Master Thesis in Computer Science

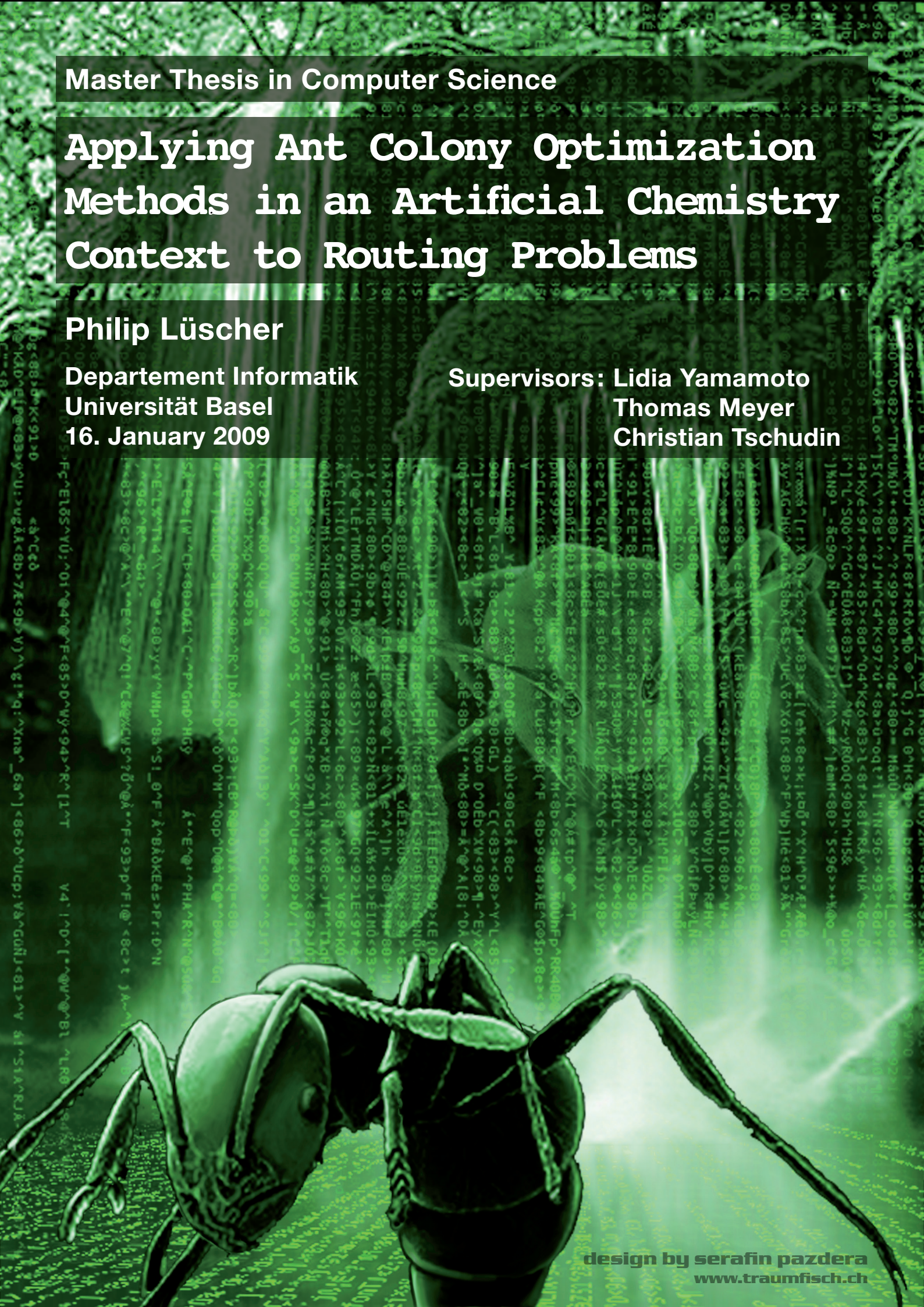
Applying Ant Colony Optimization Methods in an Artificial Chemistry Context to Routing Problems

Philip Lüscher

Departement Informatik
Universität Basel

Supervisors: Lidia Yamamoto
Thomas Meyer
Christian Tschudin

16. January 2009



design by serafin pazdera
www.traumfisch.ch

Abstract

The foraging task of ants show several similarities with the task of packet routing in networks. This thesis describes the implementation of a combination of existing ant inspired routing algorithms in fraglets, an artificial chemistry. The behavior of the algorithm was studied with network simulations based of an integration of the fraglets environment into OMNeT++. And with the help of the simulation results the influence of several parameters is explained. The main outcome is to show the feasibility of implementing an ants routing algorithm using fraglets.

Acknowledgments

I would like to thank my supervisors Christian Tschudin, Lidia Yamamoto and Thomas Meyer for the guidance and great support during the research and preparation of this thesis.

I am especially grateful to Lidia Yamamoto for her generous help in organizational issues and Thomas Meyer for the technical insights into the fraglets environment as well as for their very helpful suggestions and feedback for the thesis.

Furthermore I would like to thank Serafin Pazdera for his excellent design of the title page.

Contents

Chapter 1

Introduction

Biological systems have developed remarkable problem solving strategies. One example are trail laying ant, which organize their foraging activity in a decentralized way. This observed behavior can be mapped to technical problems in computer science and used as a source of inspiration for new algorithms.

Especially the problem of network routing has a couple of similarities with the foraging task of real ants. In the case of routing at each node a neighbor has to be selected to which a packet is sent. The selection is done based on locally available information. This process corresponds to the path finding behavior of real ants which is guided by pheromones.

There are already several implementations of routing algorithms based on ant behavior. Two of them, AntNet and AntHocNet, are described in this document.

Since foraging strategies of real ants are based on the concentrations of pheromones the idea has arisen to build an ant inspired routing algorithm based on a artificial chemistry in which the program execution is controlled by concentrations.

For the implementation of an algorithm similar to the existing approach the fraglets environment was chosen which is a artificial chemistry designed to solve networking tasks. Even though the fraglets environment already contains basic networks simulation capabilities, OMNeT++ was chosen for the simulations due to its high flexibility and rich features, especially the visualization and data gathering capabilities.

The behavior of the implemented algorithm was studied with the help of several network scenarios. Thereby the influence of parameter settings to the behavior was of special interest. Mostly simple network scenario are used to get a better understanding of the exact behavior of the virtual ants, as it is too complex to control and observe single paths of ants in bigger networks.

Besides the question if it is possible to implement a working ant inspired

routing algorithm in fraglets which was verified by the simulations, important aspects were the questions how the several parts of a routing algorithm can be mapped to chemical environment and if there is specific functionality that cannot be reproduced with fraglets.

Chapter 2

Ants

Ants are social insects that live in huge colonies, where each ant takes on a small function to ensure the survival of the whole colony. All ants together perform more complex tasks than a single ant would be capable of. That is because ants do not only act as individuals but they also develop collective behavior based on self-organizing processes within the colony.

2.1 Real Ants

Most ant colonies show some sort of collective and decentralized self-organizing behavior but the effect produced by some specific species are really astonishing. One of such a phenomenon are the living rafts of the red imported fire ants (Solenopsis invicta) which they form when the nest and the surroundings are flooded. In such a situation the colony forms a disc-like structure consisting only of living animals. Through the weight of the colony some of the workers are pushed under water, but they reposition themselves after a short time, so no individual stays too long under water. During the floating the queen and the brood stay well protected in the middle of the colony raft. As soon as dry land is reached the raft is dissolved and a place for a new nest is searched [1].

Another fascinating peculiarity are the living chains formed by weaver ants (genus Oecophylla) which are used as ropes to pull leaves together to form a nest. Then the leaves are glued together by other workers that use the silk producing larvae like glue guns [14].

2.1.1 Foraging

One aspect of the collective behavior shown by many ant species are different foraging strategies. The strength of foraging strategy of the black garden ant (Lasius niger) is very easy to observe because it is a common species and lives in almost every backyard. It is not unusual that a nearby living colony

finds a food source (i.e. a piece of a sweet fruit) which lies in the middle of a house within hours and if a source is found after a short while there is a line of ants transporting food back to the nest and looking for other food sources in the neighborhood.

One strategy common to many species is when scout ants randomly start to search for food. If an ant finds food it goes back to the nest and deposits trail pheromone on the path back. The pheromone then attracts other ants to use the same path and to increase the amount of pheromones until almost all foraging ants use the path. But a small amount of ants do not stay on the line which gives them the opportunity to find new food sources or better paths to the source. For the same reason the evaporation of pheromones is important. Without evaporation it is not possible to optimize an existing path because it would not be possible to "forget" a suboptimal one. The optimization of the path length works because shorter paths receive their pheromone reinforcement earlier and more frequently than longer paths [7, 3].

This model has been verified in several experiments, a famous one is the double bridge experiment by Deneubourg et al. [13, 6].

2.2 Ant Colony Optimization (ACO)

Ant colony optimization algorithms [11, 12] are stochastic meta heuristics based on the foraging strategies of trail laying ants, in which simple agents combine solution components to a feasible solution. Agents communicate only indirectly through virtual pheromones which influence the selection process during the solution construction. Virtual pheromones are quantities which are associated to a point in the solution construction process and a solution component and they reflect the probability to choose a specific component as next if an agent (ant) reaches the associated point.

A common ground of all ACO algorithms are virtual ants searching for a solution and the pheromone update phase. Ants are produced and started at several points within the exploration space. The exploration space is the set of all available solution components. Start points can be a node in the city-graph for a traveling salesman problem or the source node if the task is to find a route to the destination through a communication network. The ant begins to stochastically combine solution components until it reaches a solution or it gets killed if it is not possible to reach a valid solution anymore. The probability of choosing a specific solution component as the next step is proportional to the pheromone value associated with the component and the current location of the ant. In many ACO algorithms heuristic information is additionally used to influence the selection process. The distance between the cities can be used as heuristic information in traveling salesman problems to bias the ants to prefer closer cities.

When an ant has reached a solution, the pheromone update process starts. Depending on the algorithm every ant or just a few are allowed to update the pheromone values of their path. In the majority of the algorithms the amount of the pheromone update depends on the quality of the solution. Another important part of the update process is evaporation of pheromones: Continuous and uniform degradation of pheromones intensity with time reduces the risk of fast convergence to a local optimum.

The different algorithms mainly vary in the way solution quality influences the pheromone updates, how heuristic information influences the selection process and which ants are allowed to update pheromone values. Additionally a lot of problem-specific optimizations have been developed on top of the common base.

2.3 ACO Applied to Network Routing

If ant colony optimization strategies are applied to network routing, structural changes to the generic description above are necessary.

Because of the decentralized nature of the routing problem it is not possible to maintain a central pheromone table. Therefore each node maintains its own pheromone table which effects the pheromone update process. There are two sorts of ants used in the ACO algorithms for network routing:

Forward ants are started at the source node to find a path to the destination, memorizing the path they used. If a forward ant reaches the destination it is converted to a backward ant.

Backward ants travel back along the memorized path and update the pheromone tables of the visited nodes.

2.3.1 AntNet

AntNet [12] is a routing algorithm for wired networks based on ant colony optimization.

The AntNet Algorithm

Artificial ants are produced on each network node at regular intervals and travel concurrently with the traffic through the network to the destination d which is chosen according to the traffic distribution of the source node s:

$$p_{sd} = \frac{f_{sd}}{\sum_{i=1}^{n} f_{si}},$$

where f_{sd} is a measure of the data flow $s \rightarrow d$ and n is the number of known destinations.

During the tour to the destination the ant collects information about the duration of the travel, the congestion status and the identifiers of the nodes within the path.

On the way back it updates the local statistical model of the network status and the local pheromone concentration as a function of the quality of the followed path. The statistical model is used to evaluate the quality of a path and to determine the amount of pheromones that should be deposited. The role of the pheromones is to mark preferable paths and to guide the following ants. Once the artificial ant comes back to its source node, it is deleted.

Pheromones

On each network node i the pheromone matrix T_i stores the pheromone values τ_{ijd} for each neighbor node and destination combination. The element τ_{ijd} represents the learned attractiveness for a forward ant on node i to choose neighbor node j if it has to travel to the destination d.

In the AntNet Algorithm the pheromone values of one node with the same destination are normalized to 1.

$$\sum_{j \in N_i} \tau_{ijd} = 1 \qquad d \in [1, n] \text{ and } \forall i,$$

where N_i is the set of neighbors of node i, and n is the number of nodes in the network.
The pheromone values are updated by the backward ants during the travel back to the source node. The pheromone value corresponding to the link which was chosen by the forward ant is incremented and all other nodes are decremented by normalization:

$$\tau_{ifd} \leftarrow \tau_{ifd} + r(1 - \tau_{ifd}) \tag{2.1}$$
$$\tau_{ijd} \leftarrow \tau_{ijd} - r\tau_{ijd}, \quad j \in N_i, j \neq f, \tag{2.2}$$

where f is the node chosen by the forward ant when it was on node i. The reinforcement r is computed by taking into account the quality of the path found by the forward ant.

Statistical Model

In AntNet each node i maintains a simple parametric statistical model M_i of the traffic which is used to evaluate the quality of a path found by an ant. The model helps to cope with the dynamic nature of the network routing problem. The quality of a path depends on the structure of the network and the traffic within the network and both characteristics can change over time.

The adaptive local model $M_i(\mu_{id}, \sigma^2_{id}, W_{id})$ reflects the travel time from node i to destination d gathered by the artificial ants and is defined through the sample mean μ_{id}, sample variance σ^2_{id} and the best time $W_{best_{id}}$ within the moving observation window W_{id}. The statistical model is updated using information stored in the ant's memory. AntNet uses the following exponential models to calculate the sample mean and variance:

$$\begin{aligned} \mu_{id} &\leftarrow \mu_{id} + \varsigma(o_{i\to d} - \mu_{id}), \\ \sigma^2_{id} &\leftarrow \sigma^2_{id} + \varsigma((o_{i\to d} - \mu_{id})^2 - \sigma^2_{id}), \end{aligned} \tag{2.3}$$

where $o_{i\to d}$ is the new trip time from node i to destination d experienced by the ant. The factor ς weights the new sample and influences the number of recent samples which are used to estimate the average. After l samplings the weight of the k-th ($l > k$) is: $\varsigma(1-\varsigma)^{j-k}$. Therefore the number of samples effectively used to calculate the estimation is approximately $5/\varsigma$.

The best trip time from node i to destination d observed in the last w samples is stored in W_{id}. Therefore it represents a short time memory expressing an estimation for the best trip time. After each sample, the length of the window w is incremented modulus w_{max}. w_{max} is the maximal allowed length of the observation window and is set to $w_{max} = 5c/\varsigma$, with $c \leq 1$. If $c = 1$ then the short term memory value $W_{best_{id}}$ and the long term exponential mean refer to the same set of samples.

Pheromone Reinforcement

The reinforcement value r is a crucial quantity for the quality of the algorithm. The following aspects should be taken into account for the calculation:

1. The pheromone increment for a path should be proportional to the path quality.
2. The path quality is a relative measure depending on the traffic condition and it can be estimated by means of the statistical model M_i.
3. It is important not to follow all traffic fluctuation, because uncontrolled oscillation in the routing tables is one of the main problems of shortest-path routing algorithms.

The simplest way to update the pheromone values is to set r = constant. This approach works because of the fact that ants traveling on shorter paths will arrive in higher rates and therefore these paths get more pheromone increments.

A more sophisticated approach is to grade the trip time T of an ant with the help of the statistical model M_i. One possible way is:

$$r = c_1\left(\frac{W_{best}}{t}\right) + c_2\left(\frac{I_{sup} - I_{inf}}{(I_{sup} - I_{inf}) + (t - I_{inf})}\right) \tag{2.4}$$

I_{sup} and I_{inf} are estimates of the limits of an approximated confidence interval for μ. I_{inf} is set to W_{best} and $I_{sup} = \mu + z(\eta/\sqrt{w})$, with $z = 1/\sqrt{1-v}$ where v determines the selected confidence level. While the first term $\frac{W_{best}}{t}$ evaluates the ratio between the best trip time in the observation window and that one experienced by the ant, the second term $\frac{I_{sup}-I_{inf}}{(I_{sup}-I_{inf})+(t-I_{inf})}$ evaluates the distance between t and I_{inf} in relation to the size of the confidence interval and therefore consider the stability of the past samples. If $t = I_{sup} = I_{inf}$ the denominator would be zero, in this case the whole term is set to zero. The coefficients c_1 and c_2 are weighting parameters to determine the influence of each term.

To let the system be more sensitive in rewarding good paths the value r obtained in equation (2.4) is transformed by means of a squash function $s(x)$:

$$r = \frac{s(r)}{s(1)}, \tag{2.5}$$

where

$$s(x) = \left(1 + \exp\left(\frac{a}{x|N_i|}\right)\right)^{-1} \tag{2.6}$$

Choosing the Next Node

A forward ant travels from node to node until it reaches its destination. On a node the following procedure is used to choose the next node:

1. The next node is chosen among all neighbors the ant has not already visited, or among all nodes if there is no node not already visited. The probability to choose the neighbor j is P_{ijd} and it is computed as the normalized sum of the pheromone τ_{ijd} and heuristic value η_{ij} which reflects the length of the queue of the link from i to j:

$$P_{ijd} = \frac{\tau_{ijd} + \alpha\eta_{ij}}{1 + \alpha(|N_i| - 1)}, \tag{2.7}$$

   The value α is used to weight the influence of the heuristic information in respect to the pheromone values. The heuristic value η_{ij} is normalized to $[0, 1]$.

$$\eta_{ij} = 1 - \frac{q_{ij}}{\sum_{l=1}^{|N_i|} q_{ij}}, \tag{2.8}$$

   where q_{ij} is the queue length in bits waiting to be sent. The heuristic information enables the algorithm to react on the current traffic situation and therefore to be more reactive.

2. If a cycle in the path is detected, the nodes within the cycle and all corresponding information is deleted from the memory of the ant. If

the cycle is longer than half of the length of the path already traveled, the ant is deleted.

3. A forward ant is also removed if its life time rises above the value of the maximum life time parameter.

2.3.2 AntHocNet

AntHocNet [8] is a routing algorithm for wireless ad hoc networks based on the ant colony optimization principle.

The Algorithm

The first time a source node sends a packet to a new destination forward ants are produced to find paths to the destination node. After this reactive ant production which builds a good path, the proactive phase follows in which ants are constantly generated to optimize and adapt the path to changed traffic situation or topology changes. During the proactive phase an ant is started after every n-th data packet.

Packets and forward ants are stochastically routed, the probability to choose the next node n and the way to destination d is P_{nd}:

$$P_{nd} = \frac{(T^i_{nd})^\beta}{\sum_{j \in N^i_d} (T^i_{jd})^\beta}, \quad \beta \geq 1,$$

where T^i_{nd} is the pheromone value at node i associated to the next hop n and destination d. N^i_d is the set of neighbors over which the destination can be reached. β is a parameter to control the exploratory behavior. It is kept low (set to 1) for ants and set to 2 for packets. In this way ants tend to explore the surrounding of good routes to find even better ones and packets prefer the best known routes.

Forward Ants

If a source sets up a new path, it broadcasts forward ants to all its neighbors. All ants which originate from the same forward ant belong to an ant generation. If a forward ant reaches a node with routing information for its destination it is unicasted to the next node accordingly to the pheromone concentration. If the node does not have routing information, the ant is again broadcasted to all neighbors.

When a ant reaches a node which was already visited by an ant of the same generation it is forwarded only if the number of hops and the travel time is within an acceptance factor a_1 of the values of the best ant of that generation. This strategy limits the overhead which would result out of uncontrolled broadcasts, but it also limits the diversity of the paths. Therefore

a second and higher acceptance factor a_2 is introduced, it is used if the first hop is different from those taken by the previously accepted ants.

With a low probability, proactively sent ants are broadcasted at a node even if there is routing information at this node. To avoid that broadcasted proactive ants flood the network, they are killed if they do not find a node with routing information within n_b hops.

Additionally, hello messages are used to improve the performance of forward ants. By sending hello message to all neighbors a node induce the production of pheromone information at its neighbors. This enables reactively started forward ants to find destination nodes easier. Secondarily, the hello messages allow the detection of broken links, because when a node receives a hello message for the first time it expects to regularly receive other messages from the node. If a defined number of hello messages are missing the link to the distinct neighbor is determined as broken.

Pheromone Update

The backward ants incrementally estimate the travel time to the destination. $\hat{t}_d^i$ is the sum of the local estimation $\hat{t}_{i+1}^i$ for the hop from i to $i+1$; it is used to calculate the pheromone updates:

$$\hat{t}_d^i = \sum_{i=0}^{n-1} \hat{t}_{i+1}^i \tag{2.9}$$

To calculate the local estimation the average time to send one packet $\hat{t}_{mac}^i$ and the current number of packets in the send queue q_{mac}^i is used:

$$\hat{t}_{i+1}^i = (q_{mac}^i + 1)\hat{t}_{mac}^i \tag{2.10}$$

$\hat{t}_{mac}^i$ is calculated as running average of the duration t_{mac}^i between the time a packet is ready to send and the end of the transmission:

$$\hat{t}_{mac}^i = \alpha \hat{t}_{mac}^i + (1-\alpha) t_{mac}^i \tag{2.11}$$

The estimated travel time $\hat{t}_{i+1}^i$ is used to update the entry T_{nd}^i in the pheromone table T^i. If there is no entry in the table to update a new one is created by the backward ant. The pheromone values represent a running average of the inverse cost of a path in terms of estimated time an number of hops. The amount of the pheromone update is calculated accordingly to:

$$\tau_d^i = \left(\frac{\hat{t}_d^i + h t_{hop}}{2} \right)^{-1}, \tag{2.12}$$

where h is the number of hops and t_{hop} is a parameter representing the time for one hop in the unloaded network. The update of the pheromone value T_{nd}^i is calculated using a running average with parameter γ,

$$T_{nd}^i = \gamma T_{nd}^i + (1-\gamma)\tau_d^i, \quad \gamma \in [0,1] \tag{2.13}$$

Chapter 3

Fraglets

3.1 Artificial Chemistry

Many natural chemical processes can be seen as information processing e.g.:

- Chemical reaction networks controlling the movement of bacteria
- Gene transcription and translation
- Mutation and recombination
- Immune system
- Endocrine system

To understand this processes which build the ground for life, scientists have begun to study, model and simulate chemical environments. There are two main approaches to work on chemical computing: In the field of real chemical computing real molecules are used to perform computing tasks. Examples therefore are DNA and peptide computing.

The other approach occurs within the fields of Artificial Chemistries, where chemical metaphors are used as design paradigms for new hardware and software architectures [9]. An artificial chemistry is formally defined as a triple (S, R, A) where S is the set of all possible molecules, R is the set of collision rules, representing the interaction among the molecules and A is an algorithm describing the reaction vessel or domain and how the rules are applied to the molecules inside the vessel [10].

Artificial chemistry frameworks can be used to model systems in different domains e.g.:Biological, evolutionary social or parallel processing systems. Examples for artificial chemistry systems are the chemical abstract machine (CHAM) [4], Gamma [2] and P systems [17].

3.2 Fraglets

The fraglets language is a software environment oriented to chemical metaphors which was created around 2001 by Christian Tschudin [18]. Tiny computation fragments called fraglets react with each other or run through transformation within an reaction vessel. In addition fraglets can be sent over a network to other nodes containing reaction vessels. Because a fraglet contains code and data in an arbitrarily selectable ratio, the framework can be used to implement classical network as well a active network approaches or any flavor in between. The main purpose of the fraglets language are automated network protocol synthesis and evolution [15]. Due to its designation for networking the instruction set was originally designed for efficient packet processing with constant (short) instruction processing time. However this limitation was softened through the later introduction of more complex instructions.

3.3 Programming in Fraglets

In the fraglets environment there is no deterministic program flow in general. If many fraglets are in the same reaction vessel, one is randomly picked and its first element is processed if this is possible. The result of the operation is usually reinjected into the same vessel. Exceptions of this rule are operations that destroy fraglets or send the result to another node/subnode.

In the fraglets language there are two instruction categories: reactions (the different forms of `match`) and transformations (the rest). While reactions combine two fraglets, transformations alter one fraglet.

In the following examples an arrow (`-->`) represents a chemical reaction in fraglets, consuming the reactants on the left side, and producing the product fraglet on the right side.

Table 3.1 explains the function and structure of the instructions used in the examples. In all code samples elements of the form `$X` are placeholders for values and they mainly represent numerical values or single elements. The two exceptions are: `$TAIL` which stands for an arbitrary rest of a fraglet and `$DSU` which stands for *do something useful* and represents an arbitrary instruction sequence. The placeholders are not part of the fraglets language, they are used for illustration purpose only.

Example: $y = x + 24$

```
[x 18] + [match x sum y 24]
-->  [sum y 18 24]
-->  [y 42]
```

In this example the fraglet `[x 18]` is the input and `[y 42]` is the output of the program.

Input --> Result
Combination of two fraglets: `match` `[match m0 $TAIL0] + [m0 $TAIL1] --> [$TAIL0 $TAIL1]` `m0` is a tag which is an arbitrary alphanumerical element starting with a letter, used to mark fraglets.

Calculation: `sum`, `mult`
`[sum $E $X $Y $TAIL] --> [$E $X+Y $TAIL]`
`[mult $E $X $Y $TAIL] --> [$E $X*Y $TAIL]`
Other arithmetic operations which have the same structure are: `diff` for difference, `div` for division, and `pow` for power of.

Conditional branch: `eq`, `lt` `[eq $T $F $X $Y $TAIL] --> [$T $X $Y $TAIL]`, (`$X==$Y`) `[eq $T $F $X $Y $TAIL] --> [$F $X $Y $TAIL]`, (`$X!=$Y`) `[lt $T $F $X $Y $TAIL] --> [$T $X $Y $TAIL]`, (`$X<$Y`) `[lt $T $F $X $Y $TAIL] --> [$f $X $Y $TAIL]`, (`$X>=$Y`)

Other operations: `fork`, `nul`, `nop`
`[fork $H0 $H1 $TAIL] --> [$H0 $TAIL] + [$H1 $TAIL]`
`[nul $TAIL] --> []`
`[nop $TAIL] --> [$TAIL]`

Table 3.1: Fraglet instructions used in examples.

Sequences: There are two possible ways to perform operations in a distinct order on a set of data. The first one uses specific marker elements at the first position of a fraglet to tag the intermediate results. A tag fraglet can then react with the next operation which produces another intermediate result with a new marker.

Example: $y = 3 * (x + 2)$

```
[x 17] + [match x sum r1 2]
--> [sum r1 2 17]
--> [r1 19]

[r1 19] + [match r1 mult y 3]
--> [mult y 3 19]
--> [y 57]
```

The other way to determine the order of a computation is to use the recently introduced stack operations which manipulate the tail of a fraglet instead of its head. In the current version of the fraglets interpreter most of the fraglets instructions have stack equivalents. They are specified by the

letter *s* in front of the operation name. Table 3.2 shows the structure of the stack instructions used in the examples.

Input `-->` Result

Calculation: `ssum`, `smult`
`[ssum $TAIL $X $Y] --> [$TAIL $X+Y]`
`[smult $TAIL $X $Y] --> [$TAIL $X*Y]`
Other arithmetic operations which have the same structure are:
`sdiff` for difference, `sdiv` for division, `spow` for power of, `smin` and `smax`.

Comparison: `eq`, `lt` `[seq $TAIL $X $Y] --> [$TAIL 1]`, (`$X==$Y`) `[seq $TAIL $X $Y] --> [$TAIL 0]`, (`$X!=$Y`) `[slt $TAIL $X $Y] --> [$TAIL 1]`, (`$X<$Y`) `[slt $TAIL $X $Y] --> [$TAIL 0]`, (`$X>=$Y`)
Conditional branch: `sif` `[sif $T $F $TAIL $X] --> [$T $TAIL]`, (`$X!=0`) `[sif $T $F $TAIL $X] --> [$F $TAIL]`, (`$X==0`)

Fraglet manipulation: `sdup, sdel, spush`
`[sdup _ $TAIL $X] --> [$TAIL $X $X]`
`[sdel $TAIL $X] --> [$TAIL]`
`[spush $X $TAIL] --> [$TAIL $X]`

Table 3.2: Stack instructions used in examples.

Example: $y = 3 * (x + 2)$ with stack operations

```
[x 17] + [match x ssum smult y 3 2]
--> [ssum smult y 3 2 17]
--> [smult y 3 19]
--> [y 57]
```

For longer sequences stack operations are more convenient because the whole processing can be written in one fraglet. If all calculations are done with stack operations the fraglets can be seen as virtually split in two halves. The front part represents the program and the tail represents the data stack.

Conditions: For conditional branches the fraglets language has the instructions `eq` which stands for *it equal* and `lt` which stands for *if less than* (table 3.1). The stack equivalents `seq`/`slt` are comparison instructions and `sif` is used for conditional branches (table 3.2).

Example: *If* $x < 10$ *do $DSU0 else $DSU1.* For x=3 we have:

```
[slt sif yes no 3 10] + [match yes $DSU0] + [match no  $DSU1]
--> [sif yes no 1]
--> [yes ]

[yes] + [match yes $DSU0]
--> [$DSU0]
```

Repetitions: Although the language has no specific loop construct, it is also possible to build loop-like structures with fraglets. The following two methods are used in the implementations of the ant inspired routing algorithms described in section 5. The first one is similar to the *for each* statement of other languages and in the implementation it is used to process elements of list fraglets. For this program structure the following four fraglets are needed:

Initialization The first fraglet binds to the list, adds the element counter which stores the position of the next list element and tags the resulting fraglet for the next step: check and increment.

Check and increment This fraglet first checks if the end of the list is reached. If there are list elements left to process the element counter is incremented and the resulting fraglet is tagged for the element processing step. If the end is reached the resulting fraglet is tagged for finalization.

Element processing With this fraglet any element processing operations can be done. At the end of this step the remaining fraglet must have its original structure (list with a counter) and is tagged for check and increment.

Finalization The last fraglet removes the element counter and restores the original tag of the element list.

With the second method which is based on quines a *for*-loop can be emulated. A quine is a program that outputs its complete source code. In the fraglets environment quines are useful because normally the program is consumed during the execution and quines have the ability to reproduce themselves.

A simple quine in fraglets consists of two fraglets: the blueprint and the active code.

Example: Quine

```
[bp match bp fork nop bp]  // blueprint
 + [match bp fork bp nop]  // active code

--> [fork bp nop match bp fork nop bp]
--> [bp match bp fork nop bp]    // blueprint is restored
    + [nop match bp fork nop bp]
--> [match bp fork nop bp]       // active code is restored
```

This trivial quine can easily be equipped with useful operations. Thereto any command sequence can be inserted into the blueprint right behind the front tag. It is not necessary to alter the active code [19].

To emulate a *for*-loop a counter and a break condition is added to a quine. The resulting counting quines have an additional step inserted which is executed right after the match of blueprint and the active code. Within that step the loop counter at the end of the blueprint is checked. If the break condition is met, the counting quine deletes itself. If not then the counter is decreased and the blueprint plus the active code is reproduced.

Example: Counting quine
`$C` is the counter and it is initialized to the number of iterations.

```
[match bp sdup _ sif spush nul 1 sdiff fork bp nop]
 + [bp $DSU sdel match bp sdup _ sif spush nul 1
    sdiff fork bp nop $C]

--> [sdup _ sif spush nul 1 sdiff fork bp nop $DSU ... $C]
--> [sif spush nul 1 sdiff fork bp nop ... $C $C]

If $C != 0
--> [spush 1 sdiff fork bp nop $DSU ... $C]
--> [sdiff fork bp nop $DSU ... $C 1]
--> [fork bp nop $DSU ... $C-1]
--> [bp $DSU ... $C-1]
    // reproduction of the blueprint with a decreased counter
    + [nop $DSU sdel match bp ... $C-1]
--> [$DSU sdel match bp ... $C-1]
--> [sdel match bp ... $C-1]
--> [match bp sdup _ sif spush nul 1 sdiff fork bp nop]
    // reproduction of the active code

If $C == 0
[nul 1 sdiff fork bp nop $DSU ... $C]
--> []   // end of the reproduction
```

Concentrations: Besides the explicit data within a fraglet also the concentration of distinct fraglets can represent information e.g. the result of a calculation or the input to processes. In the implementation of ant inspired algorithm the concentration of neighbor node information represents the pheromone values and counting quines are used to transfer the explicitly calculated path quality into concentration changes.

Chapter 4

Integration into OMNeT++

4.1 OMNeT++

OMNeT++ [16] is a component based discrete event simulation system with strong GUI support. Its main application is the simulation of communication networks, but because of its flexibility it can be used for simulations in many other fields such as IT systems, queuing networks, hardware architectures and business processes. Another excellence is its easy expandability as a result of the modular architecture and the open source.

To different types of modules are used to build up OMNeT++ simulations:

Simple modules contain the functionality which is programmed in C++.

Compound modules are used to group simple and other compound modules to one entity. Thereby compound modules do not contain program code, but they are defined by configuration files which specify the connections between the submodules.

Both module types can have an arbitrary number of input and output gates by which a modules are connected to each other.

4.2 Existing Integration

There was an integration of fraglets into OMNeT++ already done by the BIONETS simulator project [5]. Initially it was the idea to extend this integration, but a closer inspection showed three shortcomings:

1. The synchronization of the time within the fraglets reaction vessel with the simulation time is not implemented in a reasonable manner. As the temporal devolution is crucial for the success of an implementation in fraglets, this is the major deficit of the existing integration. This

applies especially to a routing protocol where several network nodes have to work together.

2. There was a bug in the integration that leads to a crash of the simulation if a fraglets reaction vessel is inspected and if there are multiple instances of the same fraglet.

3. There is no structure that helps dividing the parts that are used for the integration of fraglets and the parts that are used for a specific simulation.

4. The integrated fraglets version is already out-dated and an update would have been necessary.

These problems lead to the decision to restart the integration.

4.3 Architecture of the New Integration

The integration of fraglets into OMNeT++ is a simple straightforward process and it was done according to the structure of the current fraglets interpreter. The result was the OMNeT++ module called `FragletSoup` described in one of the following sections. A couple of supporting modules are needed to enable the fraglets soup to be used in a network simulation:

- Measurement modules
- Fraglet and packet source modules
- Queue module
- Packet distribution module

4.3.1 FragletNode

The compound module `FragletNode` (figure 4.1) combines all simple modules which are needed to simulate a fraglets processing network node. It has a variable number of incoming and outgoing gates to other `FragletNode`s. Internally each incoming connection is connected to the `FragletSoup` through a port specific measurement module (`ThroughputMeasure`) for each incoming line. The outgoing gate of the `FragletSoup` is connected to the outgoing gate of the `FragletNode` through the `Distributor` and a specific `Queue` for each outgoing gate. Data packets which reached their destinations are handled by the `Sink` which is connected to an outgoing gate of the `FragletSoup`. Additionally, up to five source modules can produce new fraglets and inject them directly to the source. The maximum number of sources is chosen arbitrarily and it could be easily altered.

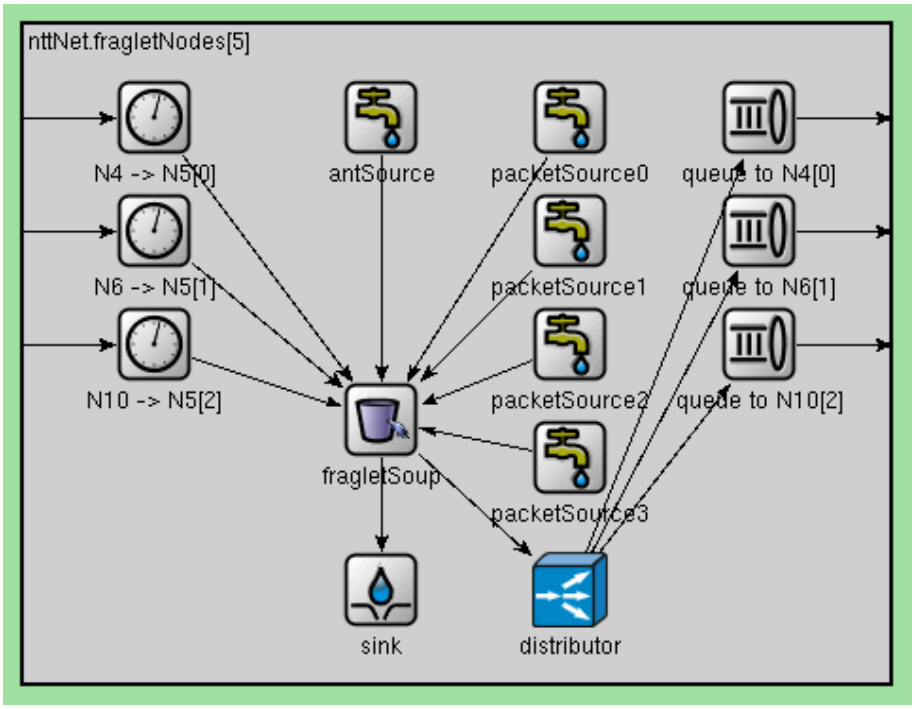


Figure 4.1: Structure of a `FragletNode`

4.3.2 FragletSoup

The simple module `FragletSoup` is responsible for initializing and running the fraglets environment. Through a variable number of incoming gates fraglets are injected into the soup. The injected fraglets can originate either from other `FragletNode` or from source modules within the enclosing `FragletNode`.

The reaction within the vessel can be synchronized to the simulation in two different ways. The first method executes all pending reactions and transformations within one single OMNeT++ event. The second synchronizes the fraglets vessel time with the OMNeT++ simulation time, the relation of both times can be adjusted by a arbitrary factor. While the first version is slightly more efficient, the second method is viable for protocols in which the dynamic behavior of reactions based on concentrations is used.

With the current fraglets interpreter version it is problematic to synchronize execution to simulation time if timers in fraglets are used. This is because the expiration of a timer is not predictable form the outside, that makes it necessary to alter the fraglets interpreter code and to send an event to the OMNeT++ simulation to correctly handle fraglets timer. One target during the integration is not to alter the fraglets interpreter source at all because that makes updates of the interpreter possible with only minor

adaptions of the integration code. Therefore fraglets timers can not be used, but this is not a severe restriction because OMNeT++ timers can be used instead. This limitation will disappear as the implementation of timers in fraglets will change in future versions.

4.3.3 Distributor

The `Distributor` gets packets from the `FragletSoup` and delivers them to the appropriate queue. The send command which was used to send the message (`osend`/`sosend`) determines if the first or the last element is used to choose the corresponding output port. The `Distributor` additionally supports anycast (`any`) and broadcast (`all`).

4.3.4 Queue

The module `Queue` delivers the functionality of a send queue with configurable queue length. Furthermore it monitors dropped packets and the queue length.

4.3.5 ThroughputMeasure

The module `ThroughputMeasure` measures the incoming amount of data per time. The time resolution is configurable.

4.3.6 Sink

The task of the module `Sink` is to handle the data packets that reached their destination and thereby to simulate an application. It reads out the values of the report message and captures them for the local statistic. A report message can be generated from within the fraglets soup with the new fraglets instruction `osink` (see 4.3.8). The sink registers delay and hops for every packet separately for every source node.

4.3.7 Measurement

In each simulation one module is responsible to collect all globally measured data. It registers the following measurements as a time vector:

- Average delay during the last observation period
- Average hops during the last observation period
- Average relative hops for the last observation period (hops / hops of the shortest path)
- Average dropped bits and packets per second (due to full queues)

- Average lost bits and packets per second (due to disabled nodes)
- Received data bits and packets per second
- Sent data bits and packets per second

and the following measurements as scalars:

- Total number of data packets and bits sent
- Total number of data packets and bits received
- Total number of data packets and bits dropped
- Total number of data packets and bits lost
- Average delay, hops and relative hops over the whole simulation

4.3.8 New Fraglets Instructions for OMNeT++

The following instructions are added to the fraglets instruction set to program within the OMNeT++ environment:

`onode/sonode` retrieve the address of the `FragletSoup` which is equal to the address configured for the `FragletNode`. (Similar to the original fraglets instruction `node/snode`)

`osend/sosend` pass a fraglet to the `Distributor` which sends it as a message to another `FragletNode`. The size of the OMNeT++ message corresponds to the length in bits of the fraglet. (Similar to the original fraglets instruction `send/osend`)

`osendws` works like osend but uses the first element of the fraglet as message size.

`otime/sotime` insert the omnet simulation time into the fraglet.

```
[sotime $X $T @TAIL] --> [$T @TAIL simTime()]
[otime $T @TAIL] --> [$T simTime() @TAIL]
```

`osink` produces a report message and passes it to the `Sink`.

```
[osink SOURCE HOPS SIZE TIME DEST @TAIL] --> []
```

Chapter 5

Implementation of ACO-Based Routing in Fraglets

Since AntNet is a highly optimized ants routing algorithm it is used as a base for the implementation in fraglets. But some aspects of AntNet do not map well to the fraglets environment. Therefore some adaptions were necessary to build the first working version: The basic implementation.

Additionally the basic version was extended such that it can cope with network topology changes which was left out in the basic version for the sake of simplicity.

With a third version a way to optimize the path finding capability of the ants was studied.

5.1 Implementation of AntNet

5.1.1 Node Structure

Figure 5.1 show the inner structure of the main reaction vessels which is created in the `FragletSoup` module of every network node. The main vessel contains all processing rules (program repository), several different subnodes and two list fraglets. The first one is a list of all neighbors. It is built up during network initialization and it is used for the creation of statistic fraglets. The second one is a list of all known destinations which is used to check if an destination is already known and therefore a subnode must exist.

A destination is added when a forward ant with an unknown destination visits a node. Besides the extension of the destination list such an ant induces the creation of a new subnode for the specific destination. The destination-specific subnodes are the equivalent of the pheromone table of the classical approaches and each subnode contains the information of one

row (the pheromone information for one destination). The concentration of neighbor node information within such a subnode represents the pheromone values. A new subnode for a destination is initialized with a uniform concentration of all known neighbors.

Additionally there exists a subnode called `destination` whose concentration represents the distribution of the data packet destination launched by this node. This subnode is used to produce forward ants according to the destination distribution of the packet flow.

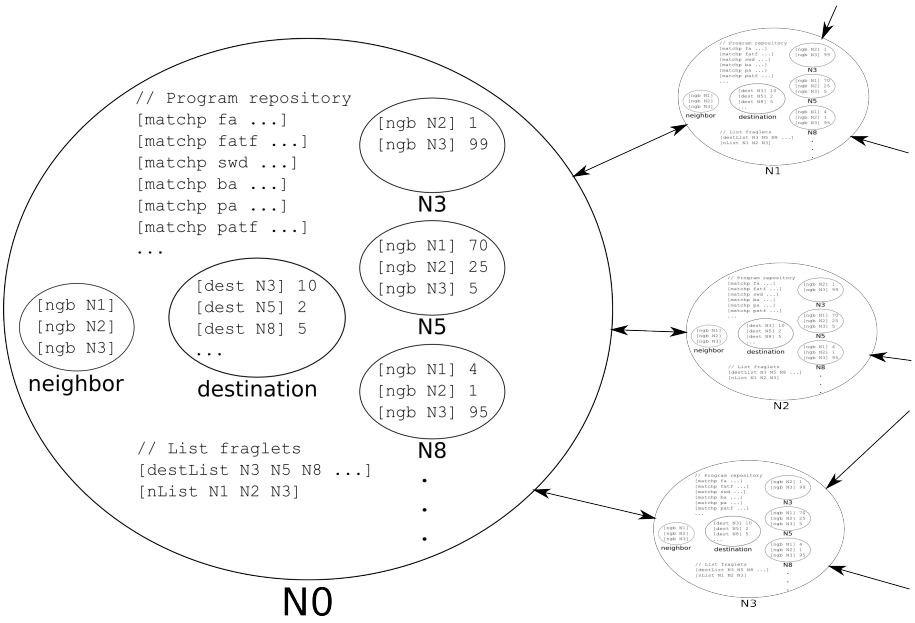


Figure 5.1: Inner structure of a network node

5.1.2 Initialization

Before the first forward ant can be started all nodes need an initialization phase. During this phase all nodes broadcast their address to their neighbors and every node builds up the subnode `neighbor` with one neighbor information fraglet `[ngb ADDRESS]` for each neighbor of the node. This subnode is used to initialize the concentration of neighbor information in the subnodes for the destinations.

Additionally to the `neighbor` subnode a neighbor list fraglet `[nList NEIGHBOR1 NEIGHBOR2 ...]` is build up during the initialization. This list is used to create a fraglet for the statistical model for each neighbor in a new destination subnode.

5.1.3 Start of a Forward Ant

The OMNeT++ module called `FragletSource` is used to create new forward ants in regular intervals. A creation fraglet is injected into the subnode `destination` where the destination of the ant is determined. Afterwards a new fraglet is expelled to the main node and handled as a forward ant.

5.1.4 Forward Ant

Ants need several information to fulfill their task. Therefore a forward ant stores the node id and the time at each node. This information is used by the backward ant to find the way back and for the pheromone update. The structure of a forward ant is illustrated in figure 5.2

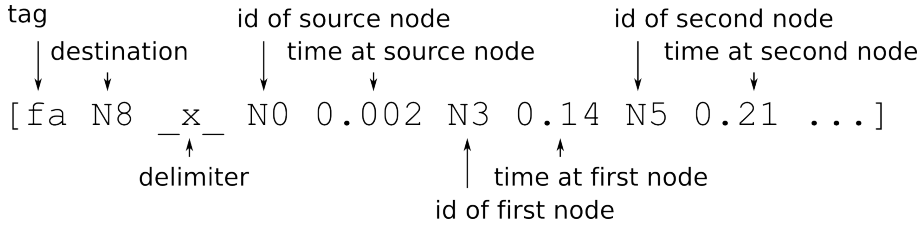


Figure 5.2: Structure of a forward ant

Figure 5.3 shows the processing of a forward ant (`fa`) within a network node. When a forward ant arrives at a new node it is first checked if it has already reached the destination. If it has, then the ant is marked to switch the direction (`swd`) and it is sent back to the last visited node as a backward ant.

If the ant has not reached the destination its path is checked for cycles which is initialized by the tag `ccInit`. The cycle check goes through the whole path and tests if the ant has already visited the actual node. The procedure is done by two fraglets, the first increments the counter that points to the current position within the path and checks if the end is reached. The second checks if the current node in the path is equal to the actual node. If no cycle is detected (the end of the path is reached without finding the current node within the path) the ant is marked for cleanup. This removes the loop counter of the cycle check, adds the actual node and the time and marks the ant as a forward ant to forward to the next node (`fatf`). If a cycle in the path is detected the associated nodes and the corresponding time information are removed from the forward ant. The cleanup procedure and the addition of the node and time is done afterwards as if no cycle was found.

If a forward ant to forward `fatf` appears within a node it is first checked if there already is a subnode for the destination of that ant. This is done by a loop which iterates through the destination list which has the form

`[destList DEST1 DEST2 ...]` and checks if the destination of the ant is already in the list. The five different fraglets which control this behavior have the following functions:

1. Initialize the destination list scan.
2. Increment the loop counter and check if the end of the list is reached. If the end is reached without finding the destination in the list the creation of a new subnode is induced.
3. Check if the current element in the list is equal to the destination, if it is the destination is already known and cleanup is induced.
4. Create a new subnode for the destination and inject a new statistic model fraglet for each neighbor of the node. Additionally, the concentration of neighbor information within the subnode `neighbor` is transferred to the new subnode. This represents the initialization of the routing table. Furthermore the new destination is added to the destination list.
5. Remove the loop counter, recreate the destination list and mark the ant as packet to forward (`patf`).

5.1.5 Routing

When a packet or a forward ant is ready for routing to the next node it is marked with `patf`. The actual routing is done in two steps: first a fraglet is injected into the subnode responsible for the destination and the actual ant or packet is marked as waiting for routing to the destination (`wfr_DEST`). Second the fraglet injected into the destination subnode matches with one of the neighbor fraglet `[ngb NEIGHBOR]` and produces a routing fraglet, which is expelled to the main node. This fraglet matches with the waiting packet/ant and sends it to the next node. The routing of a forward ant is illustrated in figure 5.3.

5.1.6 Backward Ant

When a backward ant arrives at a node it is directly send to the next node in the remaining path. When it has already reached its source node the backward ant is deleted. Additionally, a pheromone fraglet is produced, which has the form `[ph NEIGHBOR DESTINATION TIME_NODE_TO_DESTINATION]`. This information is used to update the local statistical model, calculate the reinforcement and to update the neighbor concentration in the subnode responsible for the specific destination.

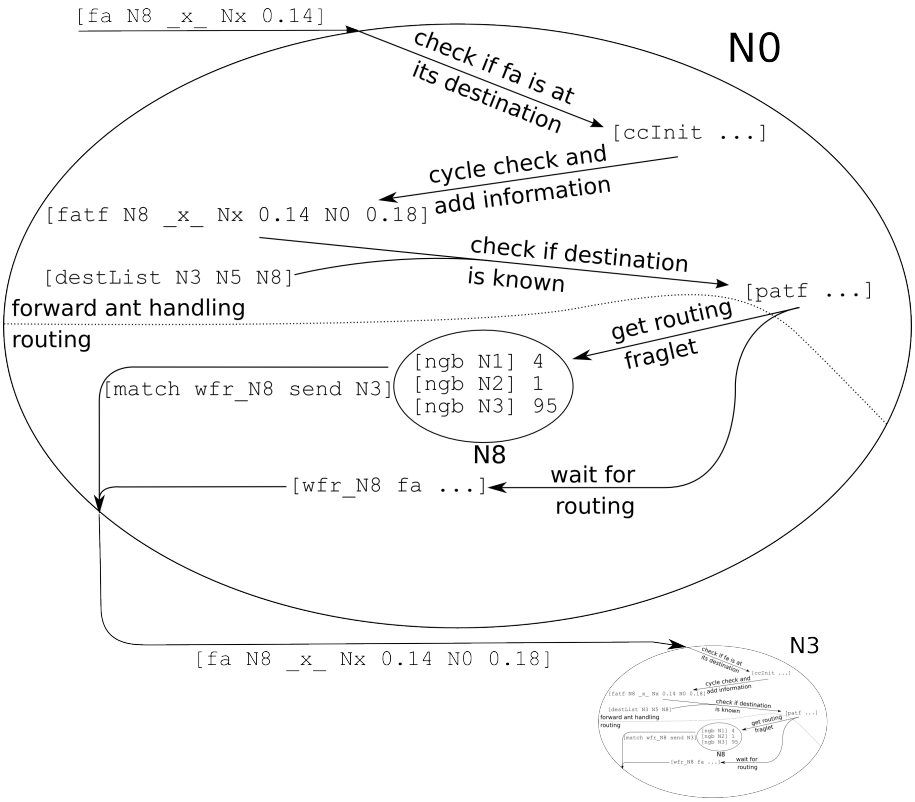


Figure 5.3: Processing and routing of a forward ant within a node

The calculation of the statistical model update is done with one fraglet. Thereto the information of the pheromone fraglet is injected into the subnode for the destination where it matches with the statistical fraglet dedicated to the specific neighbor. The updated model fraglet combined with the time information of the pheromone value is expelled to the main node where it is used to calculate the reinforcement.

Calculation of the reinforcement (according to equation 2.4 and 2.5 in section 2.3.1) is done in the following four steps:

1. The first term $c_1(\frac{W_{best}}{T})$ is calculated.
2. $I_{sup} = \mu + z\sqrt{\eta^2/w}$ is calculated.
3. The reinforcement r is calculated.
4. The reinforcement is squashed: $r \leftarrow \frac{s(r)}{s(1)}$.

After the calculation of r the fraglet for the statistical fraglet is recreated and reinjected into the subnode responsible for the destination. Furthermore the neighbor concentration in the destination subnode is updated in two

steps. First fraglets are produced which randomly delete neighbor fraglets (`[ngb NEIGHBOR]`) and second new neighbor fraglets are produced for the neighbor from which the backward ant has arrived. The amount of the fraglets is determined by the reinforcement and the total number in the subnode. Because the same amount is deleted as it is produced the total number of neighbor fraglets in the subnode remains constant.

5.1.7 Packets

When a data packet arrives at a node it is checked if the packet has reached the destination. If not then it is ready for routing to the next node and thereto it is marked with `patf`.

5.1.8 Difficulties with the Implementation of AntNet

Three aspects of the original AntNet version (section 2.3.1) are not considered in this first implementation. First the inclusion of the heuristic information, the queue length according to equation (2.7) and (2.8) is not implemented. During the implementation this was considered as an optimization of the algorithm but first tests and the following closer study of the algorithm have exposed their importance. The improvement of the adaptivness of the algorithm as described in [12] is thereby not the only aspect of the inclusion of the heuristic information. A vital effect of equation (2.7) is to maintain the exploratory potential of the forward ants even if one pheromone value reaches 100% which happens because of equation (2.4) if $T = W_{best}$ (i.e. the end of the observation window for W_{best} is reached and W_{best} is reset). The introduction of a scale factor for the pheromone reinforcement did not deliver acceptable results.

A complete implementation of the heuristic information in the fraglets environment is not straightforward. There are two possible ways to integrate the information about the queue length: A subnode whose concentration reflects the current queue status, the concentration of this subnode has to be combined with the pheromone subnodes. Because of the necessary actuality of the heuristic information and inert nature of concentrations a high rate of updates of the additional subnodes would be essential which would have a noticeable impact on the needed computation power.

Another possibility would be to inject the heuristic information as pure data fraglet which could be used in the explicit calculation of forwarding probability. But it would make it impossible to use concentrations for the routing and therefore question the usage of a chemical approach.

Second the process which selects the next hop of a forward ant does not regard if a node is already in the path. The implementation of this optimization is also problematic if the routing is based on concentrations. For each forward ant a new subnode must be created with a neighbor concen-

tration according to the pheromone values but without the nodes already visited. An adaptation would be to try to get a neighbor not visited yet and repeat the selection process if the chosen node is already in the path. With a break of the loop after n unsuccessful tries would handle the case when all neighbors are already visited. But as the good results of the second implementation approach show this rather complicated extension is not necessary.

The third aspect of the original AntNet which was not implemented was the pheromone update of subpaths by backward ants described in the section *Data Structure Update* in [12]. It was not implemented because of the estimated complexity of an implementation in fraglets and the expected small performance gain.

5.2 Basic Implementation

With the approach described in section 5.1 the routes found were far away from optimum. Therefore a new update rule was introduced as well as an injection scheme for neighbor information. With these changes it was possible to build an algorithm which produces acceptable results.

5.2.1 Combination of AntNet and AntHocNet Update Rule

Because the pheromone update of the first version seemed to be problematic it was exchanged with an update function similar to the one of AntHocNet (see equation (2.12)). For this only the number of hops is added to the backward ant and the pheromone fraglet and its processing was changed.

In this version the pheromone fraglets contain the travel time and the number of hops used by the forward ant from current node to the destination. This is a slight change of the original AntHocNet approach where the travel time is incrementally estimated and not measured. The parameter called scale factor ϕ is used to adapt the calculated reinforcement to the subnode size and additionally to configure the aggressiveness of the algorithm. With a higher factor less ants are needed to concentrate most of the traffic to one path hence this factor has a similar influence as $1 - \gamma$ in equation (2.13). The new pheromone values are calculated stochastically according to:

$$T^i_{nd} \simeq T^i_{nd}(1 - \frac{\phi\tau}{N}) + \phi\tau,$$

for the pheromone value of the updated path and

$$T^i_{nd} \simeq T^i_{nd}(1 - \frac{\phi\tau}{N}),$$

for the pheromones values of the other paths, where T^i_{nd} is the number of fraglets for a neighbor, τ is the calculated reinforcement and N is the maximal size number of fraglets within a routing subnode.

5.2.2 Reinjection of Neighbor Information

In the first implementation of AntNet it is possible that information of some neighbor node is totally displaced, which means no neighbor fraglet corresponding to this neighbors remains in a routing subnode. In this case neighbors are removed from the exploration space and ants can not choose this neighbor as next node anymore.

To solve this problem neighbor fraglets from the *neighbor* subnode are injected into the routing subnodes. This is done by fraglets injected into the *neighbor* subnode, where they react with one of the neighbor fraglets. The result of this reaction is a fraglet that randomly chooses a routing subnode an inject a neighbor fraglet into it. The selection of the routing subnode is done according to concentrations in the *destination* subnode.

5.3 Extension: Constantly Sent Hello Messages

In our first implementation the algorithm needs an initialization phase at the beginning where all neighbors send their id to each other. This *hello messages* are used to introduce a node to its neighbors. Because this is done only once at the beginning this approach is not able do cope with changes in the network topology. With a small extension this shortcoming can be resolved.

The *hello messages* are sent constantly with a fix rate to the neighbors. At the receiving node the information is stored in the subnode called *neighbor* which have a maximum number of fraglets set.

Due to the equal rate used by all nodes, the concentrations of the different neighbors follow a uniform distribution if the topology of the network does not change. If a node disappears, the information about this node displaced by the others and thereby the node is pushed out of the exploration space of the ants after a while. On the other side new nodes insert their information into the *neighbor* subnodes of the connected nodes and due to the random dilution flow stochastic equilibrium is restored after a while.

The duration of the adaption is determined by the size of the *neighbor* subnode, the rate of *hello messages* and the number of neighbors. A high rate of *hello messages* lowers the reaction time but increases the traffic overhead on the other side. The subnode size should be as small as possible to achieve a reactive setting with low message send rates, but as big as needed to store the neighbor concentrations with a feasible resolution.

Besides the ability to react on topology changes the behavior of the algorithm is not changed by this extension.

5.4 Improvement: Forward Ant Generations

To improve the path finding qualities of the algorithm, multiple forward ants are started simultaneously and only the fastest forward ant of such an ant generation is converted to a backward ant. Thereby only the best ant of a generation can update the pheromone values. The number of ants within a generation is a parameter which configures the improved path finding capabilities as well as the additional traffic overhead.

To reduce the memory consumption of the implementation only the identifier of the last generation is stored and an ant is sent back only if its generation identifier is higher than the stored one. Therefore if the best ant of a generation is faster than the best ant of the previous generation, no ant of the previous generation can update the pheromone trail.

By this improvement the overhead is lower because only the fastest are send back as backward ants, and the exploration capabilities remain the same due to the unchanged number of forward ants. With the same traffic overhead as the basic implementation the forward ant production rate could be increased and thereby the reactiveness would be increased.

Chapter 6

Simulations and Results

To demonstrate the overall performance of the implemented ant routing algorithms a model of a real network is used and with the help of different scenarios with simple networks specific effects of the algorithms are illustrated.

The basic implementation (section 5.2.1) is tested with all scenarios except the *new node appears* (section 6.3.3) scenario, because the first implementation is not able to insert new nodes into the exploration space of the ants.

The extended version with continuously sent *hello messages* (section 5.3) is only tested in the two scenarios simulating topology changes (section 6.3.2 and 6.3.3)). In the other scenarios this implementation produces the same results as the basic implementation due to the unchanged behavior of the algorithm.

The improved version with the forward ant generations (section 5.4) is tested in the *static path quality* scenario (section 6.3.4) where it can demonstrate the improved path finding capabilities and the *oscillation* scenario (section 6.3.5) which illustrates the drawback of the increased aggressiveness.

6.1 Parameters

One advantage of the implemented approach is the small number of parameters in opposition to the original implementation of AntNet which have more then ten parameters to set. Basically, there are two parameters which configure the behavior of the implemented algorithm:

(i) The *scale factor* ϕ determines the aggressiveness of the ants which correspond to the strength to push one path and inhibit the others. The impact of the *scale factor* depends on the total number of neighbor fraglets within the pheromone subnodes. This number is equal in all simulations. First tests have shown that good results are achieved with factors in the

range *[1,10]*. Therefore the values 1, 2.5, 5, 7.5 and 10 are chosen to demonstrate the influence of the *scale factor*. Additionally, two values bellow (0.1 and 0.5) and two above (50 and 80) are used to show what happens if the *scale factor* is out of the feasible range.

(ii) The rate of randomly injected neighbor fraglets into the pheromone subnodes determines the explorative behavior of the ants. Because pheromone updates done by ants inhibit this effect, the ratio of forward ants to injected neighbor fraglets is studied in the different simulations. The following ratios are used: 2:1, 1:1, 1:2 and 1:4. Thereby an ratio of 1:2 means one forward ant per reinjection.

The other parameter have only minor impact on the algorithm behavior. These parameters are:

- The production rate of new forward ants is set to 10 ants per second. Variations of the rate change the reaction tempo and the routing overhead, but they do not influence the path selection behavior of the algorithm. Therefore only one setting of the parameter is studied.
- The maximal number of neighbor fraglets within pheromone subnodes is set to 100 fraglets. This parameter does not change the behavior of the algorithm except if it is set too low which would lower the resolution of the pheromone concentration under a required limit. An increasing of the size would not have a negative effect besides the additional computational effort for the higher pheromone variations.
- The estimated time of one hop in unloaded condition (parameter h in equation (2.12)) is set to the average delivery time for a packet in the used network.

Hello Messages Extension: With the extension to constantly send *hello messages* (section 5.3) two new parameters are introduced:

- The *hello messages* send rate which is set to 2 messages per second.
- The size of the *neighbor* subnode which is set to a maximum of 100 fraglets.

Ant Generations Improvement: With the improvement to forward ant generations (section 5.4) one further parameter has to be set: The number of ants within one generation is set to 5.

To compare the improved version with the basic implementation the production rate of ant generations is set either to 2 generations per second which yield the same number of forward ants and to 10 generations per second which yield about the same number of backward ants as the basic approach.

6.2 Real Network: NTTnet

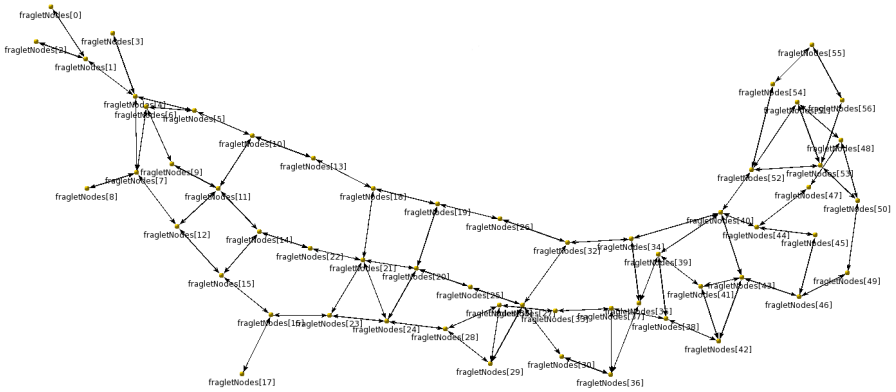


Figure 6.1: Topology of NTTnet

To measure the performance of an algorithm NTTnet, a model of a real network, is used. NTTnet is a network based on the fiber optic corporate backbone of the Nippon Telephone and Telegraph company in Japan (end of the '90s) [12]. The 57 nodes are connected through 162 bidirectional links (figure 6.1) which have a bandwidth of 6 Mbits/s each. The propagation delays vary from 1 to 5 ms.

To stress the network two traffic scenarios are used, the scenario only differs in the number of session started by one node. One scenario produces a medium load of the network with one data session per source node, the other produces a high load with four data streams. The destination for each session is chosen randomly at the beginning of the simulation. The data flow within a session is not constant, packets are sent out in burst.

Both traffic patterns share the parameter setting: All parameters are randomly chosen according to a normal distribution which is truncated to non negative values.

- Start time: $\mu = 5, \quad \sigma = 2$
- Number of bursts: $\mu = 5, \quad \sigma = 2$
- Number of packets per burst: $\mu = 80, \quad \sigma = 10$
- Time between bursts: $\mu = 0.2, \quad \sigma = 2$
- Time between packets: $\mu = 0.0005, \quad \sigma = 0.01$

The calculation expense for network with the size of NTTnet is so big that it was necessary to limit the simulation duration to get results within a feasible time (several hours to a few days per simulation run).

6.2.1 Results

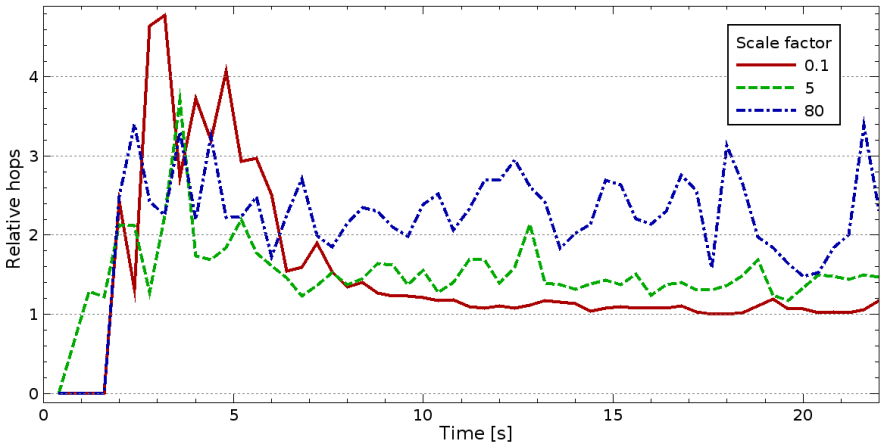


Figure 6.2: Relative hops used by packets in the NTTnet scenario with a medium traffic load

Figure 6.2 shows the relative hops used by packets to reach the destination when the network is stressed with a medium traffic load. The relative hops are calculated by $\frac{number\ of\ hops\ used}{number\ of\ hops\ of\ the\ shortest\ path}$. The plot shows the results for a passive setting ($\phi = 0.1$), a medium setting ($\phi = 5$) and a aggressive setting ($\phi = 80$).

While the passive setting produces longer routes it achieves the best result after a while, 8 seconds in this case. This setting is feasible only if data streams do not vary too much. The paths found by the aggressive setting are significantly longer, and therefore this setting is not suitable for this scenario. The best results are achieved by the medium settings, it converges faster than the passive setting and produces acceptable results over the whole simulation time.

The results of the settings change if the traffic load is increased as it can be seen in figure 6.3 which shows the relative hops for the high traffic scenario. The passive setting is not able to cope with the increased dynamic induced by the additional traffic and therefore does not find short routes. On the other side the aggressive setting produces slightly better results than the medium setting.

6.3 Simple Networks

The following scenarios with the simple networks are used to demonstrate specific properties of the algorithm under controlled conditions.

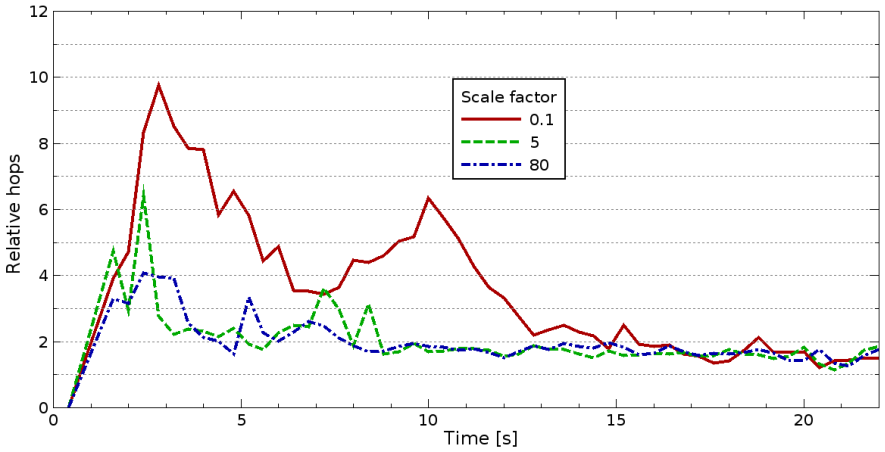


Figure 6.3: Relative hops used by packets in the NTTnet scenario with a high traffic load

6.3.1 Choosing the Best Link

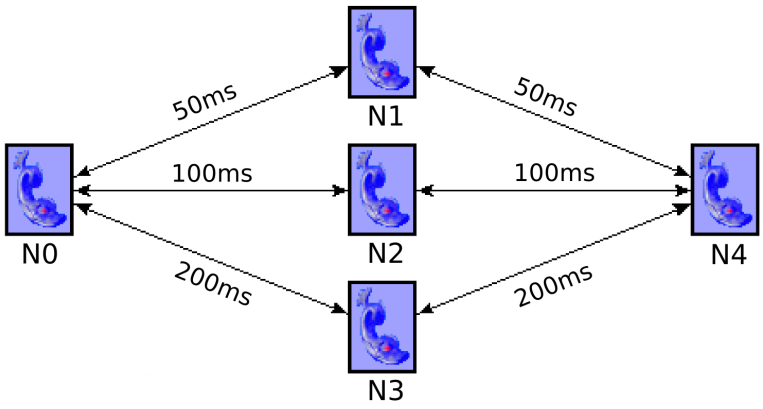


Figure 6.4: Network for scenario: Choosing the best link (6.3.1)

This scenario is used to show if the algorithms use the best link (lowest delay) to the destination node. Figure 6.4 shows the topology of the network: The source node on the left side (N0) is connected over three different links to the destination on the right side (N4). The intermediate nodes are just used for measurement. All three paths have the same bandwidth, but differ in link delay.

To determine if the best path is chosen the amount of data over all three intermediate nodes is measured. To visualize the results the percentage of the traffic which is routed over node N1 is plotted. Ten independent runs

are used to average the outcomes of the simulations.

Results for the Basic Implementation

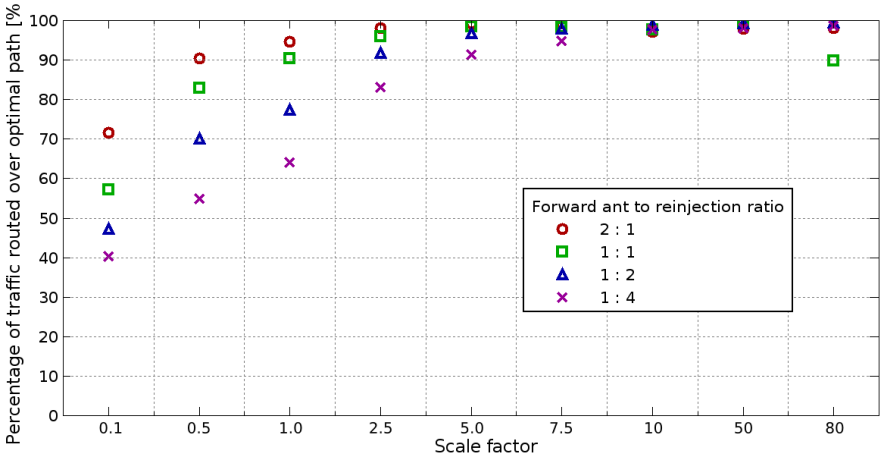


Figure 6.5: Percentage of the traffic routed over the best link

Figure 6.5 shows the results for the different parameter settings and the forward ant to reinjection ratios as described in section 6.1.

As the plot shows the aggressive parameter settings (high scale factor) achieve better results for these tasks and a high injection rate of random routing information hinder the ants to find the optimal route. This is because the reinjection is only useful to regenerate the exploration space of the ants and therefore it has just little use in this scenario. Another interesting aspect is that aggressive configurations cope better with the higher reinjection ratio.

The drop of result for setting scale factor $\phi = 80$ and a ratio of 1:1 is on account of one single bad run where the ants have choose a suboptimal route at the beginning and have not be able to switch to better path throughout the whole simulation. This behavior rarely appears but with an increased scale factor the probability increases as well. In such a case a higher reinjection ratio should help to find the optimal path.

6.3.2 Route Disappears

In this scenario the traffic is initially routed only over the better out of two different routes. Then the intermediate node of the better route fails permanently and therefore only one route remains. An algorithm should react as fast as possible and reroute the whole traffic over the backup route. Because the link fails permanently it is also important that the algorithms

quickly remove the broken node form the exploration space of the ants. The adaption process has three phases:

1. The traffic is still routed towards the broken node and therefore all data is lost.

2. The traffic is rerouted over the backup path, but a small amount of data is routed over the broken node because it is still within the exploration space.

3. The broken node disappears from the exploration space and therefore no more traffic is routed over it.

During the simulation the time spend to reach the second phase is measured. This time is an indicator for the reactiveness of the algorithm and it approximately corresponds to the amount of data lost during the first phase. To measure the time the first point in time when 100% of the traffic is routed over the backup path is registered. The measurement is done with a high sampling rate of 10 samples per seconds which has the advantage that results have an acceptable time resolution but the recording is less prone to spikes produced by randomly routed packets. Additionally, the percentage of data lost during the second phase is registered.

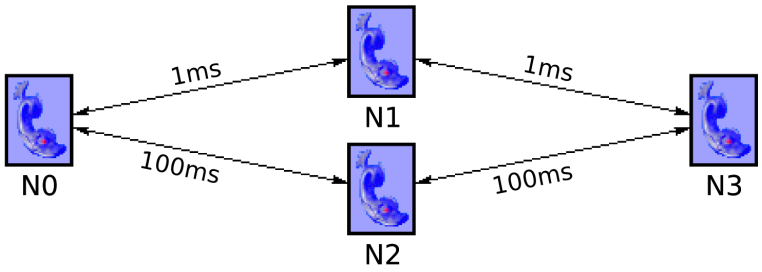


Figure 6.6: Network for scenario: Route disappears

Setup: Figure 6.6 shows the topology used in this scenario. The source on the left side (N0) and the destination on the right side (N3) are connected through two routes. The upper route (over node N1) has a significant lower end to end delay than the lower backup route (over node N2). After the initialization phase which is used to build up the path over the faster route, the node N1 is disabled.

The source sends packets with a constant rate to the destination. The bandwidth used is noticeable lower than the capacity of one link.

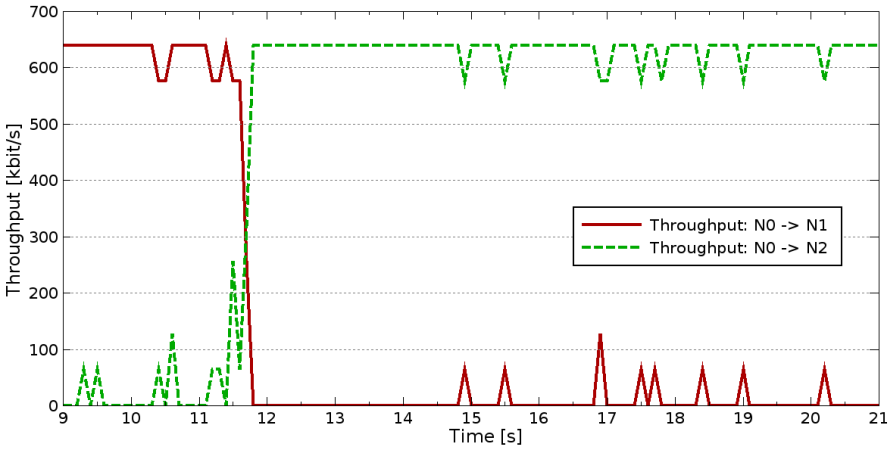


Figure 6.7: Devolution of the data flow if one node disappears with parameter setting: Scale factor $\phi = 2.5$ and a ration of 1:1

Results

Basic Implementation: Figure 6.7 shows what happens if the middle node of the preferred route (N1) fails at time $t = 10$ seconds. In this run the algorithm needs 0.6s until the first reaction and 1.8s until almost all traffic is switched to the backup route. Because the broken path is still in the exploration space the algorithm still tries to route a few packets over the failed node which explains the spikes beyond 12 second. The algorithm is not able to handle permanently broken nodes and therefore it never reaches the third phase, where the broken nodes are removed from exploration space.

The average time to reroute the traffic for the parameter settings described in section 6.1 are illustrated in figure 6.8.

Despite the steps in the plot a clear tendency can be observed, the algorithm shows stable properties over a big fraction of the parameter range and the determining factor is the reinjection ratio. A high reinjection rate increases the probability to find the backup route which was oppressed by the ants before the better route fails and therefore decreases the average time needed by the first ant to find the second route.

Figure 6.9 shows the percentage of lost data packets during the second phase which lasts forever because the algorithm is not able to remove broken nodes from the exploration space. Higher reinjection rates causes higher data loss due to the increased injection of random routing information which obviously leads to a more random routing. On the other hand the plot shows that aggressive settings are able to suppress the effect introduced by the injections. The three points for scale factor $\phi = 0.1$ which are not plotted are just out of range but follow clearly the tendency of the curves.

In this scenario the algorithm shows the best performance with a high

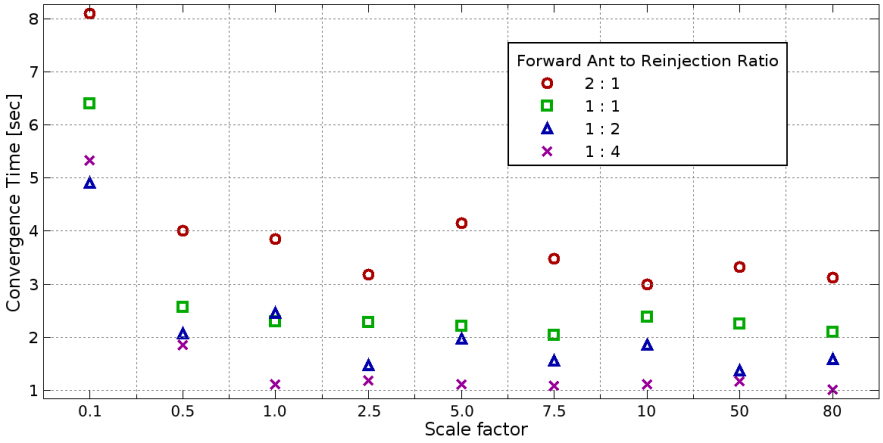


Figure 6.8: Average time to reroute traffic to backup link

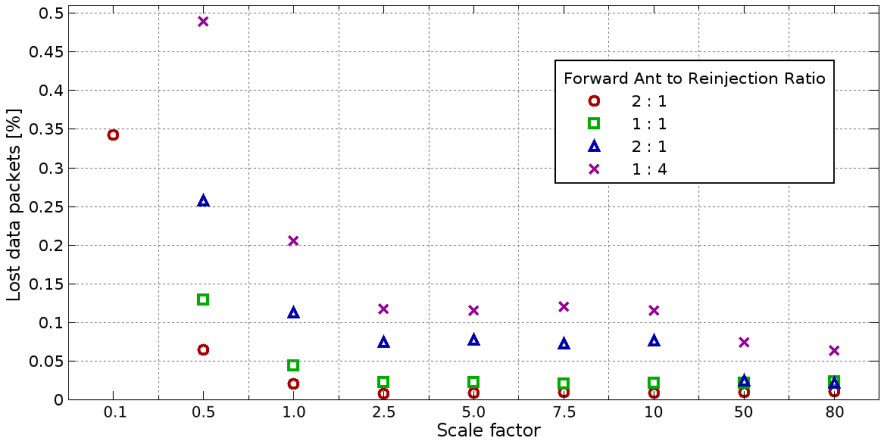


Figure 6.9: Percentage of lost data packets during the second phase

reinjection rate to lower the time needed to reroute the traffic; and to minimize the data loss produced by the high reinjection rate a very high scale factor must be chosen.

Hello Messages Extension: Figure 6.10 shows the concentration of neighbor fraglets `[ngb N1]` and `[ngb N2]` in the *neighbor* subnode of node N0. Due to the the equal rate of *hello messages* the concentrations are about the same until the node N1 is fails at time $t = 8$ seconds. Afterwards the messages from node N2 starts to displace the information about node N1. About 8.5 seconds after the node N1 failed no information about the broken node remains and thereby the broken node is not in the injected exploration space anymore.

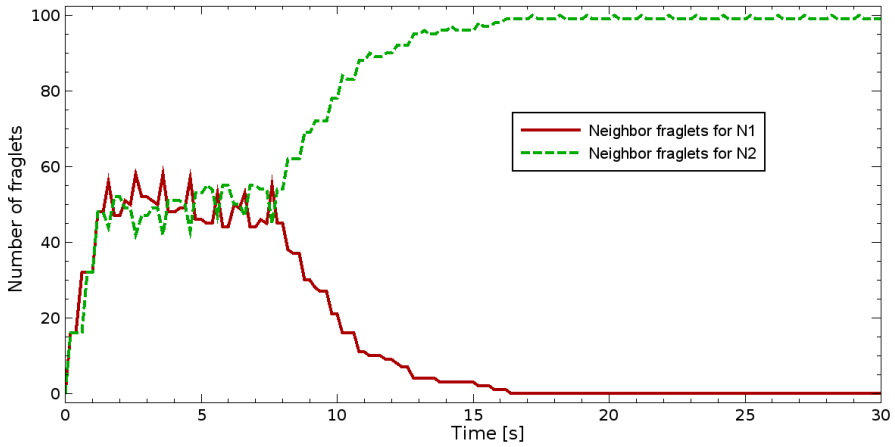


Figure 6.10: Concentration of the two neighbors in the *neighbor* subnode of node N0

6.3.3 New Node Appears

This scenario is the opposite of the previous one. It starts with only one slow route and after a while a new node appears that builds a faster route from the source to the destination. It is used to demonstrate the reaction of the algorithm to a network topology change.

Setup: For this scenario the same network topology and traffic pattern as in the previous one is used. But in contrast to it this scenario starts with the node N1 disabled and enables it after a while.

Results

Basic Implementation: The basic implementation is not able to cope with changes in the topology, therefore it is not applicable in this scenario.

Hello Messages Extension: Figure 6.11 shows the concentration of the two neighbors in the *neighbor* subnode of node N0. The first six seconds only node N2 is present. At time $t = 6$ seconds node N1 is enabled and it begins to insert its information in the *neighbor* subnode of node N0. After about 3.5 seconds the equilibrium is established.

The reaction of the algorithm is illustrated in figure 6.12 which shows that the algorithm needs about 2 seconds after the equilibrium is established or totally 5.5 seconds to find the new path.

The time difference between the the switch to the new route and the establishment of the equilibrium depends on the scale factor and the reinjection ratio and should be about the same as the time to find the remaining route in scenario *route disappears* (section 6.3.2).

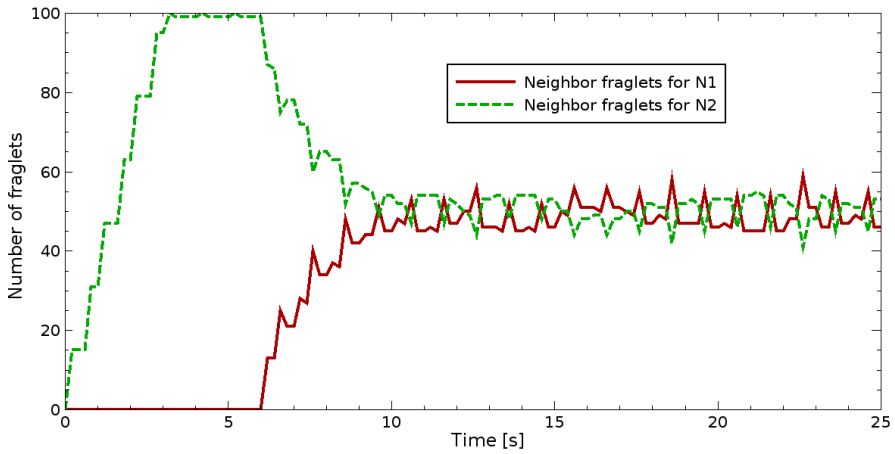


Figure 6.11: Concentration of the two neighbors in the *neighbor* subnode of node N0

The spikes in the curve are produced by the *hello message* send at a rate of two message per second.

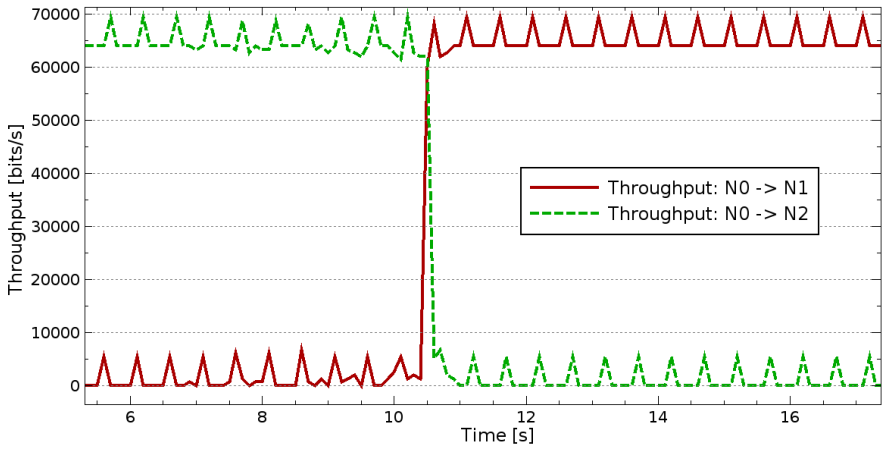


Figure 6.12: Reaction to the appearance of a better path at time $t = 6$ seconds

6.3.4 Static Path Quality

This scenario is used to show the path finding quality of the algorithms. The used network is a grid of nodes with connections to its neighbors, it is a demanding topology for ant routing algorithms. In such a network a lot of different paths to the destination exist and the length difference of the path is rather small. The task is to find the optimal path which has a

length of four hops. In this scenario the convergence to the optimal path is illustrated.

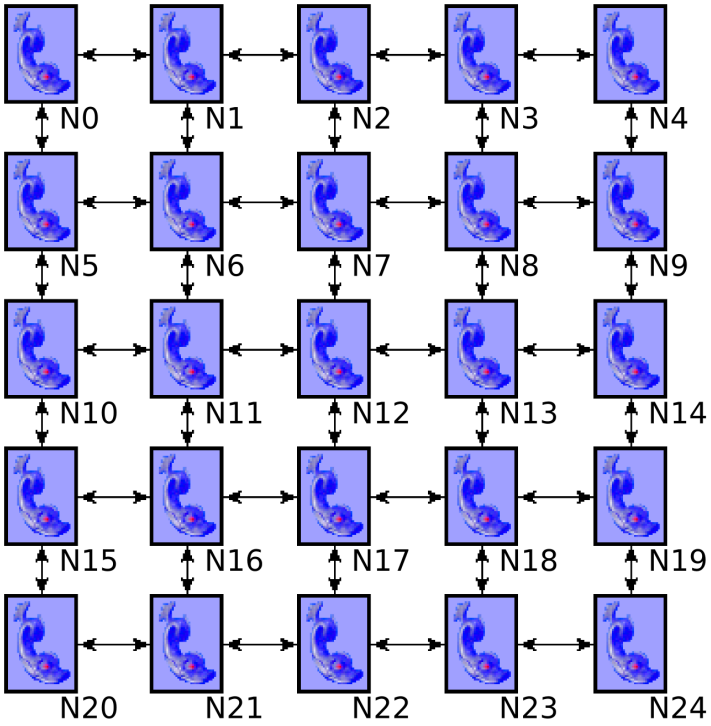


Figure 6.13: Network for scenario: Static path quality

Setup: The grid topology of the network can be seen in figure 6.13. The middle left node (N10) is the source which produces a data stream to the middle node on the right side (N14).

The source node sends out a constant data stream with a low rate which is only used for measuring purpose and should not induce dynamic effects.

Results

Basic Implementation: Figure 6.14 shows the four different convergence behaviors produced by the algorithm over the whole range for the scale factor. With a $\phi = 0.1$ the algorithm is not able to find good routes. With a higher scale factor the ants begin to converge to the optimal path. With an increased factor the convergence time is lowered as the difference between parameter setting $\phi = 0.5$ and $\phi = 5$ illustrates. The two most aggressive settings ($\phi = 50$ and $\phi = 80$) converge very fast to one path but are not able

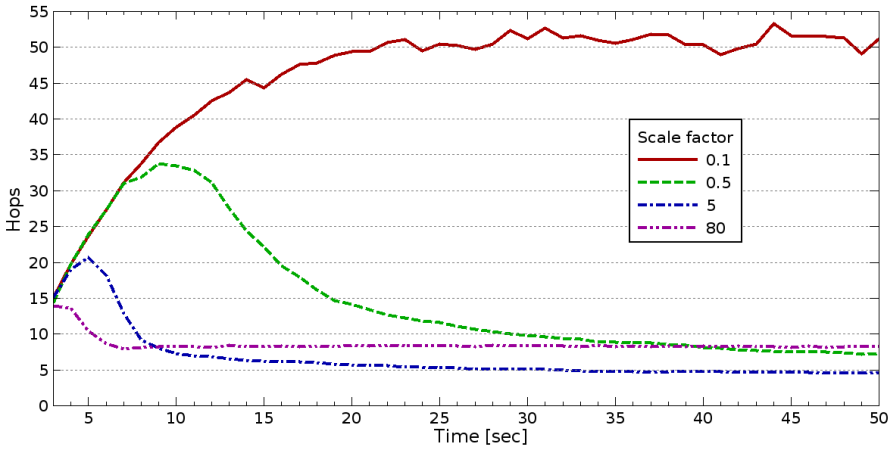


Figure 6.14: Convergence behavior of the basic implementation in the static path scenario

to find the optimal route. Because the impact of the first backward ant is too strong the following forward ants have no chance to find a better route.

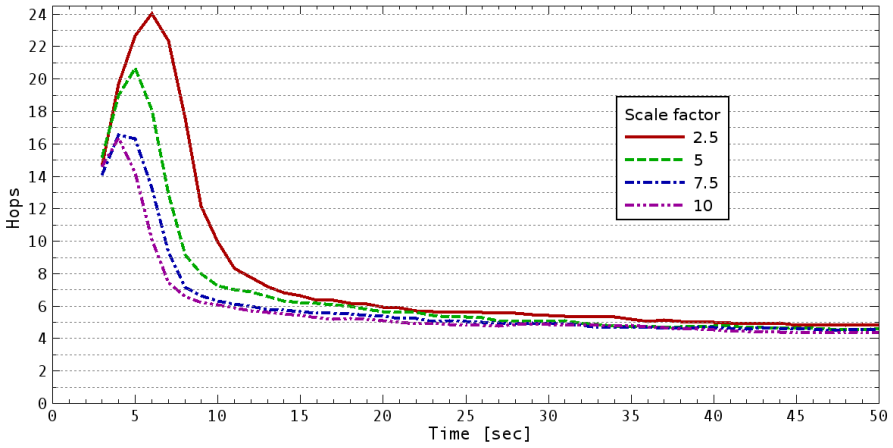


Figure 6.15: Convergence behavior for the scale factor settings that find the optimal path

Figure 6.15 shows the convergence behavior of the four parameter settings that converge to the optimal path within the simulated time, it illustrates the difference of the convergence time.

The influence of the forward ants reinjection ratio was also studied for this scenario. But the observed effects are too small to be significant.

Ant Generations Improvement: Figure 6.16 shows the convergence behavior of the improved implementation (section 5.4) compared to the basic

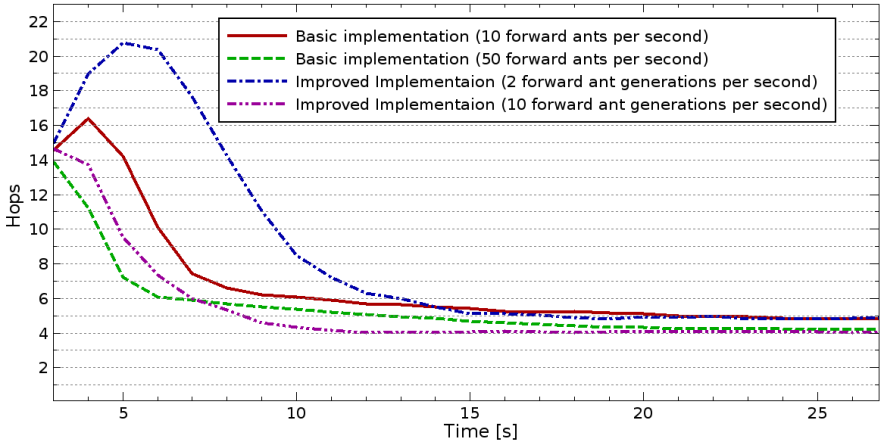


Figure 6.16: Convergence behavior of the improved implementation with forward ant generations compared to the basic implementation

implementation. The scale factor is set to 10 and a ratio of 1:1 is used, which is the best setting for this scenario. The two curves for the improved version vary in the forward ant production rate.

In the simulation five forward ant are simultaneously sent out in one generation and therefore a production rate of 2 ant generations per second produces the same number of forward ants as in the standard setting for the basic implementation. As the plot shows this setting uses longer paths in the beginning and need a long time (about 14.5 seconds) to produce as good results as the basic implementation.

But with standard production rate (10 ant generations per second) the improved version converge within 12 seconds to the optimal solution which is a significantly better convergence behavior than the basic implementation, even if the ant production rate of the basic implementation is set to 50 ants per second.

The traffic overhead induced by the ants is three times higher for the improved version with the high production rate compared to the basic implementation with standard production rate and only 2/3 of the overhead produced by the basic implementation with the higher production rate.

6.3.5 Oscillation

In this scenario it is tested whether the algorithm tends to oscillate between routes. For this purpose two sources produce a data stream to one destination. It is watched if an algorithm is able to build up and maintain stable routes.

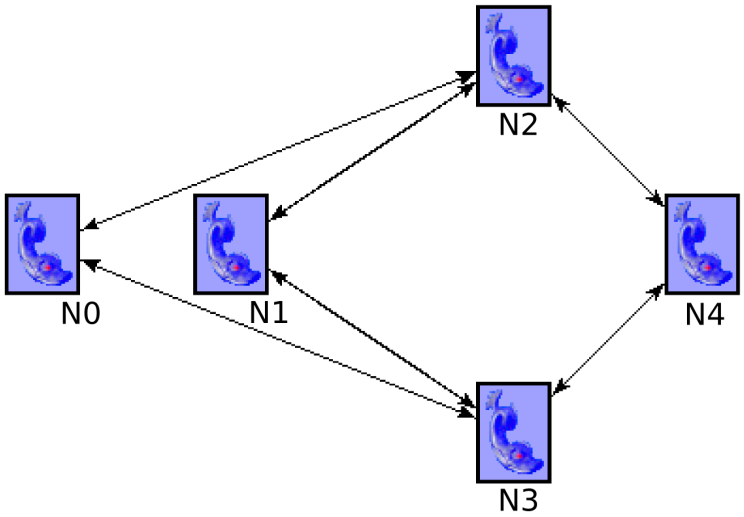


Figure 6.17: Network for scenario: Oscillation

Setup: Figure 6.17 shows the topology used in this scenario. The two nodes on the left side (node N0/N1) produce a data stream to the node on the right side (node N4). For the stimulation of the oscillation the link between node N2 and node N4 is deactivated for 0.5 seconds. The sources each produce a constant data stream which consume together 60% of the network capacity. To measure the oscillation the sampled throughput on the links N0-N2 and N1-N2 is transformed by a fast Fourier transformation to the frequency domain. The frequency spectrum of the to other links (N0-N3 and N1-N3) are equal and therefore are not taken into account.

Results

Basic Implementation: Figure 6.18 shows the frequency components for the parameter settings: scale factor $\phi = 0.1, 5, 80$ and a ration of 1:1. For the sake of clarity the other curves are not plotted, but the can be divided into three groups:

1. $\phi = 0.1, 0.5, 1$: All three have a parallel frequency distribution and higher amplitudes with increasing scale factors.

2. $\phi = 2.5, 5, 7.5, 10$: This four settings have about the same frequency spectrum and amplitudes.

3. $\phi = 50, 80$: The settings with the highest scale factors show both decreased amplitudes.

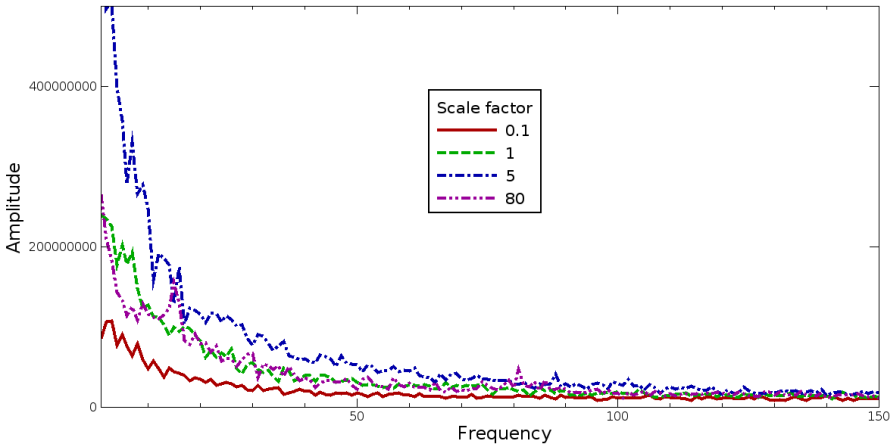


Figure 6.18: Frequency components of the sampled throughput for parameter settings: scale factor $\phi = 0.1, 5, 80$

Although there are significant difference in the oscillation amplitudes, observed oscillation behavior is not within an acceptable range for any setting. It seems that the forward ant to reinjection ration does not have any effect on the oscillation property.

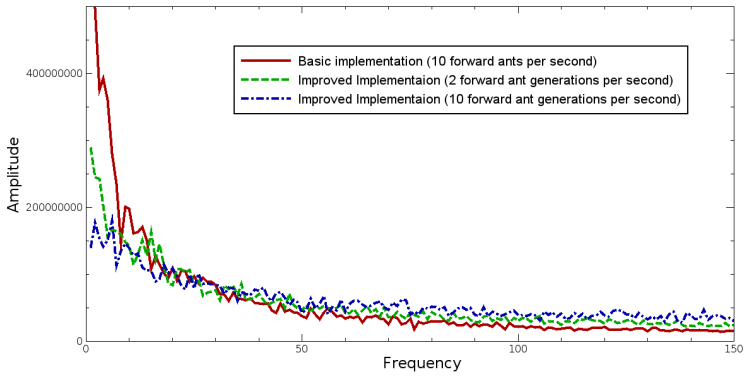


Figure 6.19: Oscillation frequency spectrum of the improved implementation with forward ant generations compared to the basic implementation

Ant Generations Improvement: As Figure 6.19 shows the ant generations produces bigger oscillations in the higher frequency spectrum compared to the basic implementation. This is an effect of the increased aggressiveness of the algorithm induced by selective deletion of backward ants.

Chapter 7

Conclusion

As the implementation and the simulation results described in this thesis show, it is possible to implement an ACO inspired routing algorithm in fraglets, a programming language based on chemical metaphor. The approach was to take a working algorithm and rewrite it in the fraglets language. AntNet was chosen as a the base of the implementation because it is an already highly optimized algorithm. But due to the complexity of AntNet and the characteristic of the fraglets language, it was not possible to map every aspect of the algorithm to the chemical environment. To build a working implementation the AntNet approach was combined with the AntHocNet update rule and some own adaptations. The AntHocNet update rule was chosen due to its simplicity and the low numbers of parameters.

To implement the developed approach in fraglets, the algorithm was divided into functional blocks like backward ant handling, pheromone update or the loop check of forward ants. This blocks corresponds to procedures in classical programming languages. In the fraglets environment either single fraglets or fraglet groups are used to emulate the corresponding functionality.

Some of these functional blocks nicely match with the ideas behind fraglets, the best example is the probabilistic routing which was easy to map to a selection process based on concentrations.

To implement the other parts, the emulation of the three structures (sequence, selection and repetition) of the structured programming paradigm are used. With this three emulated structures it is possible to program in fraglets similar to any structured programming language. But the resulting programs are neither elegant nor very efficient. Especially the emulation of loops is computationally intensive due to the high number of instructions which are needed.

In general, the unconditional execution of operation is unproblematic to implement in fraglets. It is straightforward to trigger an execution by the appearance of a fraglet but it is not possible to directly react on the absence

of a fraglet. A more flexible reaction mechanism which bases the matching decision on an affinity value and not on a equal or not condition, could help to facilitate such conditional executions. Thereby the affinity value would correspond to the similarity of tags and reflects the probability of a reaction.

Despite the difficulties induced by the fraglets language the implementation of ACO inspired algorithm in artificial chemistries is an interesting approach. There are many possibilities to optimize and improve the basic implementation described in this thesis. Especially because some advantageous aspect of the fraglets environment are not used in the implementation. One example are the capabilities for code production and modification, which could be used to implement a self healing and self optimizing version of an ACO inspired algorithm.

The simulation results have demonstrated the ability of the algorithm to route packets. On the other hand the results are not optimal and therefore it would be interesting to search for optimizations strategies which nicely match with the fraglets environment.

Another approach to overcome the implementation difficulties would be a hybrid approach, a combination of a classical and a chemical environment. In such an approach the classical parts could handle all the bookkeeping and would deliver a framework which sends ants around to gather information. The chemical part would be responsible for parameters, pheromone updates and routing which are the essential parts for the performance of the algorithm. To improve the routing performance self optimization strategies as genetic programming could be applied to the chemical part of a hybrid approach.

Appendix A

Configuration of the Implemented OMNeT++ Modules

A.1 FragletNode

AntSource, PacketSource0-3 (string): Type of the ant source and the four packet sources

address (string): Network address of the node

A.2 FragletSoup

fragletFile (string): File name of the fraglets file injected during the initialization

timeScale (string): Time evolution algorithm of the fraglets environment: `< lin | det | sto >`

transAlg (string): Transformation algorithm of the fraglets environment: `< min | bal | inf >`

reactAlg (string): Reaction algorithm of the fraglets environment: `< min | sub | prod | prode | prodeboltzmann | prodfact >`

keepEducts (bool): Don't remove the educts of a reaction from the reaction vessel. Enable this flag to simulate a catalytic flow system.

startTo (string): Print initial fraglet multiset to the specified file

traceTo (string): Print reaction traces to the specified file

inspectTo (string): Print information according to the 'i' commands of the injected fraglets file to the specified file

networkTo (string): Print reaction graph (.dot file) to the specified file

reactionTo (string): Print reaction graph (.in file) to the specified file

endTo (string): Print final fraglet multiset to the specified file

outputResolution (numeric): Time resolution of the information output

omnetTrace (bool): Print reaction traces to the OMNeT++ output channel

timeSync (bool): Synchronize the time of the fraglets environment with the simulation time of OMNeT++. If not synchronized all pending fraglets instructions are processed during one OMNeT++ event

timeFactor (numeric): Acceleration factor of the fraglets time with respect to the OMNeT++ simulation time

startEnabled (bool): Determines whether the soup is enabled or disable at the beginning of the simulation

startStopTime (string): Takes an array of start/stop times in the form: "time0 time1 time2 ...". At the specified points in time the state of the `FragleSoup` between enabled and dissabled.

A.3 Queue

size (numeric): Size of the queue in bits

startStopTime (string): Takes an array of start/stop times in the form: "time0 time1 time2 ...". At the specified points in time the state of the `Queue` between enabled and dissabled.

A.4 Sink

resolution (numeric): Time resolution of the sampled data

A.5 ThroughputMeasure

startTime (numeric): Time at which the recording of measurement data starts

resolution (numeric): Time resolution of the sampled data

A.6 Measurement

startTime (numeric): Time at which the recording of measurement data starts

resolution (numeric): Time resolution of the sampled data

Bibliography

[1] C. Anderson, G. Theraulaz, and J.-L. Deneubourg. Self-assemblages in insect societies. *Insectes Sociaux*, 49(2):99–110, May 2002.

[2] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Sci. Comput. Program.*, 15(1):55–77, 1990.

[3] R. Beckers, J. L. Deneubourg, and S. Goss. Modulation of trail laying in the ant lasius niger (hymenoptera: Formicidae) and its role in the collective selection of a food source. *Journal of Insect Behavior*, 6(6):751–759, November 2005.

[4] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[5] Bionets simulator project: http://bionets.hit.bme.hu.

[6] J.L. Deneubourg, S. Aron, S. Goss, and J.M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3(2):159–168, 1990.

[7] J.L. Deneubourg, J.M. Pasteels, and J.C. Verhaeghe. Probabilistic behaviour in ants: a strategy of errors? *Journal of Theoretical Biology*, 105:259–271, 1983.

[8] Gianni Di Caro, Frederick Ducatelle, and Luca Maria Gambardella. Anthocnet: an adaptive nature-inspired algorithm for routing in mobile ad hoc networks. *European Transactions on Telecommunications (ETT), Special Issue on Self Organization in Mobile Networking*, 16(5):443–455, September 2005.

[9] Peter Dittrich. *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, chapter Chemical Computing, pages 19–32. Springer, Berlin, 2005.

[10] Peter Dittrich, Jens Ziegler, and Wolfgang Banzhaf. Artificial chemistries - a review. *Artificial Life*, 7:225–275, 2001.

[11] Marco Dorigo, Christian Blum, and Jordi Girona. Ant colony optimization theory: a survey. *Theoretical Computer Science*, 344(2-3):243–278, November 2005.

[12] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, London, 2004.

[13] S. Goss, S. Aron, J.L. Deneubourg, and J.M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

[14] Bert Hölldobler and Edward O. Wilson. *Journey to the Ants: A Story of Scientific Exploration*. Belknap Press, 1994.

[15] Thomas Meyer, Lidia Yamamoto, and Christian Tschudin. *Bio-Inspired Computing and Communication*, volume 5151 of *Lecture Notes in Computer Science*, chapter An Artificial Chemistry for Networking, pages 45–57. Springer, Berlin, 2008.

[16] Omnet++: http://www.omnetpp.org.

[17] Gheorghe Paun, Turku Centre, and Computer Science. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 2000.

[18] Christian Tschudin. Fraglets - a metabolistic execution model for communication protocols. In *2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS)*, Menlo Park, USA, July 2003.

[19] Lidia Yamamoto, Daniel Schrekling, and Thomas Meyer. Self-replicating and self-modifying programs in fraglets. In *Proceedings of the 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS 2007)*, Budapest, December 2007.