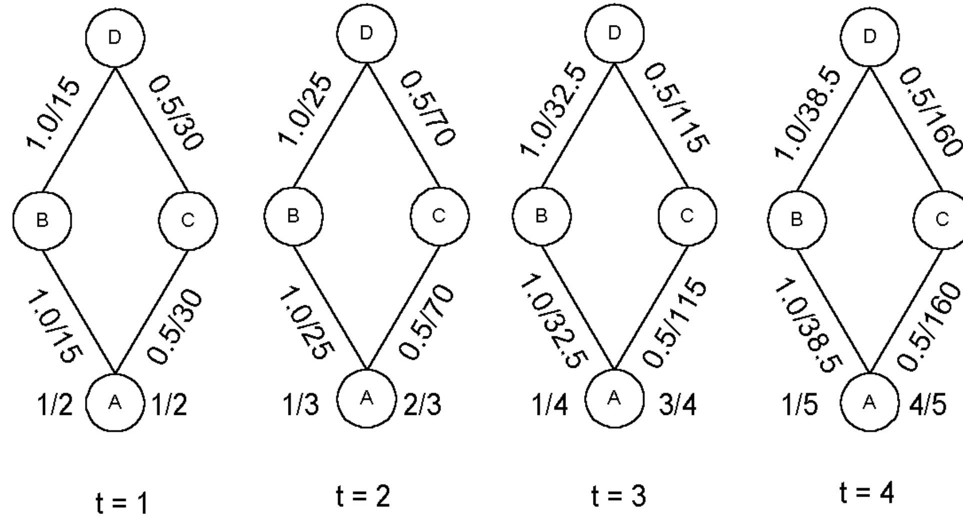

Figure 2 : Movement of ants in double bridge experiment over time.

In the first generation ($t = 1$), see Figure 2, an equal number of ants (30) will have traversed each path since there exists an equal amount of trail on the edges (A,B) and (A,C).

Each ant that chose path ACD will deposit a trail value of 1 on the edges (A,C) and (C,D) while the ants that chose ABD only deposits 1/2 trail. This leaves the graph in a state where there is twice the amount of trail on the edge (A, C) compared to (A, B), thus increasing the probability that the next generation will choose the short path ACD.

The second generation now uses the pheromone values deposited by the first generation to create new solutions. It should be evident from the stochastic state transition rule that 1/3 of the ants will chose the long path ABC, while 2/3 will chose the short path ACD.

The third generation uses the pheromone that both the first and the second generation have deposited. If we use this generation as an example, the probability distribution for the ants can be calculated by use of the stochastic state transition rule as follows:

For an ant starting in vertex A at $t = 3$ (this means using the trail values from the second generation), the probability of choosing to add vertex B to the path is given by:

$p(B) = 25/(25+70) = 5/19 \approx 1/4$

The probability for adding vertex C is thus $1 - p(B) \approx 3/4$

This means that a total of 15 ants will move by the path ABD and 45 ants will move by the path ACD. The update that each of the 15 ants on ABD will deposit is calculated by:

$newTrailOnEdge(A,B) = newTrailOnEdge(B,D) = 25 + (1/2 \cdot 15) = 32.5$ (15 ants used the path ABD)
$newTrailOnEdge(A,C) = newTrailOnEdge(C,D) = 70 + (1/1 \cdot 45) = 115$ (45 ants used the path ACD)

The shift towards there being relatively more trail on the path ACD continues in fourth generation and further generations would continue to make the system converge even further towards only choosing the shortest path.

The example given above is highly simplified, since the ants can only make a single stochastic choice, but it illustrates how ant algorithms find the shortest path by the use of positive feedback and a stochastic state transition rule. Thus it exemplifies how self-organization can emerge by use of positive feedback and a stochastic state transition rule.

A problem regarding the ant algorithm described above is that it will most likely stagnate. Assume that just before the fourth generation of ants constructs their solutions, the length of path ABD is reduced to 0.5 and thus is now the shortest path (the numbers are chosen to illustrate a point). 1/5 of the ants would still construct the path ABD while 4/5 would construct the path ACD.

If we recalculate the amount of pheromone that the ants in the fourth generation will deposit on each path we get:

$newTrailOnEdge(\mathrm{A,B}) = newTrailOnEdge(\mathrm{B,D}) = 32.5 + ((1/0.5) \cdot 12) = 32.5 + 24 = 56.5$
$newTrailOnEdge(\mathrm{A,C}) = newTrailOnEdge(\mathrm{C,D}) = 115 + ((1/1) \cdot 48) = 115 + 48 = 163$

What is relevant to notice is that even though the path ABD is now the shorter path, the system still reinforces the path ACD with more pheromone, thus continuously driving the system towards the now longer path. This situation is referred to as stagnation, and generally describes the situation where an ant algorithm is unable to discover new solutions. The main method of avoiding stagnation is negative feedback and local heuristics.

We could implement a more sophisticated state transition rule, which included knowledge that we have about the problem that is to be solved. In the example we could make the ants prefer to select the shorter of the two edges from A, thus rewriting the stochastic state transition rule to also use the length of the edges as a parameter.

Summary

In the above we have described a simple experiment that illustrates how ants are capable of finding the shortest path in a simple graph by the use of positive feedback and a stochastic state transition rule. We have indicated that trail-following behavior can lead to stagnation, leaving the system unable to explore new, potentially better solutions. We indicated how negative feedback and local heuristics can help avoid stagnation, and thus allow the ants to discover new solutions.

The Ant Colony Optimization metaheuristic

The Ant Colony Optimization (ACO) metaheuristic [Dorigo & Stuetzle, 2000] is a recently proposed discrete optimization metaheuristic for solving NP-hard problems. In the following we will present a formal definition of the problem representation on which ACO algorithms work. We will describe formally how solutions are constructed in the representation and present the ACO metaheuristic as pseudo-code.

The central properties of ACO are based upon the self-organized collective foraging behavior of ants. The key property collective in the foraging behavior of ants is their ability to find shortest paths between the location of their anthill and the location of food sources. When the ants move on a path between their anthill and the location of a food source they lay a pheromone trail. Other ants can then follow these generated paths of pheromone trails. This means that ants tend to converge on the same path, as illustrated in the example above.

In ACO, solutions are constructed repetitively by adding solution components to partial solutions stochastically. Solutions are constructed by taking into account (i) heuristic information when adding solution components (if available), and (ii) (artificial) pheromone trails which change dynamically based on the experience of the ants. Stigmergy (as defined earlier) handles the propagation of experience between ants.

As related in the brief historical survey above, a number of algorithms following the ACO metaheuristic have been presented in recent years. The problems solved can be divided into two domains – static and dynamic problems. The basic properties of the algorithm do not change because of this, and the problems are in both cases represented by graphs.

The ACO metaheuristic has been used as a template for algorithms that has achieved world-class performance[6] in both domains. Nevertheless, because of the ants' inherent ability to adapt to changes in the environment, we find that ACO algorithms are especially well suited to solving for example routing problems in networks, which constitute a dynamic problem domain.

We begin by formally defining the problem representation and construction process (the behavior of the ants) in ACO and finally the basic form of an ACO algorithm will be presented in pseudo-code.

Problem representation

The ACO metaheuristic is targeted towards optimization problems that can be solved as shortest path problems on graphs. Ants construct solutions to optimization problems in much the same way as their natural cousins. They move around in a graph and use stigmergy to communicate their experiences. In order to ensure consensus about the nomenclature used to describe this process we will give a definition of an optimization problem in this context and explain how ants construct solutions to an optimization problem by moving on what is referred to as a construction graph.

Given an optimization problem $O = (S, f, \Omega)$, where S is the set of solutions, f is the objective function used to assign a quality to a solution and Ω is a finite set of constraints, the goal of O is to find a globally minimal[7] solution $f_{\min}(s), \forall s \in S$ that satisfies the constraints Ω.

In a graph representation of an optimization problem, we denote the vertices in the graph as components. The finite set $C = \{c_1, c_2, ..., c_n\}$ is the set of components by which solutions can be created. A sequence of components $x = \langle c_{n1}, c_{n2}, ..., c_{ni} \rangle$, where $c_{n_x} \in$ C is denoted a state. The set of all possible states x we denote X – in other words $X = C^*$.

We now define the set of feasible states $\overline{X}$ as the set of states that do not violate the constraints Ω, thus $\overline{X}$ = $\{x \in X | \Omega(x) = 1\}$, where

$$\Omega(x) = \begin{cases} 0, & \text{if } x \text{ violates } \Omega \\ 1, & \text{otherwise} \end{cases}$$

The set of solutions S is a proper subset of $\overline{X}$ because $\overline{X}$ also contains partial solutions – that is, states that have not violated any constraint – but is not contained in S – as it still lack a number of components to constitute a solution.

We can construct solutions by constructing sequences of components from C. This is basically what ants do. When an ant constructs a solution by adding components to a partial solution it has to check the constraints Ω to ensure that adding the next component does not violate the constraints. This restricts the movement of ants to only generate *feasible* states which is reasonable since $S \subset \overline{X}$.

Let us define the *construction graph* $G = (C, L)$, where the vertices are the components C and the edges are the set L – the set of connections between components. Initially let G be defined as a complete graph – thereby defining L as all possible pairs of components (see the section [*Graph theory*, p. 5]). This would be reasonable – we could just have the ants check the constraints every time they add a new component c_x to their sequence of components.

But we have the opportunity of implementing some of the constraints in another way. Typically a subset of the constraints can be implemented by removing edges from L – thereby directly removing non-feasible solutions.

---

[6] According to [Dorigo & Stuetzle 2000, p. 1]

[7] Note that an optimization could just as well have been defined as a maximization problem.

Take routing from a given sender to a given receiver in a package switched network as an example. We define the optimization problem *O* as the routing problem, which is to find the shortest path to use when routing a package from source to destination. Let a set of routers *R* and a set of connections called wires *W* be given, and let the constraints be that packages cannot move over non-existing wires. Define the routers *R* as the components *C* and the connections between routers *W* as the edges *L*. Thereby the given constraints have already been implemented – namely that ants cannot move over non-existing connections. A termination criterion for ants "Has the ant arrived at the destination?" determines whether it has created a full solution belonging to *S*.

The quality function *f* returns for each possible path the time of routing over this path.

The routing problem can be stated as finding $s : f_{\min}(s), s \in S$. Ants then create a solution by traversing the construction graph $G = (R,W)$.

As an example of a problem constraint that cannot easily be implemented directly in the graph, extend the constraints with the following: A package cannot move back to a router that it has already visited. This constraint cannot be implemented directly in the graph defined above – as it requires that edges cannot exist *if* some condition (based on the *former* path) is true. But this constraint is used widely in ant algorithms (e.g. in solutions to TSP and in our routing algorithm). A new graph with components consisting of all possible former paths *could* be defined – thereby again making it able to represent all constraints in the graph – *but* it is much easier to make the ants check a *taboo list* (a problem constraint) as they move through the graph *G*.

Construction of solutions

In the following section we will formalize the solution construction process.

The ants stochastically construct solutions by making walks on the construction graph $G = (C, L)$. Components $c_i \in C$ (vertices) and connections $(c_i, c_j) \in L$ (edges) denoted l_{ij} can have an associated artificial *pheromone trail value* τ that allow for stigmergic interaction with other ants. Pheromone trail value will in the following just be referred to as *trail value*. All trail values initially contain a small positive value (τ_0).

The components and connections can also have a *heuristic value* η that can represent some a priori information regarding the problem or a run-time local information value maintained by another source than the ants. Typically η represents an estimate, made from a local perspective, of the quality of adding a particular component to a partial solution.

Every ant *k* has a memory T_k that stores information about the components it has added so far. This memory can be used to: (i) implement constraints (cf. above), (ii) to evaluate the solution found (by the objective function *f*) and (iii) to retrace the path of components backwards to facilitate delayed trail update once a solution is generated.

Further every ant *k* is assigned a starting state x^k_{start} and a set of termination criterions e^k. Typically the starting state is just a single component – which simply translates to an ant starting with an empty memory in a vertex in the construction graph *G*.

When in a state $x_r = \langle x_{r-1}, c_i \rangle \in \overline{X}$ it tries to make a move to a vertex c_j in its feasible neighborhood (as defined by the graph (*C,L*)). In other words is it tries to make a state transition to $\langle x_r, c_j \rangle \in \overline{X}$. It selects the move by a probabilistic decision rule earlier referred to as the stochastic state transition rule. This rule can be a function of (i) locally available trail and heuristical values (τ, η), (ii) the memory of the ant, and (iii) the problem constraints Ω.

The basic ACO algorithm in pseudo-code

The formal definitions above states that ACO algorithms constructs solutions by constructing paths in a graph. This process can be stated in pseudo-code, which is the intention of the following section.

Further discussion of ant algorithms can be done by simply filling out the blanks in the pseudo-code, meaning that a number of procedures must be implemented to solve a specific problem, but otherwise the ACO algorithm is used as-is.

In the literature, we have encountered a number of presentations of the ACO metaheuristic, which differ slightly in the details, but are identical in intent. This description of the ACO algorithm is an adaptation from [Di Caro et al., 1999, p. 10]. It has been altered to conform to our implementation of the ACO metaheuristic in Java see Appendix B [Class: ACOMetaHeuristic].

As can been seen in the pseudo-code below the central procedures are *run_ACO()* and *ants_generation_and_activity()*. The algorithm runs t_{max} iterations (also denoted as generations) with *m* ants in each iteration. This is done iteratively, but updates are not performed until an iteration is complete so the ants are therefore not able to perceive the changes made by other ants from the same generation.

```
procedure run_ACO() {
    while ( time++ < t_max ) {
        ants_generation_and_activity();
        pheromone_trail_update();
        daemon_actions();
    }
}

procedure ants_generation_and_activity() {
    while ( no_of_ants++ < m ) {
        new_active_ant();
    }
}

procedure new_active_ant() {
    ant = create_new_ant();
    position = get_starting_vertex();
    initialize_ant( position );
    while ( not( in_target_state( ant ) or could_not_reach_target_state( ant ) ) ){
        A = read_ possible_routes_from_current_vertex( ant );
        P = compute_transition_probabilities( A, ant, Ω );
        next = stochastically_choose_path( A, P );
        ant_move_to_next( next );
        if ( online_step_by_step_pheromone_update ) {
            deposit_trail_on_the_last_[edge/vertex]( ant );
        }
    }
    if ( online_delayed_phero_update ){
        deposit_trail_on_all_[edges/vertices]( ant );
    }

    if ( could_not_reach_target_state( ant ) ) {
        ant_deaths++;
    }
    else {
        // Update the current shortest tour
        if ( length_of_tour < shortest_tour ) {
            shortest_tour = length_of_tour;
        }
    }
}
```

Algorithm 1 : The basic ACO algorithm in pseudo-code

Each ant attempts to construct a solution in the procedure *new_active_ant()*. The central procedure are *compute_transition_probabilities(A,* ant*, Ω),* that calculates the probability distribution that the ant uses to select the next vertex to add to the solution with the call *stochastically_choose_path(A, P)*. A trail is deposited if it is specified that an ant should use step–by–step update. The amount of trail is calculated on local estimate of quality.

This is repeated until: the ant reaches its termination criteria's, arrives in its target state, or concludes that the target state cannot be reached. The construction is done probabilistically with the probability for new choices given by a function *compute_transition_probabilities*(A, ant, Ω) of a *trail value* τ and *heuristic desirability* η.

If an ant succeeds in creating a solution and it is indicated that the ant should use *delayed_trail_update,* the ant deposits a trail amount on all visited edges/vertices calculated based on the quality of the solution constructed.

After all the ants in an iteration have constructed their solution, the procedure *pheromone_trail_update()* is called. It updates all the trail values by adding the trail to the existing amount. If trail evaporation is used it is also implemented in the update function.

The update that happens is thus a function of the existing trail value plus the deposited trail value, together with the trail evaporation factor ρ . The update is commonly referred to as *pheromone trail update*.

An additional method of implementing negative feedback would be to punish ants that were not able to construct a solution. The pseudo-code specifies the expression *antDeaths++*, but any amount of complexity could be placed here.

It is important to note that the deposits made by *deposit_trail_on...()*-function calls in Algorithm 1 only are perceivable by other ants when this *pheromone_trail_update()* has been run.

To implement an ACO algorithm it now suffices to implement the missing functions.

An ant algorithm should centrally specify:

- A stochastic state transition rule (*compute_transition_probabilities(A,* ant, Ω *)*)
- Trail deposition rule(s), (*deposit_trail_on_...* – functions) depending on the chosen form of update.
- A trail update rule (*pheromone_trail_update()*)

Further it should specify a function to specify ant starting position (*get_ant_vertex()*), whether to update at each step (of the ant) (*step_by_step_trail_update = true*) or at the end of an ant run (*delayed_trail_update = true*).

The implementation should also specify the values and the procedures that determine termination of:

(i) the algorithm - t_{max},

(ii) a generation – *m*,

(iii) an ant run - *in_target_state(* ant *)* and *could_not_reach_target_state(* ant *)*.

The procedure *daemon_actions()* is in a few ACO-algorithms used for making actions based on global knowledge a single ant cannot possible have. An example could be an offline extra trail update of the *q* best tours found in a cycle.

Summary

In the above we have given a formal definition of the problem representation that ant algorithms work on. We have also given a formal definition of the solution construction process and shown how this is implemented as pseudo-code.

Metaheuristics

In this section that will be the last concerning the ACO metaheuristic, we will further specify the concept of metaheuristic, and try to place the ACO metaheuristic in relationship to other general metaheuristics.

Hard optimization problems arise in nearly every field of science, management or engineering. Problems – with sizes of practical interest – that cannot be solved within reasonable time constraints due to algorithmic complexity of known algorithms and/or space-constraints of the problem. When given some objective function that quantifies the viability of a solution, typically the approach is to invent and introduce heuristics to guide the search for a good solution.

Some heuristics have a form that makes them generally applicable to a broad class of optimization problems. These “general-purpose”-algorithms are sometimes called *metaheuristics*. Examples include Genetic Algorithms (GA) and Taboo Search (TS). In [Gutjahr 2000] the author names two broad categorizations that

can be used to classify metaheuristics: *Iterative* versus *constructive* heuristics and *single-run* versus *repetitive* heuristics:

Iterative metaheuristics starts with some complete and feasible solution and makes systematic changes to this solution in order to improve the quality of the solution. Local Search (LS) is a typical example of an iterative metaheuristic, as it steps through the components of the given solution and tries modifications within some local neighborhood of each component.

In contrast to this constructive metaheuristics constructs a solution by adding components to a partial solution until a complete and feasible solution is found. Greedy Heuristics (GH) are constructive in the sense that they linearly build a solution by repeatedly adding a component, by some local notion of viability (e.g. shortest edge connected to a neighbor-vertex of graph).

In single-run metaheuristics, an algorithm only continues to run until a solution is generated. That is, in single-run metaheuristics a generated solution is not used to generate new better solutions.

Repetitive metaheuristic algorithms let the user have some control over how much computation is done, and the quality of the solution improves as the computation time increases. Experience gained while generating solutions is used as a basis for generating new solutions.

Examples of single-run metaheuristics could be LS – a solution is presented when a local optimum is reached – or GH that stops, when the construction process is finished.

GA is an example of a repetitive heuristic. Typically this type of algorithms uses *stochastic state transitions* as a way of avoiding local optimums.

By distinguishing metaheuristics by these properties we come to four possible (rough) categories:

1. iterative, single-run (LS)
2. constructive, single-run (GH)
3. iterative, repetitive (GA)
4. constructive, repetitive (ACO metaheuristic)

Gutjahr, the author of [Gutjahr 2000], points to the fact that most metaheuristics fall into the categories 1 through 3. He further notes that considering that Greedy Heuristics are available for most optimization problems it would seem an obvious possibility to try to improve on the solutions generated by defining *repetitive* metaheuristics that use local heuristics.

The Ant Colony Optimization metaheuristic is in essence such a metaheuristic. It uses local heuristics to guide the search for good solutions and repetitively generate new solutions using stochastic decisions in a component-wise construction of feasible solutions. Thereby it clearly constitutes a constructive, repetitive metaheuristic.

We must emphasize that we find the classifications above to be useful primarily from an analytical and descriptive perspective. We do not expect new metaheuristics to be invented by examining these a posteriori categories, but it is interesting to notice that the ACO metaheuristic fits into the least populated category.

Ant System

In the following section we will discuss an archetypical ant algorithm Ant System (AS). AS was the first ACO algorithm that was developed in 1991 [Colorni et al., 1991]. This ant algorithm prototype was implemented to solve the Traveling Salesman Problem (TSP). The reason for choosing TSP was that it is a NP-complete problem and it is a well-studied “benchmark” problem. Consequentially, there are many algorithms and results to compare with.

The nature of AS means that it can be described from the missing functions that fills out the basic ACO algorithm schema. As such, it is an idiomatic example of how an ACO algorithm looks and works. Furthermore, it represents the first studies and implementations of our project. As such it has been an

inspiration to us in our creation of our routing algorithm [*Ant Routing System - an ACO network routing algorithm*, p. 35].

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) can be stated as follows:

Given a weighted directed graph $G = (V,E)$ consisting of a set of vertices and a set of edges with a weight function $w : E \rightarrow \mathbf{N_0}$, find the Hamiltonian cycle of minimum weight.

Basic idea in the algorithm

The central parts of implementing an ACO algorithm is, as noted above, specifying a *stochastic state transition rule* and a *pheromone trail update rule* together with a *trail deposition rule*.

In AS ants will start at a randomly determined vertex and continue to move to another vertex until they have completed a tour (the ants work under the assumption that the graph is complete such that non-existing edges are represented as having infinitely large weights). Further, the ants contain a 'memory', which is used to implement a taboo-list so that only feasible solutions are generated.

Dorigo et al. developed three different versions of AS, which differ by the way pheromone was placed. In two of the versions, pheromone is deposited while the solution is being built (*step-by-step*). In the third, the pheromone is deposited after a complete tour (*delayed*).

In Algorithm 1 (p. 18) it is possible to deposit pheromone in both ways. Other published versions of ACO do not include the option to deposit trail *step-by-step*, because this method generates solutions of lesser quality [Bonabeau et al., 1999]. This also fits well with the fact that the name AS now denotes the version, where the pheromone is deposited after a complete tour (and the two other versions are more or less ignored). In AS the depositing of the pheromone is proportional to the inverse of the length of the ant's tour. This has the effect that shorter tours are rewarded with larger amounts of trail.

Stochastic state transition rule

The most complex part of the algorithm is the movement of the ants. Given an arbitrary ant k at vertex i the next vertex j this is decided based on three parameters:

- A taboo list T_k containing all the visited vertices and visited edges in this run, if vertex j is in the taboo list, the possibility of choosing j is 0. So the ants are forced to create a valid tour.
- The heuristic desirability η_{ij} – in AS known as *visibility* – is defined as the inverse of the weight between i and j.
- Virtual pheromone trail $\tau_{ij}(t)$ – also known as the learned desirability – is the pheromone trail on the edge connecting vertex i and j at time t.

Dorigo et al. sets up the following stochastic state transition rule. It constructs the probability distribution in the basic ACO algorithm. $J_i^{T_k}$ denotes the restricted neighborhood of ant k on vertex i:

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in J_i^{T_k}} [\tau_{il}(t)]^{\alpha} \cdot [\eta_{il}]^{\beta}}, & \text{if } j \in J_i^{T_k} \\ 0, & \text{otherwise} \end{cases}, \quad (4)$$

- which is the possibility for ant k at time t to go from vertex i to vertex j, where α and β are adjustable parameters for the relative weight between visibility and pheromone trail.

Trail deposition and pheromone trail update

The pheromone trail update rule Dorigo et al. used in AS is defined for every edge *(i,j)* as

(5) $$\tau_{ij}(t) \leftarrow (1-\rho)\cdot\tau_{ij}(t) + \Delta\tau_{ij}(t),$$

- where $\Delta\tau_{ij}(t) = \sum_{k=1}^{m}\Delta\tau_{ij}^{k}(t)$. That is the sum of pheromone deposited on the edge by all ant at iteration t. The pheromone deposited by each ant depends on the quality of the solution that the ant has constructed. Ant k deposits the following pheromone amount $\Delta\tau_{ij}^{k}(t)$ after a complete tour on each edge (i,j) that it has used in the solution:

(6) $$\Delta\tau_{ij}^{k}(t) = \begin{cases} Q/L_k(t), & \text{if } (i,j)\in T_k(t) \\ 0 & \text{, otherwise} \end{cases},$$

- where Q is a parameter, which should have a value of same magnitude as the length of an optimal solution (although this can be disregarded). $T_k(t)$ is the tour done by ant k at the iteration t and $L_k(t)$ is the length of this tour.

Performance of AS

The complexity of AS is $O(t \cdot n^2 \cdot m)$ where t is the number of iterations in the algorithm, m is the number of ants used in each iteration, and n is the number of vertices. For a more thorough description of AS see chapter 2 in [Bonabeau et al., 1999].

The initial version of AS (with the values of the parameters: $\alpha = 1$, $\beta = 0.5$, $m = n$, $Q = 100$, $\tau_0 = 10^{-6}$ [Bonabeau et al., 1999]) did produce results similar to general-purpose heuristics (e.g. genetic algorithms) on problems of a small size but did not scale up well. On problems of dimensions larger than 50 vertices AS never reached the best known solutions although it quickly converged to good solutions. Still the performance was far from the performance of specialized algorithms for TSP.

An example run

We have made a complete implementation of the Ant System algorithm, and performed a series of test. We have chosen to include only an example in order to demonstrate the convergence of an archetypical ACO algorithm. The test is performed on a benchmark graph bayg29[8], which has 29 vertices. The best solution to TSP in this graph is 1610, and Ant System found the best solution to be 1652 after the 1642nd ant.

[8] See TSPLIB at http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/

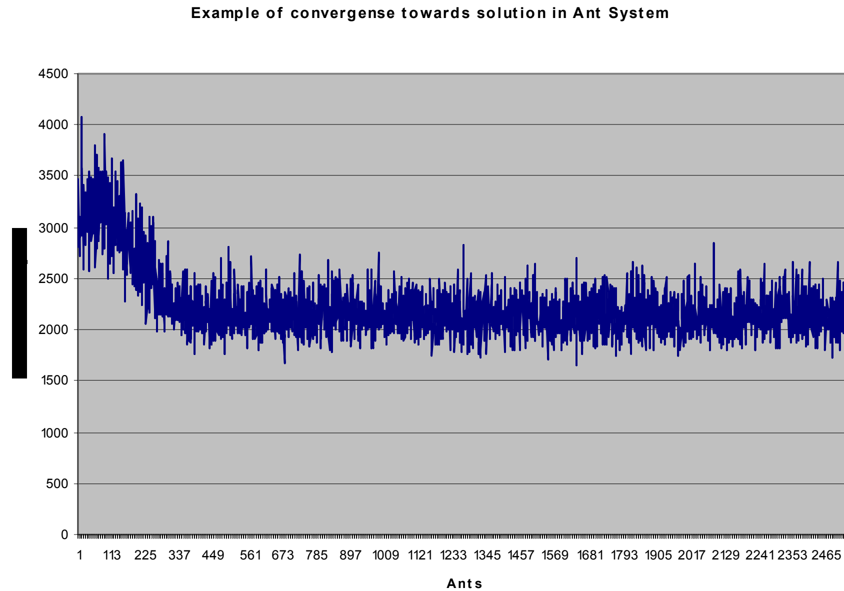

Figure 3: Example of convergence by Ant System

Convergence proof of limited version of AS

An important result in the field of ant algorithms was published in [Gutjahr 2000]. Gutjahr gave a formal proof that under certain conditions a slightly limited version of the AS (called Graph-based Ant System) can be made to converge to the optimal solution of the given problem with a probability that can be made arbitrarily close to 1.

The proof is made upon the main assumptions that (i) the optimal solution is unique, (ii) no heuristic values are 0 along the path that corresponds to the optimal solution, and (iii) that trail updates are only made, when better solutions are found.

The first condition is with regard to most practical applications the most restrictive, but Gutjahr notes that he has preliminary results that show that this condition can be dropped.

Condition (ii) is necessary as the formal proof has to take into account that heuristic values can in fact be misguiding in regard to the solution. In fact, it suffices to require that no heuristic values are 0. The third condition constitutes what we have seen published as "the elitist strategy" used for example in the Ant Colony System by Dorigo and Gambardella (see the next section).

As it is furthermore required in the proof that the problem is a static optimization problem (meaning that edge costs do not vary), this proof is not directly related to our project, but it is nevertheless an important achievement as this is the first proof of convergence for a generic ACO algorithm [Di Caro et al. 2000, p. vi].

The improved Ant System: Ant Colony System

As an ending note, it is interesting to note, that Dorigo and Gambardella introduced an improved algorithm for TSP – Ant Colony System (ACS) in 97 [Bonabeau et al., 1999]. The improvements in ACS constitute four major modifications of AS, which specialize the algorithm toward solving TSP:

- A transition rule that forces an ant with probability q to move deterministically to the locally most desirable vertex (this deprives some of the exploration ability from the ants).
- A different update rule that only allows ants that made the best tour since the beginning of the algorithm to update the pheromone trail globally (this rule is sometimes denoted the elitist strategy).

- Local immediate decrements of pheromone trail that is made every time an ant traverses an edge. This rule makes the edges less and less desirable for other ants in the same iteration. This makes the ants explore more otherwise unvisited edges.
- Use of a candidate list that makes the ants first examines some choices (those in the candidate list) for the next vertex before others. A candidate list is an often-used data structure when trying to solve large TSP instances.

Dorigo and Gambardella also improved on ACS by using a local search procedure (restricted 3-opt) where 3 edges are swapped iteratively until a local minimum is reached. Improved with a local search procedure the performance of ACS was equal to the genetic algorithm STSP that won the First International Contest on Evolutionary Optimization [Bonabeau et al., 1999].

Summary

In this section we have presented an archetypical example of a use of the ACO metaheuristic – the Ant System (AS). We have described the algorithm by defining its constituting functions. We have briefly demonstrated that AS is able to converge to a good solution by including the results from an example run made by our implementation of Ant System.

Network Model

In this chapter, we will describe the abstract network model, used as a basis for the investigation of an ant routing algorithm. As the main purpose of this report is to present our investigations of an ant-inspired routing algorithm, this chapter will *not* describe the implementation of this model. In the chapter [*Event driven network simulation*, p. 32], we will briefly touch how the model has been implemented by an event-driven simulation. In the chapter [*Ant Routing System - an ACO network routing algorithm*, p. 35] the routing algorithm we have constructed will be presented. The chapter [*Description of selected empirical observations*, p. 42] we show how this routing algorithm behaves in a simulated network based on the model presented in this chapter.

The model is a graph-based representation of a simplified version of the network layer in the OSI reference model [Tannenbaum, 1996]. The model was designed to support studies of ant-inspired algorithms in a dynamic domain; more specifically ant inspired routing in a package-switched point-to-point network such as the Internet.

Formal Definition

We start by giving a formal definition of a network (network layer) in much the same way as a graph.

Let (R, W) denote a given network N, where $R = \{r_0, ..., r_n\}$ is a finite set whose elements are called routers and $W = \{w_0, ..., w_m\}$ is a proper subset of $R \times R$ whose elements are called wires. The wire between router r_i and r_j is denoted (r_i, r_j).

Each wire in the network also have an associated delay given by a delay function $d : W \rightarrow \mathbf{R}^+$. E.g. $d(r_i, r_j)$ is the delay of the wire (r_i, r_j).

Let s, t be two given routers of (R, W). A path p from s to t in (R, W) is a sequence of routers of the form $p = \left\langle r_0, r_1, r_2, ..., r_k \right\rangle$ such that $r_0 = s$, $r_k = t$ and $(r_{i-1}, r_i) \in W$ for $i = 1,2,\ldots,k$.

(r_i, r_j) are ordered pairs for any $(r_i, r_j) \in W$, so $N=(R,W)$ is directed in the sense that each wire has a source router and a destination router.

Let s, t be two given routers of (R,W). A package q can be sent from s to t over a path p, if p is a path in (R,W) as defined above. The length of the path $p = \left\langle r_0, r_1, r_2, ..., r_k \right\rangle$, where $r_0 = s$ and $r_k = t$, is defined as the time it takes to route package q from s to t.

As opposed to vertices in a graph, routers in a network have associated delays as well as wires. Therefore, we cannot simply add up the delay of the wires along the path p to find the time of routing p. Furthermore, the delay of routers changes over time based on the number of packages they handle.

To facilitate a more precise discussion of the time taken to route a package, certain aspects of router design and routing in our network model will be further explained.

Routing

Routing defines the task on every router of evaluating the destination address of an incoming package and selecting the next router to send the package to.

Routers can be buffered or unbuffered; When buffered, a router contains queues to store packages until they can be processed, which reduces package loss.

Before we continue this discussion, it might be prudent to present some assumptions that we have made in order to simplify the task of modeling the network layer as a dynamic graph.

1. The size of the packages[9] is fixed, and all routers can handle packages of this size.
2. All routers are buffered, with buffer capacity ensuring at least one package can be held in the queues.
3. Routers have a fixed routing speed, which defines the average time it takes to route a package.
4. Wires in the model are the mediums over which packages are transferred. The weight of a wire represents a delay, indicating how long it takes to send a standard sized package over it.
5. Internal transfer speeds in routers between buffers and switching fabric are zero.

The first assumption greatly reduces the complexity in the network model. When packages have a fixed size and all routers can handle packages of this size, issues such as fragmentation of packages can be ignored, but also the need to consider varying queue capacity, since we define a queue's capacity as the number of packages that it can contain.

When all routers are buffered, we can always assume the existence of a queue, which removes a special case in the definition of length of paths in a network, since it is not needed to test whether a router *is* buffered.

The third assumption is stated to abstract away details regarding the implementation of the switching fabric in a router (the switching fabric is the entity in a router that handles the calculation of the next router on the packages path).

The fourth assumption follows naturally from assumption 1. Since package sizes are fixed, we can reduce the wire bandwidth to a variable indicating the time it takes to send one package.

The last assumption states that transfer speeds inside a router are ignored. Internal package movement between queues, buffers, wires and switching fabric happens infinitely fast.

A rough sketch of a router in our network model is included in Figure 4.

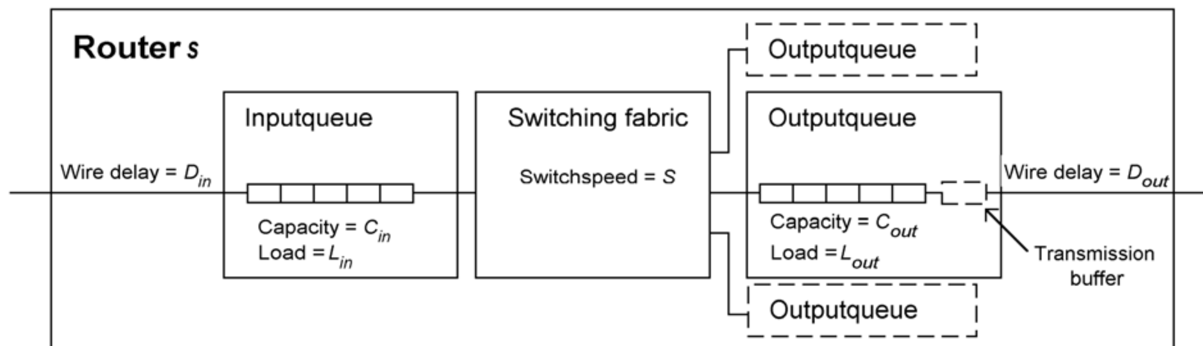

Figure 4 : Sketch of a router

A router contains (as can be seen on Figure 4) – an *input queue* with a max-capacity C_{in}, a *switching fabric* with a speed S, a number of *output queues* with a capacity C_{out} and for every output queue a *transmission buffer* that is used by a package waiting to be sent over a wire. The functions L_{in} and L_{out} return at any time t the number of packages in input queues and output queues respectively. Any number of wires can be connected to a router.

A relevant detail outlined in Figure 4 is that routers use separate output queues for each outgoing wire, but all incoming wires share the same input queue.

[9] The word packet is the official reference name for a data unit at the network layer, see [Tannenbaum 1996, chapter 5]. In the following, we will use the word package to refer to these entities.

Finally, we define the start and end-point for the routing of a package to be the switching fabric of a router, this means that the package routing is initialized by being processed by a switching fabric at the source router and that it ends in the switching fabric at the destination router.

Formal definition continued

We can now continue to give a formal description of our graph represented network model, which encapsulates all the elements in the network model.

(Note that the nomenclature used on Figure 4 above is changed slightly. To simplify the reading of the formulas below we have elected to use more self-explaining names – e.g. router speed is denoted *speed*(r) instead of $S(r)$, $C_{in}(r)$ has been extended to *capIn*(r) and so on).

We now define that -

Each router $r \in R$ has an associated switch speed, given by the function $speed : R \rightarrow \mathbf{N}$. E.g. $speed(r_i)$ is the switching speed of router r_i.

Each router $r \in R$ also has an input queue, which has an associated *capacity* function $capIn : R \rightarrow \mathbf{N}$. E.g. $capIn(r_i)$ is the capacity of the input queue on router r_i.

Each router $r \in R$ also has a *current input queue load* function $loadIn : R \times T \rightarrow \mathbf{N_0}$ with T being the set of positive real numbers we use to denominate time. That is, $loadIn(r_i,t)$ is the number of packages in the input queue on router r_i at time t. (Note that if a package arrives in a queue at time t it *will* be counted).

Each router $r \in R$ also has a function $timeSinceLastRoutingStarted : R \times T \rightarrow \mathbf{N_0}$ with T being the set of positive real numbers we use to denominate time; $timeSinceLastRoutingStarted(r_i,t)$ returns the time since the switching fabric in r_i last started to work upon a package. Specifically, it returns a value in the range $[0;speed(r)]$; 0 if no package is being worked upon, and a value in the range $[0;speed(r)]$, if a package is being switched.

Each wire $(r_i,r_j) \in W$ in the network has an associated delay given by a delay function $d : W \rightarrow \mathbf{R}^+$. E.g. $d(r_i,r_j)$ is the delay of the wire (r_i,r_j).

Each wire $(r_i,r_j) \in W$ in the network also has an associated output queue, which has a capacity function $capOut : W \rightarrow \mathbf{N_0}$, $capOut(r_i,r_j)$ is the capacity of the output queue at router r_i that buffers packages being sent over wire (r_i,r_j).

Each wire $(r_i,r_j) \in W$ in the network has a *current output queue load* function $loadOut: W \times T \rightarrow \mathbf{N_0}$, with T being the set of positive real numbers we use to denominate time. E.g. $loadOut(r_i,r_j,t)$ is the number of packages in the output queue on router r_i at time t, that buffers packages over wire (r_i,r_j). (Note that if a package arrives in a queue at time t it *will* be counted).

Each wire $(r_i,r_j) \in W$ in the network additionally has a function $timeToBufferEmpty : W \times T \rightarrow \mathbf{N_0}$ with T being the set of positive real numbers we use to denominate time. The function $timeToBufferEmpty(r_i,r_j,t)$ at time t returns the time until the transmission buffer at the output queue for the wire (r_i,r_j) is empty. This value will minimally be 0 (if there is no package in the transmission buffer) and maximally the transmission delay of the wire (r_i,r_j).

Let p be a path of (R,W). Let q be a package sent over p at starting time t_0. The length of p and hence the time to route package q beginning at time t_0 is: $length(p,t_0)= routing(r_0, r_k, t_0)$. The equations [equations (7)-(11)] below define the routing-time function *routing(…)*.

$$(7)\qquad routing(r_0, r_k, t_0) = (\sum_{i=1}^{k} routingStep(r_{i-1}, r_i, t_{i-1})) + speed(r_k),$$

$$\text{where } t_i = t_{i-1} + routingStep(r_{i-1}, r_i, t_{i-1}), \text{for } i \geq 1,$$

- *routingStep(…)* is defined by the following set of equations:

(8) $$routingStep(r_i, r_k, t) = readyForRouting(r_k, arrivedAtInputQueue(r_i, r_k, processedByRouter(r_i, t))) - t$$

(9) $$processedByRouter(r, t) = t + speed(r)$$

(10) $$arrivedAtInputQueue(r_i, r_k, t) = t + d(r_i, r_k) \cdot loadOut(r_i, r_k, t) + timeToBufferEmpty(r_i, r_k, t)$$

(11) $$readyForRouting(r, t) = t + speed(r) \cdot (loadIn(r, t) - 1) - timeSinceLastRoutingStarted(r, t)$$

What is collected in the equations above is simply, that the length of p is equal to a summation of the lengths of each step in the routing.

Calculation of routing time - an example

To illustrate and explain the equations we have constructed an example, in which we show how to calculate for a single routing step [equation (8)].

We simplify this example by assuming that a router s sends a package q directly to a receiver-router u (no intermediate routers) and that s and u have only a single output and input queue (leave out the dashed lines in Figure 4). In essence this is the development of the formula for a single *routing* step as defined by the function *routingStep*(*s,u,t*).

Assuming that q is read from the input queue at s at time t_0, it will be put in the output queue for the wire connecting s with u at the time:

$$processedByRouter(s, t_0) = t_0 + speed(s) \ .$$

The package q must then wait in the output queue until all packages in front of it has been sent.

To calculate the exact time of the arrival at the input queue of u, we must include the fact that, apart from the packages in the output queue, a package could be in the process of being sent in the transmission buffer.

The newly arrived package q will arrive at the destination after an amount of time that should be calculated by something like *'number of packages in the output queue'* · *delay*(*wire*) + *'the time still remaining for the package in the transmission buffer'*.

The time it takes the package q to be sent, calculated from the placing in the output queue until it has been transmitted over the output-wire, is:

$$sendingDelay(s,u,t) = d(s,u,t) \cdot loadOut(s,u,t) + timeToBufferEmpty(s,u,t) \ ,$$

- where the function *timeToBufferEmpty*(*s,u,t*) at time t returns the time until the transmission buffer at the output queue for the wire (*s,u*) is empty.

Thus, it is stated that a package that is to be sent is moved to a transmission buffer at the time the transmission begins. The length of the queue is decremented at the same time the transmission begins but q must wait the time it takes to send all packages in front of it in the queue *plus* the time to complete the transmission.

Note that the queue-size includes the package q. Therefore when calculating the sending delay for a package the time to send the package across the wire is included in this way.

With the above calculations done, the time when the package arrives at u is:

$$arrivedAtInputQueue(s,u,t) = processedByRouter(s,t) + sendingDelay(\ processedByRouter(s,t)\ ),$$

(note that the formal definition of the *arrivedAtInputQueue*(…)-function [equation (10)] is equivalent – the *sendingDelay*(…)-function has just not been made explicit).

When arrived at u, the package q must possibly wait in the input queue – if, of course, there are packages in the input queue.

A similar situation might exist where the package currently being processed by the switching fabric was started some time before the arrival of the new package. In our router model, there exists no processing buffers at the switching fabric, thus a package remains in the input queue until the switching fabric is done with it. The time until the switching fabric has processed the newly arrived package is then given by:

$$inputQueueDelay(u,t) = speed(u) \cdot (loadIn(u,t) - 1) - timeSinceLastRoutingStarted(u,t).$$

Again we must assume that the function $timeSinceLastRoutingStarted(u,t)$ exists, that returns the time since the switching fabric last started work upon a package.

From this follows that the $inputQueueDelay(u,t)$ - function will return 0 if the input queue only contains the newly arrived package. If a single package is in the input queue and it started being processed at t_s time-ticks ago, the input queue-delay will be: $speed(u) \cdot (2 - 1) - t_s = speed(u) - t_s$, which will get closer to 0 – as t_s gets closer to $speed(u)$. For each extra package in the input queue, an extra $speed(u)$ will be added.

Note that the subtraction of 1 from the input queue-length above is, because we do *not* want to include the switching delay for q. This is contrary to the calculation for *sendingDelay* – where the wire delay can be included. When summing up a number of routing steps, it is necessary to include the switching delay of a package only once. This is done by $processedByRouter(s,t_0)$.

We can now calculate the time it takes to route the package q, from the *start* of the switching fabric in the source router s until the time right *before* switching at destination router u, by the following calculation:

$$\text{Routing time for package } q \text{ from } s \text{ to } u \text{ at time } t_0 = routingStep(s, u, t_0) =$$

$$processedByRouter(s, t_0) + sendingDelay(s, u\ processedByRouter(s, t_0)) +$$

$$inputQueueDelay(s, u, processedByRouter(s, t_0) + sendingDelay(s, u, processedByRouter(s, t_0))) - t_0 ,$$

- which, using [equations (9), (10) and (11)] can be expressed more concisely by:

$$readyForRouting(u, arrivedAtInputQueue(s, u, processedByRouter(s, t_0))) - t_0 ,$$

- which is exactly the result that equation [equation (8)] states. Note that it is necessary to subtract t_0 again, as we are only interested in *how long* it took to route from s to u when the package starts at time t_0. As opposed to being interested in *when* package q arrives at the switching fabric in u (which is what *readyForRouting*(*...*) in the last equation above tells us).

If the router u was in fact the destination router (and not only an intermediate router in the path of q) it is necessary to add the switching time of u as is done in equation [equation (7)], as we have defined a routing to stop right *after* the switching at the destination router. The reasoning is simply that the destination router has to spend the time to notice that this package is in fact destined for it - and decide what to do with it.

With this example, we complete the formal definition of the network – and conclude that it is possible to calculate the routing time for a package. In the following sections, we will further investigate the properties of the network model defined.

Routing quality in the model

The model described above contains at an abstract level the ability to route packages. In package routing loss and throughput are generally used to evaluate the quality of routing performed in the network over a given amount of time. In the following we will discuss both of these parameters in the context of our network model.

Package loss

Package loss happens in the situation where a package could not reach its intended destination. This happens if a package arrives at a queue, which is filled to capacity. A router has two types of queues, the input queue and the output queues. It is obvious that the input queue can be filled, thus resulting in package loss, but one

could argue that if an output queue is full, the router should wait until the queue could accept a package. This situation is called *head of line blocking*, being the situation where the first element in an input queue, after being through the switching fabric cannot be inserted into an output queue. This means that in a more accurate model, either the package should be moved to an extra buffer or the router simply waits until the needed output queue is available.

To avoid head of line blocking, we assume that packages are extracted from the input queue, processed by the switching fabric and inserted into an output queue. The package is lost if the insertion of the package exceeds the capacity of the output queue.

Another source of package loss is bad routing. Potentially a package can be routed indefinitely by a cycle in the routing tables, resulting in the package never reaching its destination, and increased load on the network. A method often used to avoid this problem is a Time To Live (TTL) counter, which is decremented every time a package arrives at a router. If the counter reaches zero, the package is discarded and assumed lost. Packages in the network model also contain a TTL counter to avoid the problem of indefinite routing. The initial value of the TTL counter differs from network to network, but in general it should be based on the amount of routers that packages in general are assumed to visit.

The job of a good routing algorithm is to reduce the amount of lost packages resulting from queues being filled and avoid that TTL counters reach zero.

Throughput

Another often used parameter for routing quality is the throughput in the network. Throughput describes the amount of data that can be transferred in the network as a whole. A network has a high throughput if packages are routed fast through the network and few of the packages are lost. The time spent being routed is determined by the speed of the used routers and the load on the routers.

To summarize:

The goal of a routing algorithm is therefore to maximize throughput by reducing the average amount of time it takes to route a package in the network while avoiding package loss resulting from overloading single routers.

Networks and dynamic graphs

It should be evident from the above definition(s) and discussions that a network represents a dynamic problem domain as the lengths of the paths fluctuate over time. We feel it important to stress the subtleties of the dynamics in the network since they differ somewhat from the dynamics found, had we simply used a dynamic graph with edge weights being stochastic functions only of time.

The relevant observation to make is that the conditions for movement are different. In a dynamic graph as defined in the previous paragraph, the ants must also react to changes in the domain in which they work. However, the work they perform does not affect the underlying domain as it only changes as a function of time.

In a network, this is not the case. One could state that delays change deterministically. Whenever a path between two routers is constructed, and packages are sent over this path, the length of this path changes as a function of how many packages are sent to routers on this path.

The situation becomes even more complex by the fact that a network is highly distributed consisting of a number of routers working concurrently, which in collaboration must find the best way to route all packages sent. Each time a router sends a package to a given target router, this target router's input queue load is incremented, which in essence effects all other routers in the network since their path length to that router has been increased by the increase in input queue length on the before mentioned target router.

The described 'feedback mechanism', does not exist in the dynamic graph mentioned above, where limitations in capacity and delays do not exist.

This essentially sums up the main reason, why we decided to design (and implement) a fully working simulated network. We could have slightly extended our initial implementations, based on our definitions of a graph [*“Definition of a graph”*, p. 5] to vary the edge-costs dynamically, thereby creating a very limited simulation of a network. An implementation of this sort would not have had the feedback mechanism described above, and furthermore this would not have made us able to define the quality parameters as easily.

Summary

In the above we have given a formal definition of our network model represented as a graph. Our network model models the network layer in the OSI reference model. We have shown how packages are routed through the network and defined the start and end-point of a routing. We have defined the length of a path in the network to equal the amount of time used to route a package in the model. We have discussed two commonly used quality parameters for routing and described their meaning in the context of our network model. Finally, we have described the subtleties in the dynamics of a network, which made us choose to design this network model.

Event driven network simulation

To facilitate an empirical study of the behavior of our ant inspired routing algorithms in the proposed network model, it was necessary to simulate the movement of packages over time in the model. To this end we have designed and implemented an event driven simulation scheme. The simulation is an adaptation of the event driven simulation scheme proposed in [Weiss 1998], where a priority-queue ensures that events are executed in order.

We will in the following presuppose a basic knowledge of event driven simulation, and will not go into detail about the implementation. For a short introduction to the topic of event driven simulation see [Weiss 1998, chapter 13.2].

The concept of event location

An event-driven simulation only needs to contain a number of events with a specified time, and an event-holder, which handles the scheduling of these events.

We have a system, where the defined events are spawned by other events. This is simply because a package travels in a router-wire-router-wire-… cycle. The complexity in the system lies in knowing when a package is ready for a switching operation or a sending operation (at a router or wire respectively), because of the buffering in the input- and output queues.

Furthermore, in the formal case we had to define specific functions that handles the special cases, where another package is already being processed (*timeSinceLastRoutingStarted*() and *timeToBufferEmpty*()). When implementing this in the event-driven simulation, we came upon an idea that could simplify this slightly.

The basic idea is to keep track of when the *next event* is allowed to happen on a router or wire. The router or wire simply holds a time value that denotes the *next* time a switching/sending is not in progress. This means that an event can easily calculate the time it can schedule a next event. This will always be:

$$\begin{cases} nextEventTime(router/wire) + durationOfThisEvent & \text{, if } now() < nextEventTime(router/wire) \\ now() + durationOfThisEvent & \text{, if } now() \geq nextEventTime(router/wire) \end{cases}$$

If the *nextEventTime* is sometime in the future, we must schedule the *next* event by the time-value in *nextEventTime*. Else, the router/wire is ready and the event can be scheduled by the present time: *now*(). The *durationOfThisEvent* denotes the duration of the current event, which of course must finish before the next event in this location can begin.

It is now required of an event, that it updates the *nextEventTime* of the router or wire. The update-value will generally be the same time that the new event was scheduled for; this is exactly the time that the router has finished switching or the wire is finished sending (a small complexity is induced by the existence of the transmission buffer, though – see Routing Event below).

We have chosen to name this concept *event location*, as it refers to the fact that we utilize that events happen on locations in this simulation.

Simulation events

We will now define the events that can occur in this event-driven simulation. The goal here is to simulate routing in our network model, a task requiring several steps that has already been discussed in chapter [*Routing*, p.25]. We will briefly outline the events we have defined, their duration and purpose. The events and their properties should follow naturally from the description of routing in the network model and the discussion of package loss in [*Routing quality in the* model, p. 29].

Package creation is initiated by an *input queue event.* This should be interpreted as if an external entity is responsible for creating the package and has sent it to the initial router in our model. This mean that all new packages must wait in the input queues, and they risk being lost if the queue capacity is exceeded.

Input Queue Event

This event occurs whenever a package arrives at an input queue on a router. The package is lost if the input queue capacity is exceeded when it is attempted to insert the package in the input queue.

If the queue is not full, an input queue event creates a *routing event*. This event is scheduled to occur when the package has waited in the input queue and has been processed by the switching fabric. The time at which the *routing event* occurs is given by:

$RoutingCompleteEventTime(r_i,t) = nextEventTime(r_i,t) + speed(r_i)$, if the input queue is *not* empty

$RoutingCompleteEventTime(r_i,t) = now() + speed(r_i)$, if the input queue is empty

(- where r_i is the router where the event is located in the network)

Routing Event

This event occurs when a package has been processed in the switching fabric of a router. The length of the input queue on the router is decremented – note that this happens implicitly because the queues are only simulated with the help of the *nextEventTime*.

If the package has not arrived at its destination, it calculates the next router for the package by using a routing algorithm. If the selected output queue is not full, its length is incremented (implicitly by way of *nextEventTime*). A *start-sending event* is scheduled to occur, when the package has waited in the output queue. The new event time is calculated by:

$StartSendingEventTime(r_i,r_j,t) = nextEventTime(r_i,r_j,t)$, if the output queue is *not* empty

$StartSendingEventTime(r_i,r_j,t) = now()$, if the output queue is empty

(- where (ri,rj) is the wire connecting the event location router r_i and the destination router r_j in the network)

Note that we do *not* include the time to be sent over the wire in this event. This is because we have elected to include a transmission buffer between the output queue and the wire. As the internal transfer speeds are zero, it should be possible for a package to be moved directly to the transmission buffer, if the output queue is empty.

Start Sending Event

This event happens when a package is in the transmission buffer, and ready to be transferred over a wire. Implicitly it decrements the length of the output queue in which the package was located to indicate the move to the transmission buffer. It then creates an *input queue event* to indicate that the package arrives at the destination after being sent over the wire connecting the current router with the selected target. The calculation of the time for the new *input queue event* is simply:

$InputQueueEventTime = t + d(r_i, r_j)$

(- where r_i is the router where the events happen, and r_j is the next router selected by the routing calculation. t is the time when the start sending event happens).

Ant packages in the simulated network

It should be clear from the above that the specified events occur in a circular fashion with one creating the next until the package arrives at its destination. Note that this is similar to the movement of ants in the ACO metaheuristic[10]. Here the ants move from vertex to vertex until they have reached their success criteria or have died. In our ant-based routing algorithm ants move as packages (aptly named ant-packages) and perform stochastic state transitions and lay trail as ants in the ACO metaheuristic. The main difference between the network simulation and the ACO metaheuristic is that ants move concurrently instead on sequentially.

[10] In the procedure *new_active_ant*() in [Algorithm 1, p. 18].

Summary

In the above we have defined the events in our event driven simulation. We have indicated how a package is processed by a series of events, which move the package between routers and ensure that consistency in the state of the routers by (implicitly) updating the queues. We have also indicated that the use of events simulates the ACO metaheuristic with the main difference that ants in the simulated network move concurrently.

Ant Routing System - an ACO network routing algorithm

In this chapter, we present our routing algorithm Ant Routing System (ARS) that is based on the ACO metaheuristic. ARS is inspired by Ant System (as presented in [*Ant System*, p. 20] and another ant routing algorithm - AntNet. AntNet is a routing algorithm proposed by Di Caro and Dorigo based on the ACO metaheuristic, which have performed very successfully in simulations. It is beyond the scope of this report to introduce to all the methodologies and ideas proposed in AntNet - for more information, see [Bonabeau et al. 1999] or [Di Caro & Dorigo 1998]. We will start by stating the problems we aim to solve, and the basic idea behind the algorithm. We will explain which heuristics have been used and how they contribute in solving the routing problem.

Motivation

A major problem in routing today is congestion. Routers risk sending packages through bottlenecks, thereby contributing to congestion, and increasing the probability of package death. If only routers had significantly different routing tables, a natural distribution could be expected, but unfortunately routers normally share information regarding which paths are the shortest. A router also tends to route all packages to a given destination through the same wire.

If routers could somehow individualize their choices, and also utilize a stochastic distribution of packages via different wires, then we believe that some of the bottleneck phenomena could be avoided. This will of course open up for a completely new discussion regarding unwanted side-effects and increased traffic on the network as a whole, but that is an entirely new problem, which we do not have the resources to discuss here. Suffice to say that in this project report we will concentrate on developing some basic route-finding and distributive capabilities for some limited settings.

In the following we will give a description of what we believe is a productive way of routing packages in a network with limited capacity. Limited capacity in a network means that sending a stream of packages between routers (that are not neighbors) requires more than one distinct path in order to ensure package arrival because of the limited capacity of a single path. This necessitates distribution of package transfers over several paths.

We believe that the ideal way to organize this distribution would be to sort the paths after lengths, shortest first, and then start utilizing the ordering by selecting the shortest paths first and then use them to their full capacity before utilizing the next in order.

In a dynamic domain such as a network, the ordering could change when traffic is directed over a specific path. Consequentially it is necessary to continuously reevaluate and adapt to the changes in latency over the different paths. The strategy we have tried to implement in Ant Routing System is thus to utilize the shortest paths to their full capacity, compensating for the fact that using them can cause them to cease to be the shortest because of the increased latency created by increasing queue lengths.

This can be done by converging to a number of good solutions, while avoiding stagnation by the use of exploration.

An overview of Ant Routing System

Ant Routing System (ARS) implements the above mentioned strategy by using the following mechanisms: local heuristic value rule, transition probability rule, random step, positive feedback and negative feedback. We will give an informal presentation of their use in ARS, and describe how they participate in solving the routing problem.

The local heuristic value used in ARS works by estimating the time it will take to transfer a package to an adjacent router, by utilizing knowledge about the speed of the wire and the load on the output queue connected to the wire. The estimate is used to decrease the probability of selecting wires with high use, thus reducing the risk of package loss. This also has the effect of promoting exploration, since the wires that are highly used are, other things being equal, also the wires with the highest trail value and the longest queues.

Positive feedback is, as always in ACO algorithms, used to ensure that better paths are given a higher trail value and thus a higher probability of being used. The quality of a path is inversely proportional to the time a package has spent being routed from the paths source to its destination. Positive feedback is in ARS implemented in the way that all the routers on the used path are rewarded, by an increase in trail value according to the quality of the used path.

The stochastic state transition rule in ARS uses local heuristics and trail value to create paths. We use two parameters α and β to determine the relative influence of local heuristic and trail value.

Uniformly random steps are used in ARS to counteract stagnation. At a given router, there exists a small probability, given as an argument to the algorithm that the package is routed completely at random (but not to any router it has already visited).

Negative feedback in ARS is implemented by decrementing trail values where packages are lost. This leads traffic away from bad paths, and thus lessens the risk of package loss. This mechanism counteracts convergence and subsidiary promotes exploration since it lowers the probability of selecting a used wire thus increasing the probability of selecting the other wires leading away from the router.

All these mechanisms participate in promoting that packages are routed over good paths and in reducing the risk of them becoming lost. Positive feedback continuously enforces the trail values in the routing table to promote the use of the best known paths. Negative feedback are used to counteract the possibility that positive feedback makes the system stagnate, and also makes it possible for the system to react if a good path suddenly becomes bad or overloaded. Negative feedback caused by package loss and long output queues are also used to promote exploration of new paths.

A number of parameters determine the importance of the different mechanisms in ARS and they are presented in section [*Parameters of the algorithm*, p. 38], and put into context in the following chapters.

Before we start a more formal presentation of the functions used in ARS we will present a number of requirements that ARS makes to the network in which it works.

Requirements of the network

We have modeled and implemented a simulated network as explained in the chapter [*Network Model*, p. 25]. This simulated network is the domain in which the implemented routing algorithm – presented in this chapter – works.

By modeling a network and letting the algorithm work with representations of objects in a physical network, we were forced to think in terms of what is possible in a "real" network. The algorithm that we have implemented requires a few functionalities in the network protocol that are not commonly in use.

- *Routing tables:* To support the stochastic path-selection performed in ACO-algorithms, it is necessary to increase the amount of information stored in a routing table. The general problem of routing from any router $s \in R$ to any router $d \in R/\{s\}$ in a network $N = (R,W)$ (as defined in [*Formal Definition*, p. 25] can be seen as $|R|$ problem instances in the same network.
  In deterministic routing algorithms, it suffices to point to the next router on the path to d – which effectively translates to a routing table containing $|R|-1$ entries (no need to route to oneself).
  In stochastic routing, it is necessary to support the path selection algorithm by storing information in the routing table regarding the quality of selecting each outgoing wire. The amount of space required in a routing table increase, as the network-graph becomes increasingly dense. This means that in a fully connected network the routing table at each router will contain $(|R|-1)\times(|R|-1)$ entries.

  The routing tables implemented for our algorithm are organized in this way and contain entries that hold a trail-value that is used in the stochastic selection process. A more formal definition is given in section [*Description of the routing table*, p. 37].
- *Ants are packages:* All the packages, which are used to route in ARS, are ants. As every ant maintains a memory that is used as a taboo-list, and for retracing the path to update trail-values, this memory has to be stored in the package – either as payload or as a designated header-field.

- *Instant router update:* We use a delayed update of trail-values, so information regarding the quality of an ant's path should somehow be propagated back through the network to the routers on the used path. In our implementation, this propagation is not effectuated, only simulated. Instead, all routers are instantly informed of the quality of the path by the metaheuristic[11].

Mechanisms in the algorithm

In the following sections, we will specify and define the parts and mechanisms of the Ant Routing System – routing algorithm.

Description of the routing table

As noted above, special requirements to the routing table exist. In order to ensure consensus about the definition of the routing table we will give a formal definition of the routing table used in the algorithm.

The trail value $\tau^{d}_{r_i r_j}(t)$ denotes the trail value at time *t* for the wire ($r_{i,}, r_j$) that connects to the destination router *d*. A routing table in a network *N = (R, W)* for a given router $r_i \in R$ connected to a set of *m* routers $\left\{ r_j \in R \,\middle|\, (r_i, r_j) \in W \right\}$ contains $(|R| - 1) \times m$ trail-values. A routing table in r_i thus contains for every destination $d \in (R/\{r_i\})$ router a set of trail-values:

$$\forall d \in (R/\{r_i\}) : \left\{ \tau^{d}_{r_i r_j}(t) \middle| (r_i, r_j) \in W \right\} \tag{12}$$

Each set of trail values at r_i for a given destination *d* is in ARS initialized so that each of the trail-values are equal and so that they sum to 1. This is an invariant of the algorithm.

Local Heuristic Value

As a local heuristic value we use a run-time calculated estimate of the time $\eta_{r_i r_j}(t)$ of using wire $w_k = (r_i, r_j)$ at router r_i at time *t*. It is calculated by the function:

$$\eta_{r_i r_j}(t) = \eta_{w_k}(t) = d(r_i, r_j) \cdot loadOut(r_i, r_j, t) + d(r_i, r_j) \,, \tag{13}$$

This is an estimate of the time required to move through the output queue of r_i and over the wire w_k. The routing algorithm can easily access this information in the router. The reason why this is only an estimate is that by multiplying the wire delay with the output queue length, it is assumed that the transmission buffer is empty. Hence, this estimate becomes worse as the wire delay of w_k increases. This is in essence a pure greedy heuristic; no notion is made of where the package is destined.

[11] In AntNet, this happens by the spawning of a so-called back-ant, when the routing of a package is finished. This is a high priority ant-package that retraces the path selected by the 'original' ant-package (in AntNet called a forward-ant), and makes the necessary updates. This should optimally happen as quickly as possible, which is why the back-ant should be a high-priority package. This requires that the network is able to route based on the type of service wanted.

Parameters of the algorithm

The parameters of ARS are as follows:

- Two parameters α and β are used to scale the relative importance of trail value $\tau^{d}_{r_i r_j}(t)$ and the heuristic value $\eta_{r_i r_j}(t)$ [12]. By setting any of these values to 0, trail following behavior or the use of local heuristics can be 'turned off'.
- We use a parameter s_τ to scale the amount of trail deposited. This value should be set to no more than the minimal route between any routers in the network. Generally, this value is set equal to the delay of the 'shortest' wire: $s_\tau = \min\{d(w_i) : w_i \in W\}$.
  By setting this value to zero no trail is deposited upon completion of an ant-package tour.
- Further, a parameter p_r, where $0 \le p_r \le 1$, defines the probability of a package being routed randomly at any given router in the network. This is implemented as a restricted random step as defined in [*Restricted random walk*, p. 6]. The random move possibility can be turned off, by setting this value to 0.
- A parameter b_τ is used to for setting a lower and upper bound on trail-values in the routing tables. The range of possible value for any trail value $\tau^{d}_{r_i r_j}(t)$ is $[b_\tau ; 1 - b_\tau]$. The value of b_τ must be in the range $]0;½]$. This is necessary to preserve the invariant for trail-values (see the comments concerning bounds on trail in [*Discussion of implemented heuristics,* p. 40]). Furthermore, if the bound collides with the invariant for the trail-values, the bound is violated. E.g. if the user decides to set $b_\tau = ½$ and one router has three outgoing wires, it is then impossible to respect the bound, because the invariant – that the trail-values sum to 1 – must be respected.
- Furthermore, we use a parameter n_τ to scale the amount of trail *removed* from a routing table entry $\tau^{d}_{r_i r_j}(t)$ on router r_i, when an ant-package is lost on the *next* router r_j in its path. This is an immediate negative feedback on the quality of choosing r_j, when routing to destination d. This reinforces the effect of the heuristic value when the state of the network is such that packages are being dropped, but furthermore (together with the bound on trail values above) has the possibility of avoiding premature convergence of one of the trail values. This value must be in the range $[0;1]$. If this value is 0, the negative feedback is turned off.

Transition probability rule

The rule for calculating transition probabilities in ARS is somewhat similar to the rule in AS. The probability at time t on router r_i of an ant-package k destined for router d of selecting router r_j is given by:

[12] This is similar to the α- and β-parameters used in Ant System [*Ant System*, p. 20].

(14)
$$p_{r_i r_j}^{k}(t)=\begin{cases}\dfrac{1}{\sum\limits_{r_l \in J_{r_i}^{T_k}} 1} & \text{,if } \chi \le p_r \\ \dfrac{\left[\tau_{r_i r_j}^{d}(t)/b_\tau\right]^\alpha \cdot \left[\eta_{r_i r_l}(t)\right]^\beta}{\sum_{r_l \in J_{r_i}^{T_k}}\left[\tau_{r_i r_j}^{d}(t)/b_\tau\right]^\alpha \cdot \left[\eta_{r_i r_l}(t)\right]^\beta} & \text{,if } \chi > p_r\end{cases},$$

- where $J_{r_i}^{T_k}$ is the neighborhood N_{r_i} restricted by ant-package k's path-list T_k as defined in [*Restricted random walk*, p.6], and χ is random uniformly distributed variable in the range [0;1]. As in AS the probability $p_{r_i r_j}^{k}(t)$ is 0 if router r_j is in T_k.

Trail deposition and pheromone trail update

The rules used for trail update are split into a set of rules related to the delayed positive deposition on *all* routers on the path of an ant-package; and a set of rules used to update the trail on a *single* router, when an ant-package is lost.

Positive feedback

Deposition and updating of artificial pheromone trail happens immediately upon the completion of an ant-package k's run. For all $r_i \in T^k$ the set of trail-values $\left\{\tau_{r_i r_j}^{d}(t) \middle| (r_i, r_j) \in W\right\}$ are updated by the following formulas:

An update value is initially calculated by:

(15)
$$\tau_{upd} = \frac{1}{length(p,t)} \cdot s_\tau,$$

- where s_τ is the trail scale parameter presented above, p is the path generated from T^k, and *length*(p,t) is the time spent in the network by the ant-package as defined in [*Formal Definition*, p. 25]. Generally the trail deposition rules below require the value τ_{upd} to be in the range $[-1;1]$ (to allow for negative updates – see below).

Note that the reason why s_τ must not be larger than the minimal route between any routers in the network is based on this requirement of τ_{upd}. If so, the τ_{upd} would break the $[-1;1]$ boundary. The calculated τ_{upd} is used in the calculation of the new trail values, upon every router r_i:

If $r_j \in T_k$:

(16)
$$\tau_{unbound} \leftarrow \begin{cases}\tau_{r_i r_j}^{d}(t) + \tau_{upd} \cdot \tau_{r_i r_j}^{d}(t) & \text{,if } \tau_{r_i r_j}^{d}(t) \le 0.5 \\ \tau_{r_i r_j}^{d}(t) + \tau_{upd} \cdot \left(1 - \tau_{r_i r_j}^{d}(t)\right) & \text{,if } \tau_{r_i r_j}^{d}(t) > 0.5\end{cases},$$

- and if the calculated value violates the bounds $[b_\tau; 1 - b_\tau]$:

(17)
$$\tau_{r_i r_j}^{d}(t) \leftarrow \begin{cases} b_\tau & ,\text{if } \tau_{new} < b_\tau \\ 1-b_\tau & ,\text{if } \tau_{new} > 1-b_\tau \\ \tau_{new} & ,\text{else} \end{cases}$$

- and for the remaining values (if $r_j \notin T_k$), where τ_{new} denotes the trail value calculated for the chosen path (by the formulas given above):

(18)
$$\tau_{r_i r_j}^{d}(t) \leftarrow \tau_{r_i r_j}^{d}(t) - \left(\tau_{new} - \tau_{r_i r_j}^{d}(t)\right) \cdot \left(\frac{\tau_{r_i r_j}^{d}(t)}{1-\tau_{new}}\right)$$

Negative feedback

When an ant-package k is lost the *last* router r_i on its path will penalize the trail-value of the router r_j it selected, i.e. the router where the package was lost.

The only changes to the rules for trail-deposition above are the calculation of the update value τ_{upd}. It is now simply:

(19)
$$\tau_{upd} = n_\tau$$
, where n_r is a parameter.

The formulas [Equations (16) and (17)] and [Equation (18)] are used unchanged as above to update the routing table entries for each $\tau_{r_i r_j}^{d}(t)$ in r_i for every d.

One of the success-parameters of the design of formulas [Equations (16) and (17)] and [Equation (18)] was to make them maintain the invariant for trail-values for both positive and negative update-values τ_{upd}.

Discussion of implemented heuristics

The trail update rule we have constructed rewards and penalizes trail values proportionally *less* when they approach the upper and lower bound of the trail values. This is useful, as it in itself works against premature convergence, but we found it useful to experiment with further ways of immediately responding to local peak load states. The last three parameters (p_r, b_τ, and n_τ) in the list in [*Parameters of the algorithm,* p. 38] are representatives of three possible solutions we formulated for this.

The (small) probability of a random transition was our first idea. In our tests, the probability p_r is typically set to a value in the range $[0.00;0.02]$. It is a simple well-known approach in stochastic optimization to always allow for some exploration. However the arguments against using this heuristic are that this is a somewhat course-grained method of avoiding convergence. Furthermore, if an implementation based on every package being an ant-package were to be made, this heuristic would seriously compromise the possible quality-of-routing.

The bounds-on-trail heuristic was invented together with the trail deposition rule. The rule requires some small bound on trail-values larger than zero. The deposition rule makes trail values that have converged to a value very close to 0 or 1, change *very* slowly. This is because the quantity of the update is proportional to the numerical closeness of the former trail-value to 0 and 1, respectively.

Hence, if a trail value ever approach the bounds: $\tau_{r_i r_j}^{d}(t) \approx 0 \vee \tau_{r_i r_j}^{d}(t) \approx 1$, this value has effectively converged to a non-changeable value. In a dynamic problem instance, we find this unfortunate and harmful as it effectively 'freezes' a route that has 'just' proven to be good over some time. In our implementation the bounds is implemented as a simple cut-off. If a trail-value violates the bounds it is set to the value of the bound it violated. In our tests we have typically used $b_\tau = 0.1$, giving a range of possible trail values: $[0.1;0.9]$.

We had a number of discussions how to design negative feedback that was to be applied to the system as the consequence to the death of a package. We wanted to implement a localized and more controllable enforced exploration than was possible with the random-transition heuristic.

When a package *p* is dropped, we require the router r_i that last handled *p* to (somehow) time-out at time $t_{timeout}$ and notice, that the package was dropped by the router r_j that *p* was routed to. This of course requires some form of acknowledgement of packages at the network layer; but given this, router r_i immediately performs a trail-deposition with a negative value equal to the parameter n_τ upon the trail value $\tau^d_{r_i r_j}(t_{timeout})$. We discussed propagating a proportionately less negative update further back through the path of *p*, but this would further require that a copy of (the path-list of) *p* has been temporarily held at r_i. As it is, we have typically tested ARS with $n_\tau = 0.1$ and set $t_{timeout} = 0$ (as we can detect instantly whether a package is lost) to simplify the simulation.

Summary

We have presented a routing algorithm Ant Routing System (ARS), which we have designed, implemented and tested during this project. We have discussed the background for implementing an ant-inspired routing algorithm, and described the properties and mechanisms of ARS in overview and detail. Finally, we have discussed the purpose and the motive behind each of the heuristics implemented in ARS.

In the next chapter, we present a number of tests, which are designed to illustrate the properties and give hints as to the performance of ARS.

Description of selected empirical observations

The time schedule of this project have not made it possible to systematically test and optimize the parameters of the algorithm, but we will in the following report the results and tendencies we have found while developing and testing the routing algorithm on a few limited simulated networks. We discuss the results and propose ideas for further tests.

We have made some restrictive simplifications about the network in our experiments:

- The packages in the network are created at a constant interval. There are no peak load situations with a sudden increase of traffic, where more packages are created in the network as a whole.
- There is always only one router sending and only one router receiving packages. There is no other traffic on the network.

The results of our tests should be seen as a demonstration of the procedures we could follow in a more thorough and systematical tests. In experiment 1 we will briefly demonstrate the effects of varying some of the basic parameters. In experiments 2 and 3 we will use the set of fittings for the algorithm that are defined below. These fittings have been found to be effective through "trial-and-error" methods during our studies:

- α: 3 – the relative importance of trail-values.
- β: 1 – the relative importance of local heuristic-values.
- p_r: 0.02 – the probability of a package being routed only using a restricted random step (as defined in [*Restricted random walk*, p. 6].
- s_τ: 10 – the scale of the trail deposited, this value must be smaller than or equal to the minimum wire delay in the network.
- n_τ: 0.3 – the trail removed from a routing table entry on router, when an ant-package is lost on the next router in the path of the package's.
- b_τ: 0.1 – the parameter used to for setting a lower and upper bound on trail-values in the routing tables. This means that the allowed range for trail-values is [0.1;0.9]

We will measure the quality of routing in the network by two parameters: package death and the average distance traveled by all packages. These two parameters (together with a number of relevant data concerning the simulation) are collected or calculated during a simulation and stored as output of the simulation. Furthermore the amount of dead packages are measured at fixed intervals and stored in another output-file.

For every run of a simulation we furthermore generate a picture of the network with a graphical representation of the number of ant-packages that has traversed each wire.

The network definition file and the simulation setup file are copied to the aforementioned directory. The network definition file is the file that defines the topology of the network (number of routers and wires, and the attributes of these).

The simulation setup file defines the routing algorithm to use, the parameters that the algorithm will use, the duration, if and when to collect statistical data, the package creation interval, which routers create the packages, and which routers receive the packages.

For more information regarding these files see the appendix A [*Format of input files*]. We will use the information gathered in the output-files and pictures generated as illustrations in the following sections.

Comparison with a "benchmark" algorithm

We have developed an implementation of a routing algorithm based on Dijkstra's Single-Source Shortest Paths algorithm as comparison when we examine our routing algorithm in network topologies where it is not possible to route all the packages by the same path without loosing packages.

The routing algorithm is run once in the beginning of the simulations, thus for each router creating a routing table containing a shortest path to each destination router. The algorithm is allowed to update the routing tables of the routers at every *T* time step, where *T* is an argument given to the algorithm. This means that the algorithm at each *T* time step has an updated global snapshot of the network.

The shortest path is not calculated on wire-delays only. The estimated delay denoted γ_{ij} of using wire $w_{ij} = (r_i, r_j)$ at time *t* is calculated by the following formula if $capOut(r_i, r_j) < loadOut(r_i, r_j, t)$:

$$\gamma_{ij}(t) = d(r_i, r_j) \cdot loadOut(r_i, r_j, t) + d(r_i, r_j) + speed(r_i) \cdot loadIn(r_j, t) \quad (20)$$

If $capOut(r_i, r_j) = loadOut(r_i, r_j, t)$ then $\gamma_{ij}(t) \approx \infty$ (infinity is just implemented by an arbitrary high value).

So the estimated wire delay of w_{ij} is calculated as the sum of the time to move through the output queue of the source router; over the wire, and through the input queue of the destination router. If the output queue of the source router is already full, the infinite value will ensure that this wire will not be chosen.

The above-mentioned routing algorithm is not perfect. A perfect routing algorithm would be able to take into account the states of the network at all times, both past, present, and future, and would therefore be able to foresee congestion, and thereby increase distribution before the congestion occurred.

As an example take the situation where *x* routers all send a package to arrive at a router *d* at time *t* and *d* has an empty input queue at time *t*. If the capacity of the input queue at *d* is less than *x*, packages will be lost. Our benchmark algorithm cannot foresee this situation.

Furthermore it has other shortcomings that we will discuss in more detail in the experiments where they are observed.

Network topologies

The experiments presented in this chapter have been made on two special network topologies: double bridge and 10x10 grid. The double bridge network is similar to the double bridge experiment presented in [*Double bridge experiment*, p. 11]. We have chosen this network topology to test some basic properties of ARS. The network topology represents the simplest topology where both distribution and convergence to a single path can be tested.

The 10x10 grid network topology was chosen as an example of a ‘large’ network in which it is easy to investigate and describe traffic, since it has a regular structure. We wanted a regular topology because of the ‘reduced’ complexity and simply because it is easier to get a general view of the paths in the network as can be seen from the pictures shown later in this chapter.

Double bridge

We tested the basic performance and abilities of ARS upon a network topology derived from the double bridge experiment. The network consists of four routers connected by four wires:

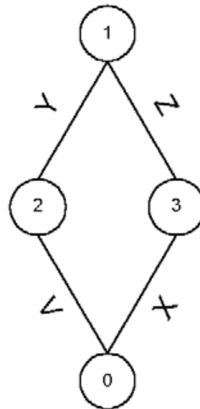

Figure 5 : The double bridge network

The router 0 is the sender of packages in the simulations with this network topology. The input queue of 0 is set to an arbitrarily high value so that – no matter the creation interval of packages – packages will not be rejected from the network. In effect, this is a buffer for recently created packages. This means that the package-switching delay of this router decide the rate at which packages flow into the rest of the network (this delay is set to 7).

The routers 2 and 3 have set input queue and output queue-sizes with space enough for 10 packages. The package-switching delay of these routers is set to 10.

The router 1 is the receiving router in simulations with this network topology. We have conducted experiments with an input queue length of 10 packages (as above), but the package-switching delay is only 1, so that the router does not drop packages from its input queue (as a package is removed from its input queue at every time step).

10x10 grid

We will furthermore perform experiments on a 10x10 grid network, which of course consists of 100 routers connected in a grid with wires (with a delay of 10) between adjacent routers. We used this topology to test ARS ability to route in a 'larger' (but still simple) network.

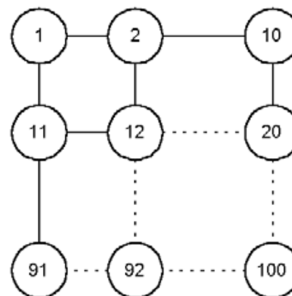

Figure 6 : The 10x10 grid network

The router 23 is the sender of packages in the simulations with this network topology. Router 23 is similar to router 0 in the double bridge network.

The router 85 is the receiving router in simulations with this network topology. Again this router is similar to the receiving router above.

The rest of the routers have set input queue and output queue-sizes with space enough for 10 packages. The package-switching delay of these routers is set to 15.

The reason why we have chosen these two routers is simply that they are internal routers in the network topology – making as many feasible paths as possible. The exact location of the routers was somewhat arbitrarily chosen.

This topology furthermore has the property that a number of paths are of equal initial quality. All paths in the rectangle spanned by router 23 and 85, which only consists of moves to a router with a higher index (moves to the "right" and "down"), will initially be of same length.

Definitions

The term *single path* below should denote that at least 95% of the packages sent from the router r_i reaches the router r_j through the same path. If it is possible to route by a single path without loosing packages we refer to the network load as *light.* Otherwise we refer to the network load as *heavy*.

Theses

Although we originally planed to present experiments in both networks with both light and heavy network load, we chose not to include experiments with light network load. The reason for this is that there is no problem in routing when all packages can be routed by the same path. Furthermore the experiments did not provide any observations that the experiments made with heavy network load did not provide.

To illustrate the performance and the abilities of ARS, we have constructed a set of simple theses that will form the basis for a set of tests.

1. First we want to demonstrate the basic abilities of ARS by varying the parameters controlling some of the primary mechanisms: positive feedback and negative feedback, and the balance between relying on local heuristics and relying on trail-values for calculating probabilities for movement. To show clearer and "clean" effects, we have elected to make experiments with exaggerated values; and furthermore the possibility p_r for random moves, have been set to zero.
   The following initial tests will be performed on the 10x10 grid network. The network load will be *heavy*. As in all the following tests it will be performed with a single source router and a single destination router.

   1.1. In this test we will prioritize trail above local heuristics by setting α high and β high (relatively). This experiment is performed without negative feedback ($n_\tau = 0$) to demonstrate the abilities of the algorithm to converge to a single path.
   We expect to find that the packages converge more or less to a *single path*, because of the prioritizing of trail above local heuristics, even though this path looses packages due to load. As the local heuristics are not prioritized the packages will not be led away from routers that are relatively full.
   As there is no negative feedback when packages die, and only a low priority on information regarding the 'filled'-status of routers, packages will probably be lost in relatively high degree due to filled queues. Only the effect caused by the fact that paths get slower due to filled queues will work against *single-path* behavior.

   1.2. In this test we will prioritize local heuristics above trail by setting α low and β low (relatively). This experiment is performed without negative feedback ($n_\tau = 0$) to demonstrate the abilities of the algorithm to distribute the packages.
   We expect to find that the packages are distributed over the entire network, because of the priority of local heuristics above trail. This will force the packages to be routed primarily to routers, which have a light load relative to the other router-possibilities.
   Packages will probably be lost in relatively lesser degree due to filled queues, but in a higher degree due to the fact that ant-packages maintain a taboo-list. Packages can therefore reach a 'dead end' caused by their own taboo-list. (We refer to this as a no-possible-routing-event).

   1.3. In this test we will prioritize trail above local heuristics by setting α high and β high (relatively). This experiment is performed with negative feedback ($n_\tau = 0.5$) to demonstrate the abilities of the algorithm to converge, but still be able to react on loss of a package.

We expect to find that the packages initially are routed via a few good paths. But when packages start to be lost along these paths, the router that lost the package will be circumvented for a time-period. There are several possible outcomes of this scenario, because trail is highly prioritized, but the most likely is an oscillation between a few paths.
Packages will be lost due to filled queues when 'shifting' between paths, but we expect lesser package death than with no negative feedback.

1.4. In this test we will prioritize local heuristics above trail by setting α low and β low (relatively). This experiment is performed with negative feedback ($n_\tau = 0.5$) to demonstrate the abilities of the algorithm to distribute the packages, but still be able to react on loss of a package.
This test has been included for completeness. As the high priority of local heuristics will ensure that routers exposed to a limited load will be assigned low probabilities. As the load on router-queues approach their full capacity, the probabilities will become even smaller. This means that the most likely cause of package loss will be that packages cannot be routed as they have reached a 'dead end' caused by their own taboo-list.

To further illustrate the capability of ARS in a simulated network, we present a set of experiments with parameters fitted for the networks. The general thesis is that ARS (with a set of fitted parameters) is able to optimize package throughput with a single source router and a single destination router -

2. – given that the network load is *heavy* and the network is setup as the *double bridge network*.

   2.1. All wires *V*, *X*, *Y*, and *Z* have equal delay.
   We expect to find that the system does not converge to a single path. This would be optimal, because the network load is heavy, so many packages are lost if only one path is used.

   2.2. The wires *V*, *Y*, and *Z* have equal delay and wire *X* have a delay that is several magnitudes larger.
   This experiment should show that the system converges to the path $p = \langle 0,2,1 \rangle$ over the wires *V*, *Y* and use this path fully and route the extra load over the other path.

   2.3. The wires *V*, *X*, and *Y* have equal delay and wire *Z* have a delay that is several magnitudes larger.
   The outcome of this experiment should be the same as above.

   2.4. The wires *V* and *Z* have equal delay and the wires *X* and *Y* have equal delay, which are larger.
   Because the network load is heavy, we expect to find that the system does not converge to a single path. The optimal solution would be to utilize both paths evenly.

3. – given that the network load is *heavy* and the network is setup as the *10x10 grid network*.

   3.1. A comparison between ARS and Dijkstra-'benchmark'-algorithm based on a number of run simulations. We use Dijkstra-'benchmark'-algorithm as the standard of reference, as the Dijkstra-'benchmark'-algorithm is a near optimal routing algorithm when allowed to update its routing tables with a low interval.
   We hope to find that ARS can minimize the package loss while still providing reasonable routing delays.

4. *Unexplored*: ARS is able to route with multiple source routers and multiple destination routers. While this would be the most realistic and perhaps also most interesting experiment to perform, we do not feel that we would be able to describe the experiment in sufficient detail, because the very large complexities related to this kind of network traffic. As mentioned, this test would have been too extensive to conduct systematically during the course of this project.

Results of experiment 1 – test of basic capabilities in a 10x10 grid network

To sum up the points in experiment[13] 1 we wanted to show the basic abilities of ARS by altering some of the parameters. In experiment 1.1 we prioritized trail above local heuristics, we therefore expected the path of the packages to converge more or less to a *single path* and that the packages would die on the queues of the routers. As can be seen on Figure 7 the packages mainly follows two paths (both short paths), so our conjecture was not completely wrong. As expected almost all of the dead packages died on the input queues of the routers.

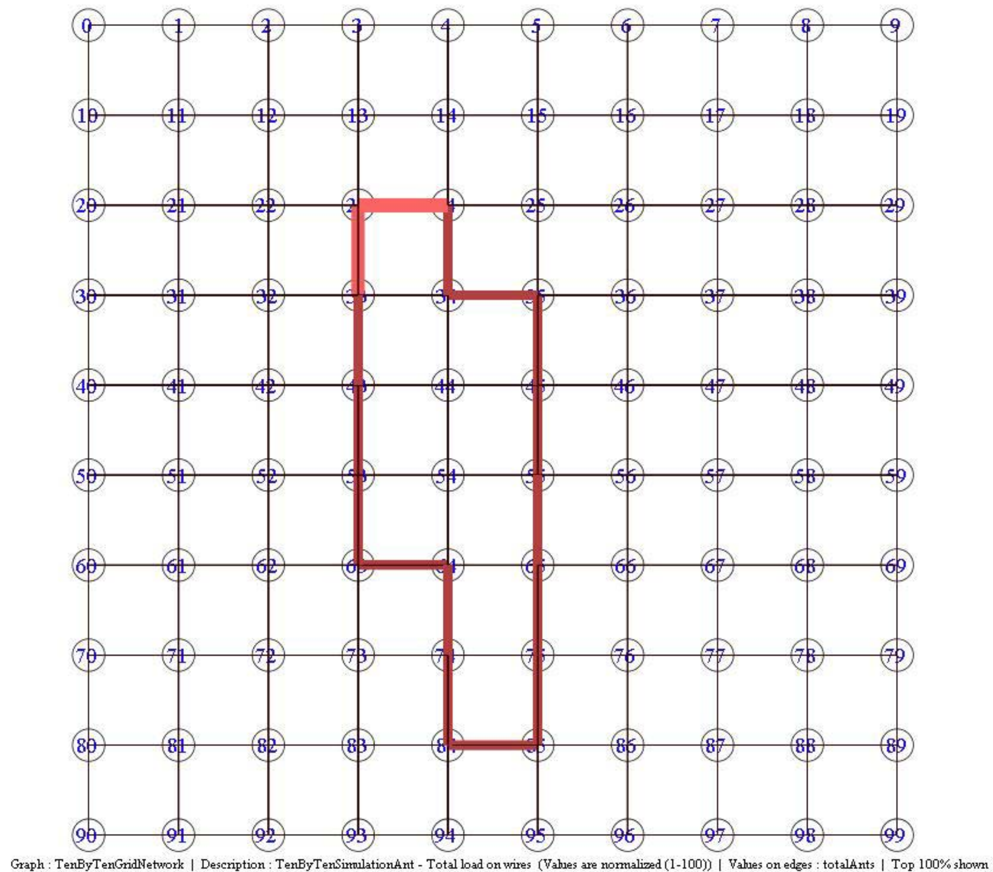

Figure 7 : The paths generated from experiment 1.1

Number of packages	10000
Dead	3431
TTL	0
INPUT	3050
OUPUT	216
NPRDies	165
Average time for package	452.68

Table 1: Results from experiment 1.1

[13] The results of all experiments can be found in Appendix C, which is divided into one subchapter for each experiment.

In experiment 1.2 we reversed the settings, so we gave local heuristics a higher priority than trail. Because of this setting we predicted that the packages are distributed over the entire network and that they had a larger probability of dying of no-possible-routing than in experiment 1.1. As anticipated the packages has no favorite path and most of the network is traversed by packages (see Figure 8[14]). The distribution of where the packages is lost are also shifted, so that approximately an even number are lost on input queues and due to no-possible-routing.

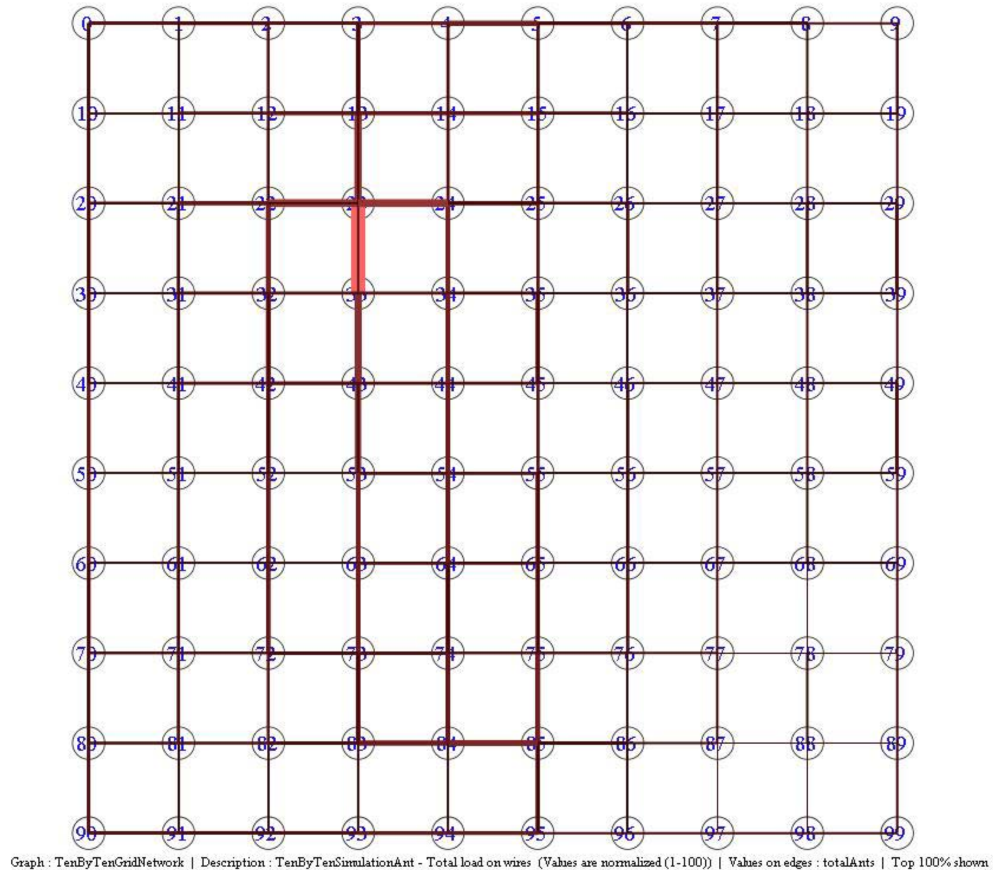

Figure 8 : The paths generated from experiment 1.2

Number of packages	5000
Dead	2866
TTL	0
INPUT	1286
OUPUT	0
NPRDies	1580
Average time for package	672.9

Table 2: Results from experiment 1.2

[14] This picture is generated by our Java-package by use of our modification of the grappa-package. The color/width of the edges represents the total number of ants, which has traversed this edge in a simulation (the coloring and width is normalized in relationship to the most traversed edge in the simulation). The redder/wider the edge the more ants has traversed the edge.

In the experiment 1.3 (see Figure 9) we basically performed experiment 1.1 again just with negative feedback, so that the algorithm should be able to react to loss of a package. We predicted that the packages were routed by a few good paths, but that these would be sidestepped for a time-period when packages were lost. The negative feedback did as expected and forced the packages to follow more paths than in experiment 1.1. and the paths are moreover longer. The negative feedback also reduced the number of lost packages as expected (to 11% of the original number).

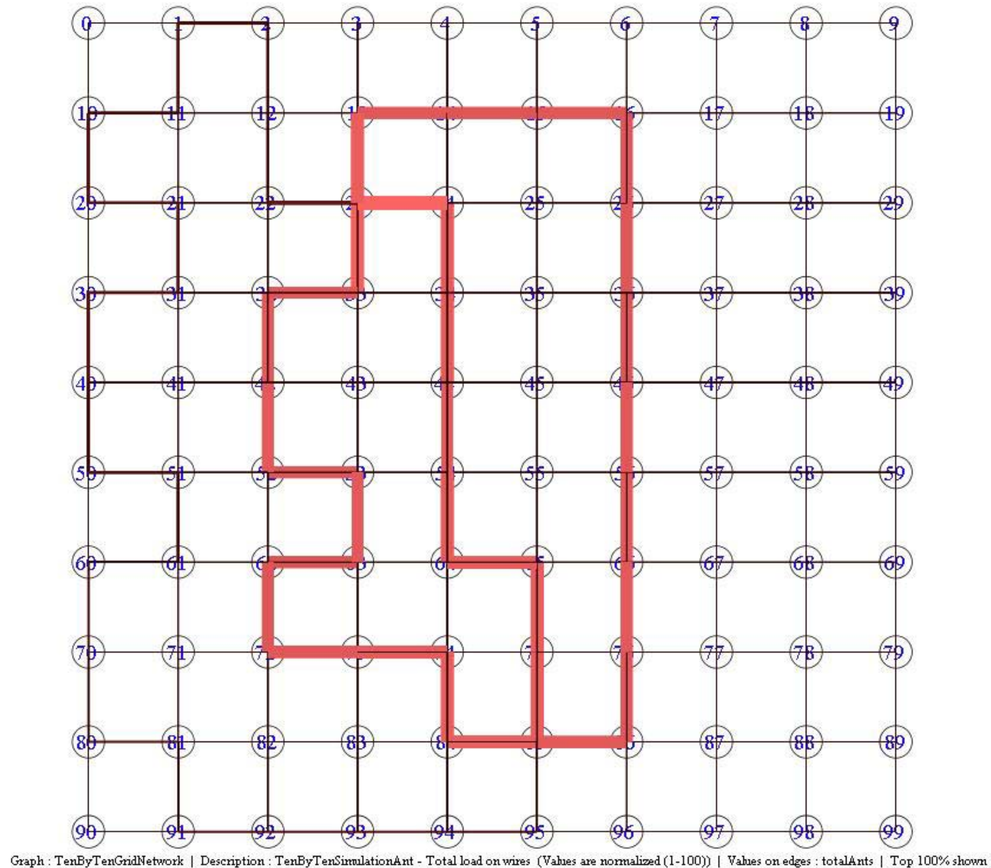

Figure 9 : The paths generated from experiment 1.3

Number of packages	5000
Dead	362
TTL	0
INPUT	216
OUPUT	0
NPRDies	146
Average time for package	396.12

Table 3: Results from experiment 1.3

Experiment 1.4 is experiment 1.2 with negative feedback. We expected that this would force the packages to be routed to routers with a low load. As can be seen on Figure 10 the packages are distributed over the entire network, as a result from this almost 3/5 of the packages were lost in the network primarily due to no-possible-routing.

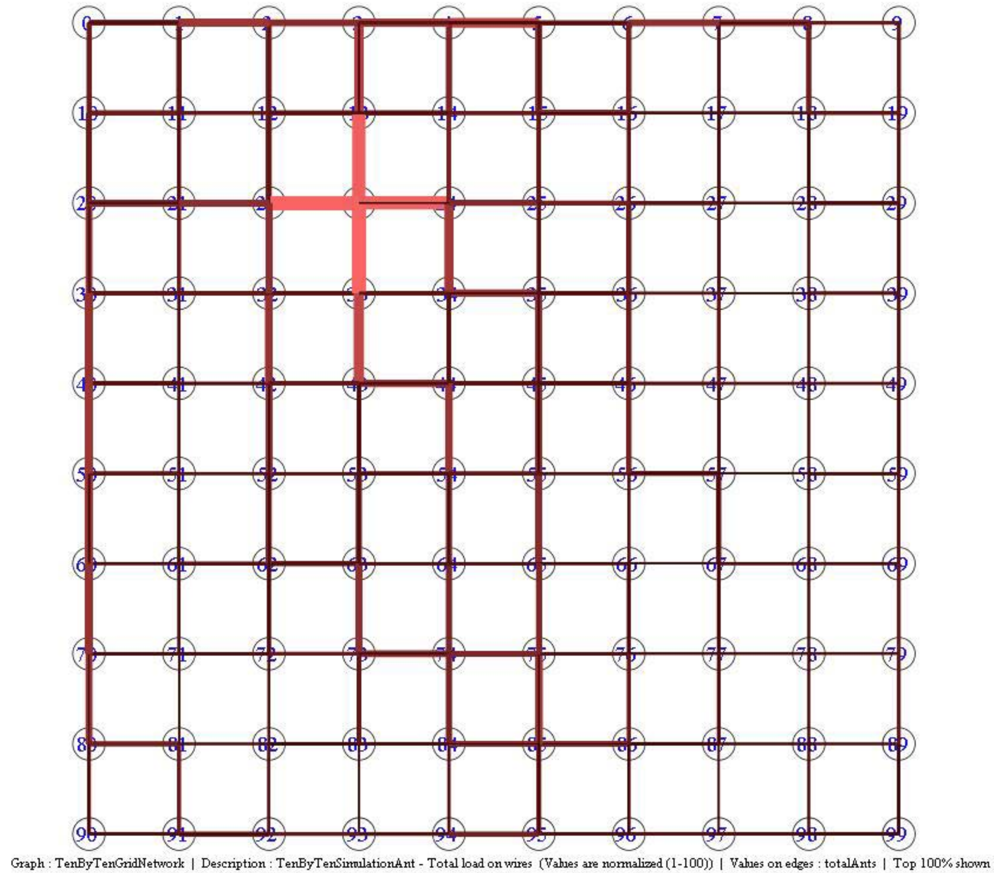

Figure 10 : The paths generated from experiment 1.4

Number of packages	5000
Dead	2849
TTL	0
INPUT	851
OUPUT	0
NPRDies	1998
Average time for package	789.38

Table 4: Results from experiment 1.4

Summary

We have shown that it is possible to control how ARS routes in a (grid) network by varying the parameters given to ARS. By varying α and β relatively to each other it is possible to control if ARS should send all the traffic by some highly loaded paths or if it should distribute the load out on more routers by not converging to just some paths.

By also adjusting on negative feedback it is possible to reduce the loss of packages considerably, but this also forces ARS to utilize more routers/paths. This also increases the average time that a package spends in the network.

The adjustments to the different parameters also greatly affect which type of package deaths we experience. If the packages are forced along some highly loaded paths the packages are lost on the queues of the routers. The more the packages are distributed the greater is the possibility of no-possible-routing deaths, because of the packages reaching a dead end due to their taboo-lists. There is no possibility of packages dying because

of TTL in ARS. This is because of the taboo-list and the size of TTL, which allows the packages only to visit *x* routers, where *x* is equal to the number of routers in the network. So TTL could only happen if a package visited a router twice (this is not allowed by the taboo-list).

Experiment 2 – double bridge network with heavy network load

In experiment 2.1, where all wires had an equal delay, we expected to find that the system did not converge to a single path, because a single path could not handle the entire load in the network. As can be seen on Figure 11 (because the experiments 2.1, 2.2, and 2.3 all generated approximately the same picture, we decided to use only one picture for all three experiments) most of the traffic is handled by the left branch and the remaining traffic is handled by the right branch.

In experiment 2.2 the picture does not change much. In this experiment we have a larger delay on wire *X*[15] and this forces more traffic to the left branch and the remaining traffic is handled by the right branch, which we expected. The result of this can be seen on Figure 11.

As we stated under our theses for experiments 2.3, we predicted that the outcome of experiment 2.3 should be the same as the outcome of experiment 2.2, which showed to be correct.

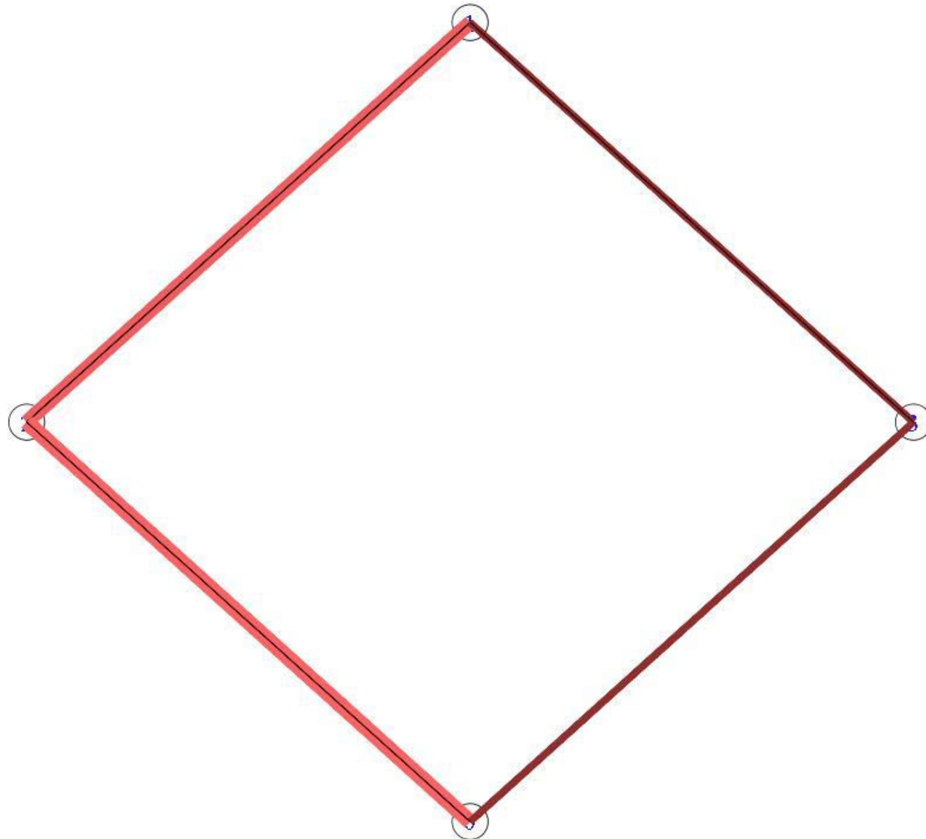

Figure 11 : The paths generated from experiment 2.1, 2.2, and 2.3

[15] See Figure 5 for the names of the routers and the wires.

2.1	
Number of packages	1000
Dead	33
TTL	0
INPUT	0
OUPUT	33
NPRDies	0
Avg. time for package	1073.66

2.2	
Number of packages	1000
Dead	148
TTL	0
INPUT	0
OUPUT	148
NPRDies	0
Avg. time for package	1158.49

2.3	
Number of packages	1000
Dead	152
TTL	0
INPUT	0
OUPUT	152
NPRDies	0
Avg. time for package	1151.3

Table 5: Results from experiment 2.1 & 2.2 & 2.3

The picture below (Figure 12) show the result of running our 'benchmark' algorithm on the setup of experiment 2.4, where the wires *X* and *Y* has a longer delay than *V* and *Z*. We allowed the 'benchmark' algorithm to update all its routing tables every 50 time-step in the simulation. We predicted that to fully utilize the network, the routing algorithm had to use both branches – and this is done by our 'benchmark' algorithm. Our 'benchmark' algorithm handles the job of routing the packages such that it 'only' losses 19% of them, which is near-optimal due to the load of the network.

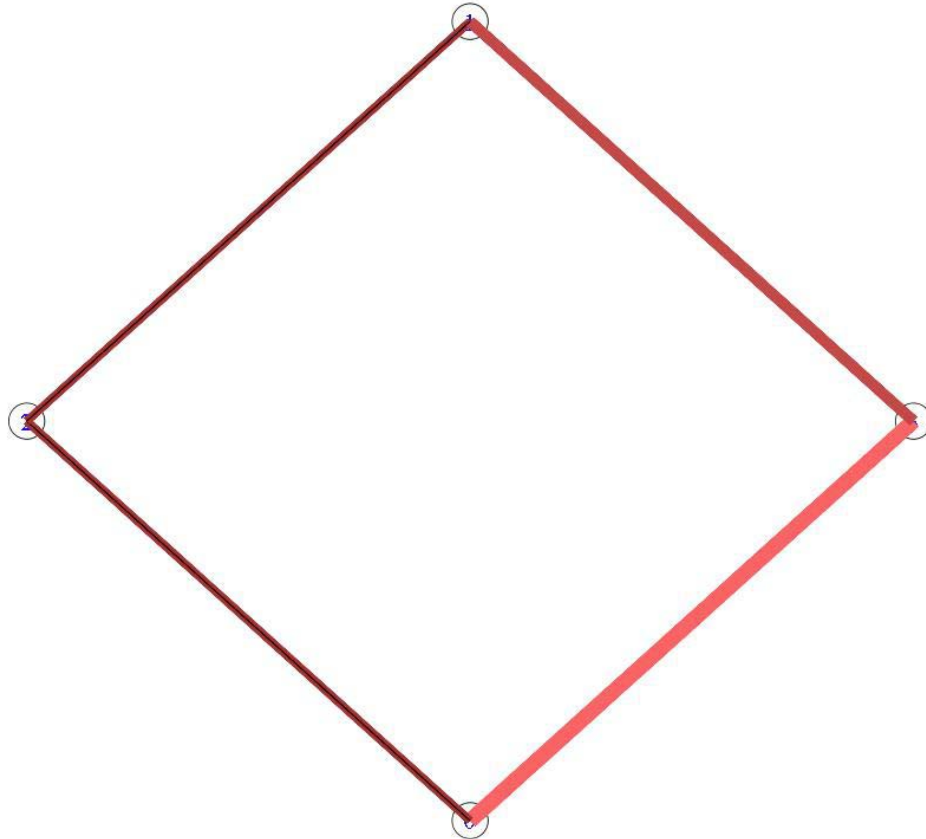

Figure 12 : The paths generated by our 'benchmark' algorithm from experiment 2.4

Number of packages	1000
Dead	186
TTL	0
INPUT	0
OUPUT	186
NPRDies	0
Average time for package	1132.38

Table 6: Results from experiment 2.4 'benchmark'

The picture (Figure 13) from our experiment 2.4 where ARS is run shows a result similar to Figure 12. Both the branches are utilized in order to fully exploit the network. The primary branch is this time the left branch (the opposite of the 'benchmark' algorithm) because of the local heuristic. ARS almost handles the job of routing as well as the 'benchmark' algorithm as it 'only' losses 24% of the packages.

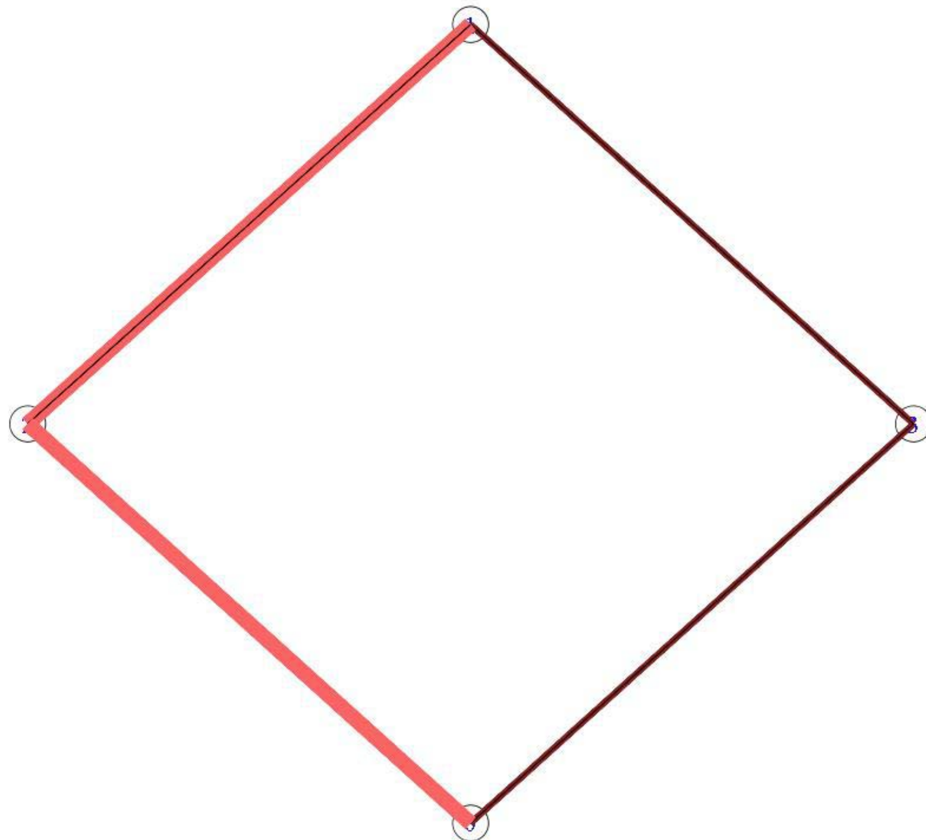

Figure 13 : The paths generated from experiment 2.4

Number of packages	1000
Dead	239
TTL	0
INPUT	0
OUPUT	239
NPRDies	0
Average time for package	1149.09

Table 7: Results from experiment 2.4 "ARS"

Summary

In this experiment we wanted to show that in a heavily loaded simple network ARS is able to utilize several paths to achieve a better throughput than by just using a single path. All our tests have shown that ARS is able to route most of the traffic by one branch (until it cannot handle more) and the remaining traffic by the other branch.

Compared to our 'benchmark'-algorithm, which was able to only loose 19% of the packages, ARS performs a little worse by loosing 24%, but because of the network setup this is quite reasonable.

Experiment 3 –10x10 grid network with heavy network load

Our third experiment was carried out in the 10x10-grid network where the network load was heavy. In this experiment we ran both ARS and our 'benchmark'-algorithm and compared the results. Like previously we allowed our 'benchmark'-algorithm to update all its routing tables at every 50th time-step.

The picture below (Figure 14) shows how our 'benchmark'-algorithm chose to route the packages. As can be seen on the picture the algorithm distributes the packages over most of the routers in the 'center' of the network and there are no clear routes except near the start and destination routers.

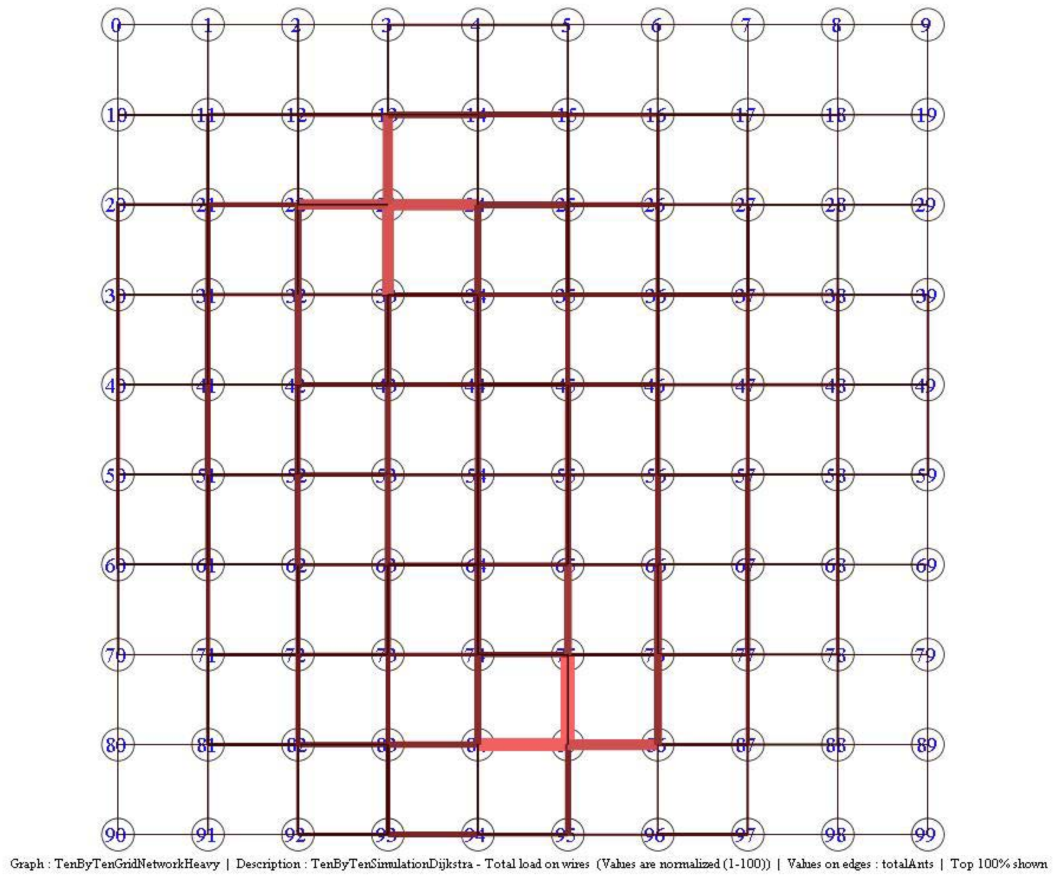

Figure 14 : The paths generated from experiment 3 by the “benchmark” algorithm

Number of packages	3000
Dead	14
TTL	0
INPUT	11
OUPUT	3
NPRDies	0
Average time for package	570.62

Table 8: Results from experiment 3 "Benchmark"

ARS (see Figure 15) on the other hand chooses to utilize two short paths and two longer paths to route the packages by. ARS also routes more traffic by the two short paths and less traffic by the two longer paths (this can be difficult to seen on the pictures when they are not in color).

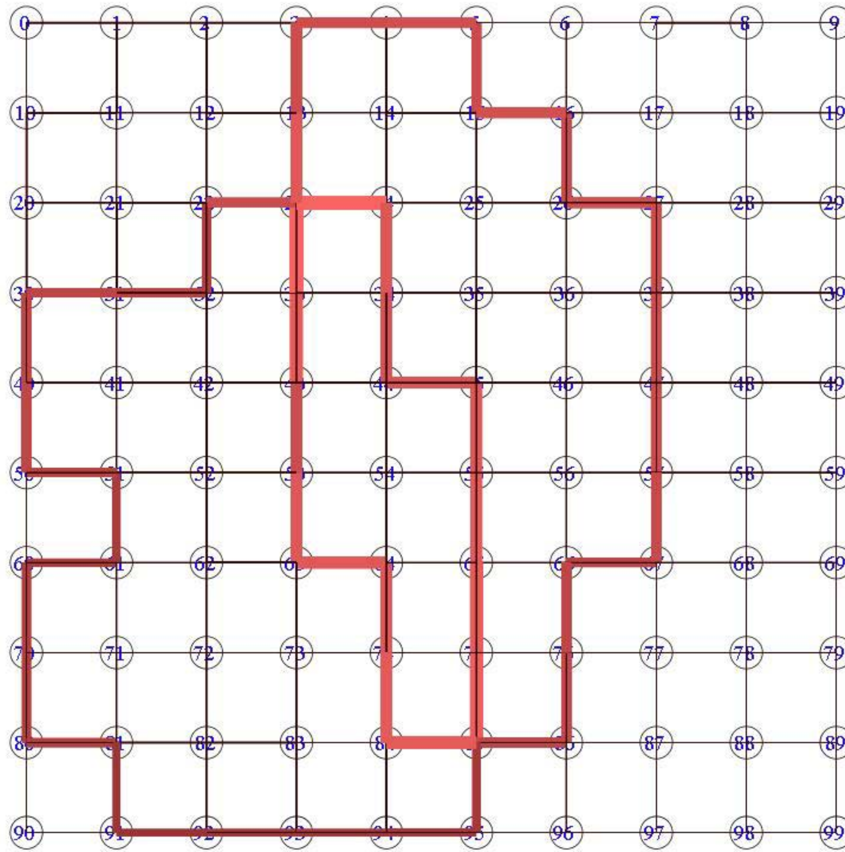

Figure 15 : The paths generated from experiment 3 by ARS

Number of packages	3000
Dead	383
TTL	0
INPUT	175
OUPUT	4
NPRDies	204
Total time load of packages	1412167
Average time for package	539.61

Table 9: Results from experiment 3 - ARS

Comparison of ARS and the 'benchmark' algorithm

As can be seen in Table 10 of the 3000 packages sent from router 23 to router 85 our 'benchmark' algorithm only drops 14 of the packages (approximately 0.5%), while ARS drops 383 packages (approximately 12.7%). A great deal of the packages dropped by ARS (about 280), are dropped in the initial convergence (see next section) and the remaining 100 in the rest of the simulation (see [Experiment 3, Appendix C]). The types of package death support this, because NPRDies (death from no-possible-routing) indicates that the packages have reached a dead end due to their taboo-list.

	ARS	"Benchmark" algorithm
Number of packages	3000	3000
Dead	383	14
TTL	0	0
INPUT	175	11
OUPUT	4	3
NPRDies	204	0
Average time for package	539.61	570.63

Table 10 : Comparison of ARS and the 'benchmark' algorithm

A surprising observation is that ARS is able to route the packages faster through the network than the 'benchmark' algorithm. The reason for this is the difference in the way the traffic is distributed (see Figure 14 and Figure 15). The 'benchmark' algorithm distributes the traffic by utilizing most of the routers in the middle of the network without any clear pattern and ARS distributes the traffic by using four different paths. The reason why our benchmark algorithm is slower is that it is recalculated every 50 time-step. Because of this continuous recalculation (based partly of the load of the path) the routing tables are always changing. So it is quite possible that a package is routed back and forth between two routers (with approximately the same delay to the destination router) due to the loads on the neighbor routers.

This problem occurs because we include load into the delay calculation for the path [Tannenbaum 1996, p. 361], so that the routing tables oscillate between several paths to the destination.

This could force our 'benchmark' algorithm into problems with TTL, but because TTL is in our algorithms as default equal to the number of routers in the network (100 in this case) this did not happen. This can also be seen in Table 10 where none of the algorithms losses packages due to TTL.

Clearly there is a compromise between getting all the packages through the network and getting them fast through the network.

Initial convergence

Figure 16 shows the number of dead ants as a function of time. This curve is typical for all longer simulations in networks with heavy load where both positive and negative feedback is used. As Figure 16 shows we can divide the simulation in two sections: the initialization and the rest of the simulation. A proportionally large number of ants die in the initialization, while the system is still 'configuring' to the environment, but this trend flattens off as ARS converges to a balance between the different paths found guided by the positive- and negative feedback.

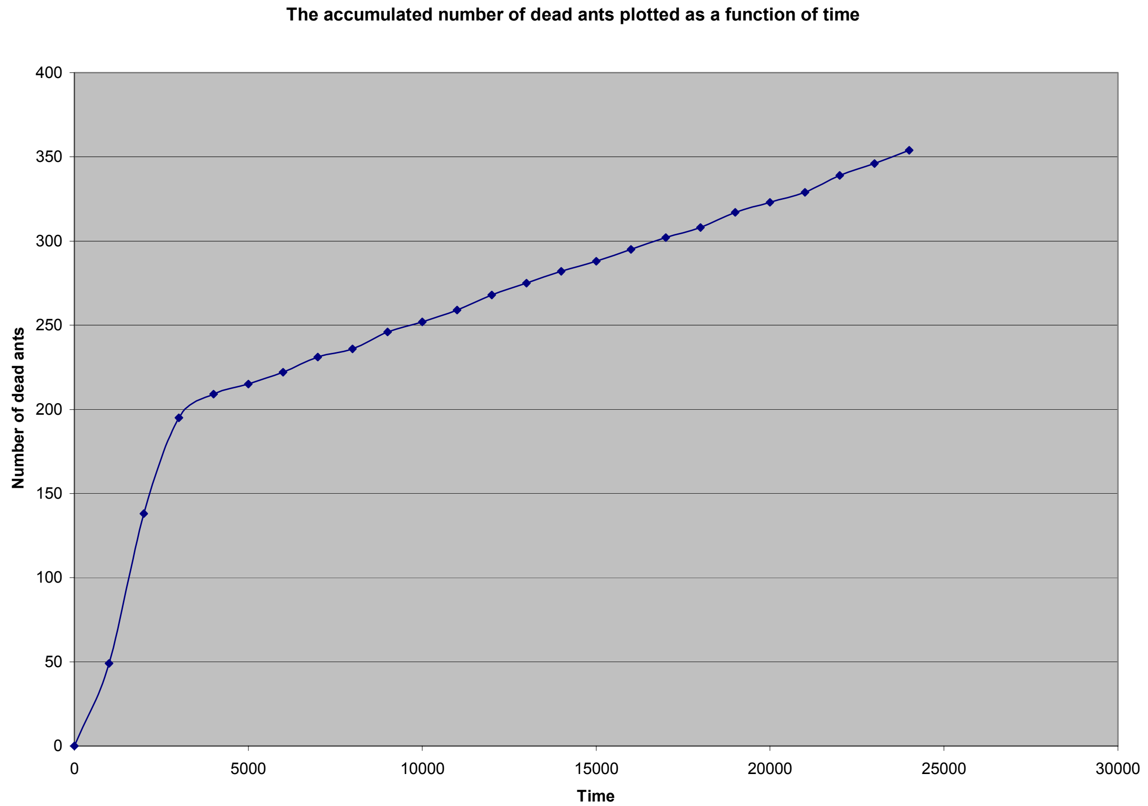

Figure 16 : The accumulated number of dead ants plotted as a function of time

Figure 17 shows the number of dead ants per interval each of the 24 intervals (0–999, 1000-1999,…). The figure shows how the number of dead ants per interval in the beginning of the simulation increases, but already after the 2000 time-step this tendency is reversed. And from time-step 4000 and forward the number of dead ants varies between 5 and 10 ants.

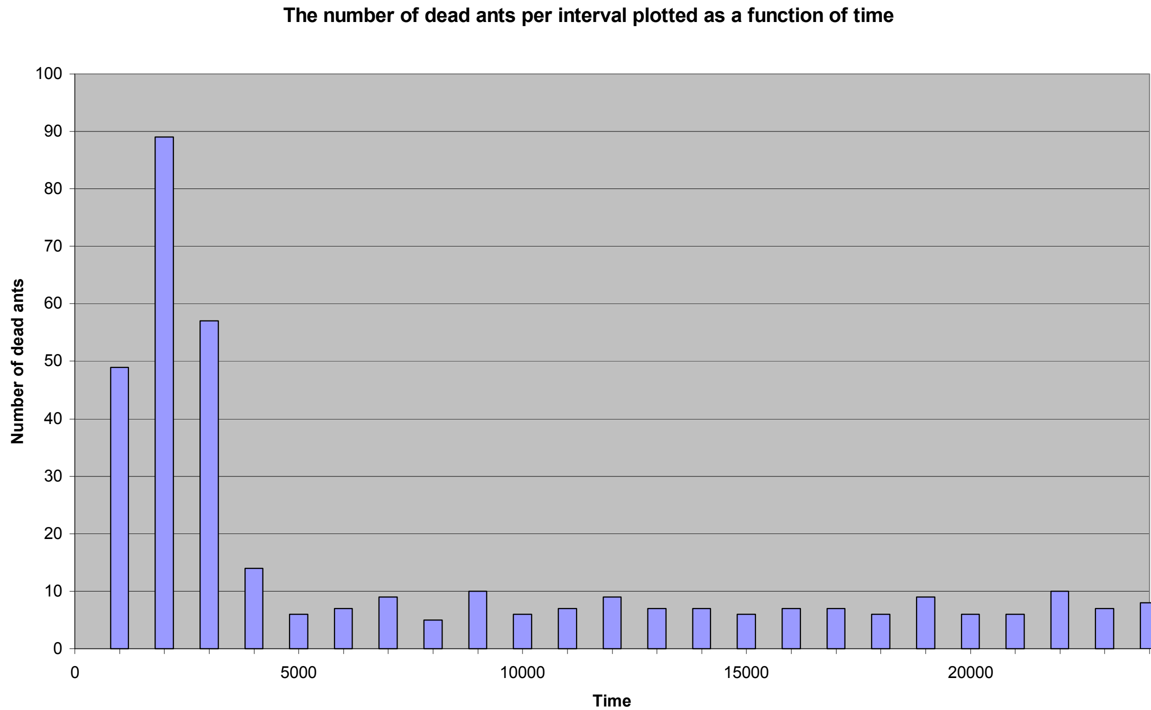

Figure 17 : The number of dead ants per interval plotted as a function of time

Summary

We have in this chapter demonstrated that ARS is highly configurable with regards to how much distribution one wants, if the packages should be routed fast or more reliable. We have also shown that in a heavily loaded network ARS is able to utilize several paths to achieve a better throughput than by just using a single path. And in a heavily loaded network set up as a double bridge experiment ARS is almost as reliable as our 'benchmark'-algorithm.

In a heavily loaded network setup as a 10x10 grid ARS was able to route the packages faster through the network than our 'benchmark' algorithm, which utilizes most of the routers in the center of the network. Our 'benchmark' algorithm has problems (because of its routing based on the load) with packages being routed back and forth between routers. Because of the parameters given to ARS it is able to use some good paths from the source to the destination (with a greater package loss however).

A great deal of the package loss in networks with heavy load happens in the initial phase where ARS is still 'configuring' to the environment. But the rate at which the packages are lost drops considerably, when ARS has been allowed to 'discover' the network.

Conclusion

The main purpose of our project was to study ant algorithms applied to a dynamic domain. It is our belief that we have succeeded in this. We have implemented a routing algorithm working on a simulated network and conducted simulations to test the algorithm.

The main question that was to be answered in this project was:

Can an ant-inspired routing algorithm be used to obtain a distribution of package transfers, which result in improved throughput and reduced package loss in a simulated package-switched point-to-point network?

We have demonstrated that by combining positive feedback, negative feedback and local heuristics, ARS can distribute the package transfers over several paths in a heavily loaded simulated network when packages are sent between two distinct routers. This ensures a higher throughput and a lower package loss. The general case of packages being transferred between all routers in the network remains unexplored due to the time constraints of this project.

We have demonstrated that by prioritizing local heuristics and negative feedback over trail following behavior we can adjust Ant Routing System (ARS) towards distribution of package-traffic. This is at the cost of increased routing time and increased package death as a consequence of the ant-packages taboo-lists.

We have also demonstrated that by prioritizing trail following behavior over local heuristics and negative feedback ARS is able to find short paths and use them indiscriminately. This results in package loss in a heavily loaded simulated network because the routers on the used paths do not possess the capacity to handle the traffic.

While the time has been too brief to produce extensive and statistically valid tests, we have through our work attained such familiarity with the problems and our implementations that we are confident about the usefulness of our results. It is important to bear in mind that our goal was to *demonstrate* that the given problems could be solved with ant algorithms, while we did not aim to *prove* it.

ARS is our main accomplishment. We were able to lift ant algorithms out of the graph environment, and also able to make it work in a dynamic environment. We obtained interesting results from combining the different mechanisms that composed ARS. We were able to adjust the behavior of the algorithm based on which properties we wanted to be reinforced, being this a greater or lesser tendency to distribution, exploration, or to reduce package loss.

Ideas for further studies and implementations

We have illustrated some basic properties of ARS, but many questions still remain unanswered. How will ARS react when traffic is sent from all routers in the network? What is the practical overhead of using ARS in a real network? Is it even practically applicable to a real network?

One could speculate that using ant-inspired algorithms to route packages is a sensible choice given the adaptive nature of ant algorithms. On the other hand ant inspired algorithms are highly dependent on parameters that determine the relevance of different heuristics.

Can it be guaranteed that a set of parameters exist that will allow ant inspired algorithms to effectively route packages in any network topology under different traffic loads, or is it required that ant algorithms can make intelligent runtime adjustments to their parameters in order to ensure optimal routing?

We will end this project with a description of some interesting thoughts that we have had while working on this project.

Ant algorithms and inductive machine learning algorithms

Finally we will shortly present a few thoughts and ideas that we have had during this project. We will formulate some ideas on how machine learning theory can be applied to ant algorithms. We will also present an idea on how ant algorithms could be combined with inductive machine learning.

Inductive machine learning algorithm theory applied on ant algorithms

In the following we denote algorithms based on the ACO-metaheuristic as ant algorithms.

We saw in [*Problem representation*, p. 15] that the problem O for an ant algorithm was defined as $O = \{S, f, \Omega\}$, with S being all solutions, f the quality function, and Ω the constraints on the possible combinations of its components. We defined that all $s \in S$ had assigned a quality through the quality function f.

Given that the constraints Ω are respected, an ant algorithm and its domain needs only to be connected though the quality function f. This abstraction between algorithm and domain enables the algorithm to find solutions without knowing the intrinsic dynamics of the domain it is exploring. So even though the phenomenon is highly complex, and the interdependency of factors in the system are not known, and given that it is possible to measure the outcome of a process in that system, then it should be possible to make workable solutions for that system.

This is also an ability that is sought after when working with inductive machine learning algorithms, namely the ability to find structure and dynamics in bodies of highly complex empirical data.

Inspired by this we have made some very preliminary observations regarding similarities between ant algorithms and inductive machine learning algorithms (hereafter denoted as IMLA).

Graph algorithm or machine learning?

In our work with ant algorithms we have come across a number of issues regarding the conceptual understanding of ant algorithms, their function, and position with respect to other algorithms. One of the primary issues of speculation has been whether ant algorithms should be classified as machine learning algorithms.

The most basic task of ant algorithms is to find short paths in graph problems, which makes it useful in relation to problems like TSP and routing. On the other hand, an ant algorithm has in its function some similarities to inductive algorithms. For example, the ant algorithm receives information about a possible solution, and the quality assigned to the given solution. This is then repeated, until the ant algorithms can produce an acceptable solution.

The question that now presents itself is whether this adaptive capability has similarities with the ability to make categorization or generalization that IMLA have. We will discuss this after a brief introduction to some very basic concepts from machine learning.

Hypothesis space, Hypothesis and Quality

The term *hypothesis space* is introduced in the book *Machine Learning* [Mitchell 1997] where it is defined as any possible 'legal' state that the domain can produce as an input to the learning algorithm. (For example, for a neural network, it would be any possible input placed on the input layer). From this explanation it follows that a single hypothesis is a single legal input to the machine learning algorithm from (or in) its domain. This is equivalent to the formal definitions from [*Theory*, p. 5], and we thus conclude that:

$$HypothesisSpace = S$$

And *solutions* $s \in S$ are equivalent with *hypothesis* and we denote the nth solution as s_n.

Some IMLA operate with a quantifiable *quality* as a result of a given hypothesis, which is given to the IMLA together with the hypothesis for processing.

In short, an IMLA receives a number of pairs of {*hypothesis, quality*}. After processing these it will be able predict the *quality* of a new hypothesis given to it, with some degree of certainty. In the case that this quality is quantifiable, it will be possible to calculate the error percentage relative to the best known hypothesis.

So similar to the quality function *f* that was defined in [*Problem representation*, p. 15], we define a quality function $Q : S \to \mathbf{R}_+$. To quantify an error on basis of the 'best case' would be to say that the error e_s for the hypothesis *s* is defined as:

$$e_s = | Q(s_{best}) - Q(s) | ,$$

– where s_{best} is the best solution found so far. Then the percentile error ε_s for *s* can be calculated as:

(21) $$\varepsilon_s = \frac{| Q(s_{best}) - Q(s) |}{Q(s_{best})}$$

We then define the result space as:

$$R = \{ s \in S \mid \varepsilon_s \le \varepsilon_0 \} ,$$

– where ε_0 is an external value, typically decided by the user of the system. It could for example be 0.05, which would indicate that a solution would not diverge more than 5% from the best possible solution.

This formalization is valid for both IMLA and ant algorithms.

PAC learnability

For IMLA there exist the Probably Approximately Correct (denoted as PAC) framework, which makes it possible with a certain degree of probability, to determine the number of learning hypothesis required for an IMLA to produce a result within the error boundary ε_0 [Mitchell 1997, p. 201].

For inductive algorithms there is an inequality that gives a boundary for PAC learnability:

(22) $$m \ge \frac{1}{2{\varepsilon_0}^2} \left(\ln | S | + \ln\left(\frac{1}{\delta} \right) \right),$$

[Mitchell 1997, p. 211]

– where $|S|$ is the size of the hypothesis space, ε_0 is the maximum allowed percentile error for the produced solution, as according to the previous section, δ is the probability that results produced by the algorithm will *not* be correct within the ε_0 boundary[16], and *m* is the number of experiments needed.

The inequality above presents a measure of the probability of a machine learning algorithm to be able to present a solution with an error below a certain threshold, for a given number of 'learning cases', and for a given size of the hypothesis space.

PAC applied on Ant System

It could be interesting to investigate whether, and if so to which extent, the PAC-learnability rule could be applied to AS. It is outside our resources to do that here, but we would like to briefly present our preliminary work regarding this.

For TSP the hypothesis space *S* would consist of all the Hamiltonian cycles of the graph. And the length of each cycle would be the quality of the hypothesis.

For TSP the PAC- inequality could be used as follows:

[16] $(1-\delta)$ is therefore the probability that results produced by the algorithm will be correct within the ε_0 boundary.

The size of the hypothesis space for TSP is (as mentioned in [*Hamiltonian cycle*, p. 5])

(23) $$|S| = (|V|-1)!\ ,$$

– where $|V|$ is the number of vertices in the graph.

Combined with equation (22) this would give

(24) $$m \geq \frac{1}{2\varepsilon_0^{\,2}}\left(\ln\left((|V|-1)!\right)+\ln\left(\frac{1}{\delta}\right)\right)$$

In the following we will use some sample values extracted from a typical run of our Ant System implementation. This should not be perceived as a statistically valid test. It is merely an indication of a lead that could be interesting to follow (and might be so in another report). Firstly we will define and calculate the values for the different parameters in the inequality:

$|S|$: We have used the benchmark graph bayg29, which has 29 vertices, as a sample graph from TSPLIB[17] for solving TSP. Inserted in equation (23) it gives that $|S| = 28!$.

ε_0: The optimal solution to TSP on this graph is 1610, which means that $Q(s_{best}) = 1610$. In appendix D we have include a sample run for Ant System on bayg29 in Figure 3 p. 23. A typically observed[18] $Q(s) = 1764$.

We determined ε_0 according to equation (21), with the $Q(s_{best}) = 1610$, and a typical observed $Q(s) = 1764$. When inserted in equation (21) the resulting ε_0 is 0.0956, or ~9.5%. Please note that we earlier mentioned that ε_0 was typically given to the system as a parameter. In this case we have calculated ε_0 instead, to get an idea of the size of the error.

δ: To be conservative, we chose that $(1 - \delta)$ should be 0.999, meaning that the probability of getting a result with an error less than ~9.5 should be 99.9%. δ is then 1-0.999 = 0.001.

For $|V| = 29$, $\varepsilon_0 = 0.0956$, and $(1 - \delta) = 0.999$, equation (24) gives that $m \geq\sim 4088$

In other words: if you want to be 99.9 percent sure that you can find a solution to TSP using (our implementation of) AS that has an error that is less than ~9.5 %, then you have to use approximately 4088 ants (according to the PAC applied to a single result).

Our observations do not indicate whether Ant System abides by this. (It would take additional time to make an empirical study of this). It could be interesting though, since the convergence speed (for ant algorithms) is a parameter that is hard to quantify, and the above framework might contain some productive leads of how it could be done.

Summary

In the above, we applied the PAC inequality to ant algorithms. Whether this application is valid or not remains to be investigated. Nevertheless the idea of predicting the probability of a successful outcome is enticing.

Inductive machine learning algorithms combined with ant algorithms

Hypothesis suggestion vs. quality function emulation

Generally speaking, ant algorithms can be said to use heuristics to delimit the hypothesis space, and through the Stochastic State Transition mechanism explore the hypothesis space. The ant algorithms can suggest

[17] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/

[18] The run produced the best result 1652 at the 1642th ant. The value 1764 has been chosen arbitrarily to indicate that it is a possible value encountered early in a specific run. We encountered the value 1764 at the 384th ant. We ran the ant TSP with 2550 ants (51 iterations).

hypotheses that can be tested with the *Q*-function. Ant algorithms learn to produce (suggest) viable hypotheses.

IMLA mainly operate through receiving already suggested hypotheses *s* and the resulting quality *Q*(*s*). The IMLA consequentially results in approximating the quality function *Q*. This is a good idea when *Q* is not known, and the overall task is to be able to classify hypotheses according to *Q*. IMLA learn to classify hypotheses.

The curious ant

As we mentioned above, the ant algorithms produces hypothesis, which are tested. The ant algorithm then adapts to these tests, and produce new hypotheses that (hopefully) produce even better results.

We believe that the ability to *produce* feasible solutions requires less learning than the ability to *recognize* feasible solutions and classify them accordingly.

IMLA like neural networks learns from pre-selected hypotheses and their results. While neural networks would need a large number of hypotheses to generalize over the entire hypothesis space, ant algorithms are primarily interested in hypotheses that can lead to a ‘working solution’.

A visual metaphor could be the following: An IMLA learns about the entire city map to learn to deduct whether a selected route would be good or not. An ant algorithm only adjusts to a route from A to B, and could thus forget about the rest of the city map.

Local exploration and overrepresentation

This advantage is even more visible in dynamic domains. Assuming that changes occur in the domain, thus changing the quality of solutions the ant algorithm has converged towards, the ant algorithm need only focus on the probability distributions that effects the construction of solutions negatively after the change in the domain. Since it only needs to relearn locally, it can adopt to the changing environment very quickly.

An IMLA like neural networks is engineered towards maintaining a representation of the entire hypothesis space, in a generalized form. The task of only updating experience regarding the local change could therefore be troublesome. The representation is based on learning hypotheses that are distributed (more or less evenly) over the totality of the hypothesis space. To adjust to the local change would mean that the system should receive a number of new learning hypotheses teaching it about the new state of the local area.

The task of identifying classes of hypotheses that ‘teach’ about a given local area is not trivial. Even if such hypothesis classes could be identified, there is the problem of the hypothesis space being ‘over-represented’ in that area.

If a neural network is taught to classify pictures of people by gender, and suddenly men with mustache are heavily over-represented in the empirical material, well shaven men risk the possibility of being wrongly classified, simply because the machine believes that most men have mustache.

So the speculation is here that ant algorithms have an advantage over IMLA when adjusting to local changes in the domain.

Combining IMLA and ant algorithms

IMLA has the ability to build quality functions on the basis of a number of learning hypotheses. The IMLA generalizes over a large number of hypotheses with corresponding qualities, and based on this it is able to evaluate a new hypothesis.

An ant algorithm needs a quality function to evaluate suggested hypotheses in order to suggest better ones. In Ant System, the quality was obtained through a measurement of the walked distance, likewise in ARS. But it is conceivable that a problem could have a troublesome quality function, if for example the problem involved submitting a hypothesis to a test. This test could be troublesome, costly, or otherwise unwanted, especially if taking into account that an ant algorithm would need to submit a large number of bad hypotheses to produce a good one.

It could perhaps be interesting to speculate how these two mechanisms could be combined:

First a quality function Q is build according to a large number of empirical observations about a phenomenon. This quality function could then be the basis for an ant algorithm to build new hypotheses regarding the phenomenon. In other words we could feed the hypotheses suggested by the ant algorithm to the quality function provided by the IMLA.

The advantage of the above scheme is that it is possible to test suggested hypotheses without submitting them to the environment providing the initial empirical observations. This could mean that substantial time could be saved, because the submission of suggested hypotheses could be time costly, or in another way hard to do. An additional effect could be that previously collected empirical data could be put to good use.

The advantage of the above scheme would be that the subjection of the large number of learning hypotheses produced by the ant algorithms to the environment could be avoided. The assumption for this is of course that the IMLA would (implicitly) be able to accurately capture the significant information about the hypothesis space.

The result of this is that ant algorithms could be used to extract information from an IMLA that has been used to extract information from an empirical body.

Bibliography

Bonabeau, E., Dorigo M. & Theraulaz G. (1999). *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press.

Bonabeau, E., Dorigo M. & Theraulaz G. (2000). “Ant algorithms and stigmergy” *FGCS* **16** pp. 851-871.

Bonabeu E. & Théraulaz G. (2000). “Swarm smarts”, *Scientific American*, pp. 72-79, March issue.

Colorni, A., Dorigo M. & Maniezzo V. (1991). “Ant System – An Autocatalytic Optimising Process”, *Technical Report IRIDIA*/1991-016, Italy: Politecnico di Milano.

Colorni, A., Dorigo M. & Maniezzo V. (1996). “The Ant System: Optimization by a colony of cooperating agents”, *IEEE Transactions on Systems, Man, and Cybernetics*-part B, Vol. 26, No.1, pp.1-13.

Cormen, T. H., Leiserson C. E., Rivest R. L. & Stein C. (2001). *Introduction to Algorithms* (Second Edition), MIT Press/McGraw-Hill.

Di Caro, G. & Dorigo M. (1998). “Ant colonies for Adaptive Routing in Packet-switched Communications Networks”*, Parallel Problem Solving from Nature*, pp. 673-682.

Di Caro, G., Dorigo M. & Gambardella L. M. (1999). “Ant Algorithms for Discrete Optimization”, *Artificial Life* **5(2)** pp. 137-172.

Di Caro, G., Dorigo M. & Stuetzle T. (2000). “Guest editorial – Ant algorithms”, *FGCS* **16** pp. v-vii

Dorigo, M. & Gambardella L. M. (1997). “Ant Colonies for the Travelling Salesman Problem”, *Technical Report IRIDIA*/1996-3, Belgium: Université Libre de Bruxelles.

Dorigo, M. & Stuetzle T. (2000). “The ant colony optimization metaheuristic: Algorithms, applications and advances”, *Technical Report IRIDIA*/2000-32, Belgium: Université Libre de Bruxelles.

Gutjahr, W. J. (2000). “A Graph-based Ant System and its convergence”, *FGCS* **16** pp. 873-888.

Mitchell, T. M. (1997). *Machine Learning*, Singapore: MacGraw-Hill.

Tannenbaum, A. S. (1996). *Computer Networks* (Third Edition), Prentice Hall.

Weiss, M. A. (1998). *Data Structures and Problem Solving Using Java*, Addison-Wesley.

White, T. (1997). *“*Routing with Swarm Intelligence”*,* Technical Report SCE-97-15, *Systems and Computer Engineering*, Carleton University,.