Genetic Algorithms as a feasible re-planning mechanism for Beliefs-Desires-Intentions agents

G. Shaw

May 2015

Abstract

The BDI agent architecture includes a plan library containing pre-defined plans. The plan library is included in the agent architecture to reduce the need for expensive means-end reasoning, however can hinder the agent's effectiveness when operating in a changing environment. Existing research on integrating different planning methods into the BDI agent to overcome this limitation include HTNs, state-space planning and Graphplan. Genetic Algorithms (GAs) have not yet been used for this purpose.

This dissertation investigates the feasibility of using GAs as a plan modification mechanism for BDI agents. It covers the design of a plan structure that can be encoded into a binary string, which can be operated on by the genetic operators. The effectiveness of the agent in a changing environment is compared to an agent without the GA plan modification mechanism.

The dissertation shows that GAs are a feasible plan modification mechanism for BDI agents.

Key Words

Agent, Autonomous agents, Belief-Desire-Intention, BDI, Genetic algorithms, Planning

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Autonomous agents are a form of software program that is capable of independent actions in pursuit of its goals. While the exact definition of what constitutes an agent is not universally agreed upon, some of the common criteria used to describe agents include the requirement to be situated in a specific environment, to be able to sense the environment and to act on the environment in pursuit of its own agenda (Franklin and Graesser 1996; Russell and Norvig 2010; Wooldridge and Jennings 1995a).

Based on those definitions, this dissertation will define an agent as a piece of software which is in an environment and is part of that environment and which can perceive and act upon that environment in pursuit of its goals, without input from an external source.

Agents began as an extension of Distributed Artificial Intelligence (DAI), a sub-field of Artificial Intelligence (AI). One of the earliest agent architectures to be explored was the deliberative architecture. The earliest of these deliberative architecture agents were GPS (Allen Newell and Herbert A Simon 1961) and STRIPS (Fikes and Nilsson 1972). A later architecture, better equipped to handle time-constrained problems, was the reactive AI theory (R. Brooks 1986; R. A. Brooks 1990).

In the late 1980s and early 1990s, researchers began looking at alternatives to the purely reactive architecture. One of the results of this was the Belief-Desire-Intention (BDI) architecture (Jennings et al. 1998).

The BDI architecture is based on the theory of practical reasoning (M. E. Bratman et al. 1991). In the theory intentions are considered as incomplete, structured courses of action. The theory also covers what makes intentions rational, mostly relating to whether the intention is achievable and how a rational agent should act when intentions that it has are no longer achievable.

While originally a theory of philosophy (M. Bratman 1987), Bratman, along with Israel and Pollack, considered the applications of the theory of practical reasoning to Artificial Intelligence (AI) (M. E. Bratman et al. 1991). The architecture that they proposed for agents, the BDI architecture, is intended for resource-bounded rational agents. The basic components of the architecture include beliefs, what the agent knows about the environment; desires, general goals that the agent wishes to achieve and intentions, goals that the agent has committed to and that are achievable. The BDI architecture also includes a plan library, a set of known actions that can be used to achieve goals.

While having a pre-built plan library does allow the agent to pursue its goals quickly and to avoid the cost of resource-intensive deliberative planning (Walczak et al. 2006), the library does constrain the agent to the environment it was designed for and the plans can become inappropriate if the environment changes sufficiently. Changes in the environment can result in the plans contained within the library becoming less efficient and even causing them to fail to deliver the results the agent expects (Meneguzzi et al. 2004).

This lack of deliberative planning in BDI agents has been the topic of many papers over recent years with various options proposed for extending the BDI architecture to include some form of planning.

Meneguzzi et al (Meneguzzi et al. 2004) added a form of propositional planning using a Graphplan-based algorithm and De Silva and Padgham (Silva and Padgham 2004; De Silva and Padgham 2005) investigated integrating hierarchical task networks with BDI agents. Walczak et al (Walczak et al. 2006) investigated the integration of a state-based planner into a BDI Agent. The 3APL agent programming language allows for plan revision rules to be defined, allowing plans to change during execution (Ten Hoeve et al. 2003). The XFRM planner, designed for robots in changing environments, revises the plans based on their outcomes (Beetz and McDermott 1994). A plan modification system for reactive agents has also been investigated (Boella and Damiano 2002). Their system allows for plans to be decomposed into partial plans and then reassembled to form different plans. This use of partial plans is similar to the plan design in this dissertation.

These bodies of research are discussed in more detail in Section 2.5.

1.1 Problem Statement

The BDI Agent architecture uses a plan library, defined at the time that the agent is created, which it uses to achieve its goals. This plan library is fixed for the life of the agent.

The fixed plan library exists so that the agent does not have to perform resource-intensive planning during its normal execution. In a static environment, a static plan library is sufficient. In a dynamic environment, the static plan library may be problematic, as the plans may become less suitable as the environment changes (Meneguzzi et al. 2004).

Previous research into integrating planning have investigated state-based planners, hierarchical task networks and propositional planning (Walczak et al. 2006; Silva and Padgham 2004; De Silva and Padgham 2005;

Meneguzzi et al. 2004). However there has been no investigation into using genetic algorithms to modify plans within a BDI agent.

Genetic Algorithms (GAs) are a form of evolutionary computation, originally devised by Holland (John H Holland 1975). They are search and optimisation algorithms based on the principal of biological evolution. GAs have been applied to many problem areas, including that of motion planning and network routing (Ahuactzin et al. 1991; Sugihara and Smith 1997; Ahn and Ramakrishna 2002).

GAs have been used in combination with agents in a number of ways. They have been used to evolve a genotype from which agents can be developed, as well as to select intentions as part of the deliberation cycle.

In this dissertation GAs will be investigated to determine whether they are a feasible method of plan modification for BDI agents. The scenario in which this will be tested is that of a simulated network routing problem. A directed graph is used for the network. The BDI agents will be tasked with transmitting messages across the simulated network, using the route with the lowest latency. The message transmission times will be used to determine the performance of the agents.

1.2 Research Question

Based on the problem statement defined above, the question which this study attempts to answer can be refined as follows:

Is a GA a feasible mechanism for plan modification for BDI agents?

1.3 Methodology

The methodology in this dissertation is the experiment. The aim of the study is to test a theory, specifically the theory that GAs can be a feasible plan modification mechanism for BDI agents. The testing of a theory is one of the goals of experimental research in Computer Science (Olivier 2009).

In order to test the theory, this study uses a simulated communication network as the environment. A BDI agent, referred to as the Environment Agent (EA), operates on the environment to simulate entropy. This agent changes the latency on nodes and links within the network. These changes ensure that the plans with which the BDI agent starts will change in efficiency as time progresses

The component responsible for delivering messages across the network is a BDI Agent referred to as the Messaging Agent (MA). The MA has a set of plans which are valid when each test starts. Due to the operation of the EA, the plans may not remain valid for the duration of the test. The performance of the MA will change over the course of the test due to the operation of the EA. The exact definition of the performance of the agent, and how it is measured, are discussed in Section 4.2.3.

The GA plan modification component is integrated into the MA. The comparisons between the performance of the MA without the GA plan modification component and performance of the MA with the GA plan modification component is used to determine whether or not the GA is an effective mechanism for plan modification in this specific environment.

1.4 Scope

The scope of this dissertation is limited to determining whether GAs can be a feasible mechanism for plan modification in BDI Agents within a simulated network environment. As such, the only comparisons will be between the BDI Agent running without the GA plan modification mechanism and the BDI Agent running with the GA plan modification mechanism.

1.5 Limitations and Exclusions

This dissertation is not concerned with creating new network routing protocols nor will it include any comparisons with existing network routing algorithms.

The GA plan modification process is run synchronously with the MA in this dissertation. Asynchronous operations are not investigated.

No form of predictive analysis using the GA plan modification process is considered.

It is not the goal of this study to compare the GA plan modification mechanism with any other form of planning.

Only one plan design and GA encoding will be considered. Comparisons between different forms of plan encoding will not be investigated.

1.6 Significance

This study is significant as there have been no previous studies using GAs for planning within BDI Agents. Other planning mechanisms, such as HTNs and propositional planning, have been used (Walczak et al. 2006; Silva and Padgham 2004; De Silva and Padgham 2005; Meneguzzi et al. 2004). GAs have been used for other aspects of BDI agents, including the initial design (Calderoni, Marcenac, and Courdier 1998; Calderoni and Marcenac 1998; Dellaert and Beer n.d.), but there does not appear to have been any research on using GAs for planning mechanisms within BDI agents.

GAs have been used in planning in areas other than agents, including motion planning (Sugihara and Smith 1997; Ahuactzin et al. 1991) and network routing (Ahn and Ramakrishna 2002; Nagib and Ali 2010).

1.7 Dissertation Layout

This document is laid out as follows:

Chapter 1 contains the problem statement, research objectives, a summary of the method, the scope of the study, limitations and exclusions and the dissertation layout.

Chapter 2 covers the background and details of BDI agents and will discuss several of the studies on integrating planning into the BDI agent.

Chapter 3 discusses Genetic Algorithms and their use in planning and network routing problems.

Chapter 4 covers the design of the feasibility study. It will detail how the Genetic Algorithm is integrated into the BDI agent, how the plans are structured to allow a Genetic Algorithm to operate on them, and the options for triggering the plan modification.

Chapter 5 details the results of the experiments and will show what effect the inclusion of the GAs had on the performance of the MA in the changing environment.

Chapter 6 summarises the research and suggests possible future lines of research.

Chapter 2

Autonomous Agents

In this chapter the literature on Autonomous Agents are discussed with particular focus on architectures, history and current research. To get a common ground for discussion, the different definitions of what an agent is will be examined and summarised. Three architectures, specifically the reactive, deliberative and Belief-Desire-Intention (BDI) architectures will be then be discussed in terms of their history, design advantages and shortcomings.

The BDI agent architecture will be discussed in detail. The use of a plan library will be examined in detail including its advantages and disadvantages.

Literature on integrating planning algorithms into BDI agents will be covered, since the integration is directly relevant to this study. A number of studies within the area will be examined and discussed.

2.1 History of Agents

The history of autonomous agents is a complex one with diverse roots. Several different disciplines within AI as well as disciplines outside of computer science have contributed to the current state of agent research.

Agents began as an extension of Distributed Artificial Intelligence (DAI), a sub-field of Artificial Intelligence (AI). DAI evolved and specialised into two sub fields, multi-agent systems, which studies interactions between agents, and autonomous agents, which studies the individual agents.

Within the field of autonomous agents, the deliberative architecture was one of the first to be developed. It was seen as a natural derivation of the traditional field of logical planning. The earliest of the planning systems were GPS (Allen Newell and Herbert A Simon 1961) and STRIPS (Fikes and Nilsson 1972) in the 1960s and 1970s.

By the mid 1980s it was clear that, in many circumstances, first-order principles are not adequate when dealing with a time-constrained environment (Chapman 1989; Wooldridge and Jennings 1995b). From this came the reactive AI theory (R. Brooks 1986; R. A. Brooks 1990).

In the late 1980s and early 1990s, researchers began looking at alternatives to the purely reactive architecture, which, while well suited to certain domains and problems, was less suitable for complex tasks, scenarios with changing tasks or in scenarios where the inputs have to be integrated over time (Russell and Norvig 2010; Jennings et al. 1998). One outcome was the BDI architecture, based on theories of human reasoning and thought (M. E. Bratman et al. 1991).

The other main outcome was the concept of layered architectures, where an agent has two or more layers, each with a different architecture. Research into layered architectures and their application is still continuing today (Jamali and Ren 2005).

With the origins and history of agents covered, the varying and often contradictory definitions for what an agent is, will be discussed next.

2.2 Definition of an Agent

A review of the literature surrounding agents reveals a number of different definitions of agent. There are common elements that are found in more than one of the existing definitions. In this sectio,n several of the definitions are examined and the commonalities between them discussed.

Russell and Norvig (Russell and Norvig 2010) defined an agent as anything that can be viewed as perceiving its environment and acting upon that environment. This definition is broad and far reaching and allows many unrelated items to be classified as agents. For example, a thermostat could be classified as an agent by this definition.

Maes (Maes 1995) stated that agents are computational systems that inhabit some complex environment, sense their environment and act upon it in order to achieve one or more goals.

Wooldridge and Jennings (Wooldridge and Jennings 1995b) offered two notions of an agent. They offer a weak notion of agency where an agent has the properties of autonomy, social ability, reactivity and pro-activeness. Their alternative stronger notion of agency also included mobility, veracity, rationality and a requirement that an agent will not lie.

Franklin and Graesser (Franklin and Graesser 1996) came to a definition after an exploration of the varying definitions for agents that had been mentioned in literature. They stated that "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."

The most commonly mentioned elements included in the definition of an agent are:

- Environment: An agent is situated in and is part of an environment. Often they are specific to an environment and if placed in a different environment can no longer function (Maes 1995).

- Perceptive: The agent perceives its environment through sensors. It may, depending on the requirements of the architecture, build up an internal representation of the environment. An agent does not require input to be given to it by another program or by a user (Russell and Norvig 2010).

- Reactive: An agent reacts to changes and occurrences in its environment and acts on that environment in pursuit of its own goals and objectives. It reacts in a timely manner to changes in the environment (Wooldridge and Jennings 1995a).

- Flexible: The actions which the agent takes are not scripted or predecided but are chosen by the agent in response to situations that it has perceived and recognised and in pursuit of goals or objectives that it has (Franklin and Graesser 1996).

- Autonomous: Agents operate without direct intervention of users or other programs. Agents have control over their internal state. Russell and Norvig phrase it as “A system is autonomous to the extent that its behaviour is determined by its own experience” (Russell and Norvig 2010).

- Goal-orientated: An agent has objectives and goals. The agent can take actions that are not a reaction to the environment, but are in pursuit of one or more of its goals. It must determine, before taking these actions, that the goals are achievable and that the planned action advances those goals (Wooldridge and Jennings 1995a).

- Learning: An agent learns or adapts by making changes to its behaviour based on previous experiences. Not all agent architectures allow for, or encourage learning. The reactive architecture does not allow for agents to learn or change over time. The BDI architecture allows new intents to be added to the intention list based on experiences or changes in the environment (Franklin and Graesser 1996).

In summary, it can be said that there are two important aspects that distinguish an agent from a program.

1. An agent is in an environment and is part of that environment.

2. An agent can perceive and act upon that environment in pursuit of its goals, without input from an external source.

A program will lack one or more of those aspects. This definition of an agent will be used for the remainder of this dissertation.

Having looked at the definitions and aspects of what makes an autonomous agent different from a standard computer program, three of the agent architectures will be examined along with how they are implemented and how they function.

2.3 Agent Architectures

There are a number of different architectures that have been proposed and used over the years. Some of the earlier agent architectures were the deliberative agent architectures (A. Newell and H. A Simon 1976; Fikes and Nilsson 1972; Wooldridge and Jennings 1995b) and the reactive agent architectures (R. Brooks 1986).

More recently, the BDI architecture was developed by Bratman (M. Bratman 1987). It is based on a theory of rational action in humans and on the tradition of practical reasoning. Practical reasoning is reasoning about actions and the choosing of actions. It is a form of folk-psychology that describes choice of actions in terms of attributes, such as beliefs and intentions, and behaviour as the interaction between those attributes

In this section three agent architectures are examined namely the Deliberative Agent architecture, the Subsumption Agent Architecture and the BDI Agent Architecture.

2.3.1 Deliberative Agents

A deliberative architecture contains a complete and representative symbolic model of the world in which the agent exists. This environmental model is expressed in terms of logical formulae. The agent then makes plans concerning what actions to take in the environment in a logical, deductive manner, finding the best possible action to take in any circumstance. This reasoning is based on symbolic manipulation in much the same way that a theorem-solving application would go about solving a theorem (Wooldridge and Jennings 1995b).

A well-known planning systems using a deliberative, logical method is STRIPS developed by Fikes and Nilsson (1972).

There are a number of limitations to a deliberative architecture, deriving from the use of logical formulae to express the environment in which the agent operates. To build such an agent, one must translate the environment into a symbolic description. If the environment is static, then this can be done completely and accurately within an acceptable time frame. For a dynamic environment, this symbolic description must be modified as the environment changes. This modification must be done quickly enough, so that the agent does not deduce an action based on outdated information.

Another problem with the deliberative agent architecture is one of complexity. Real-world environments are information-rich and can require a large number of logical formulae to describe completely. The more formulae exist, the more work the symbolic manipulator must do to determine the optimum action. It becomes difficult in complex environments to guarantee that the symbolic manipulator will complete its work in an acceptable time, or to guarantee that it will complete at all (Russell and Wefald 1991).

2.3.2 Reactive Agents

R. Brooks (1986) first suggested the reactive architecture as an alternative to the deliberative architecture. He introduced the Subsumption Agent Architecture, an architecture built from several simple layers, each constructed from one or more finite state machines.

In the Subsumption Agent Architecture, situations that are perceived in the environment map directly to actions that the agent can take. These actions are built into several layers, either horizontal or vertical. Lower layers having a higher priority than higher layers, as shown in Figure 2.1. The levels are implemented as augmented finite state machines (AFSM).

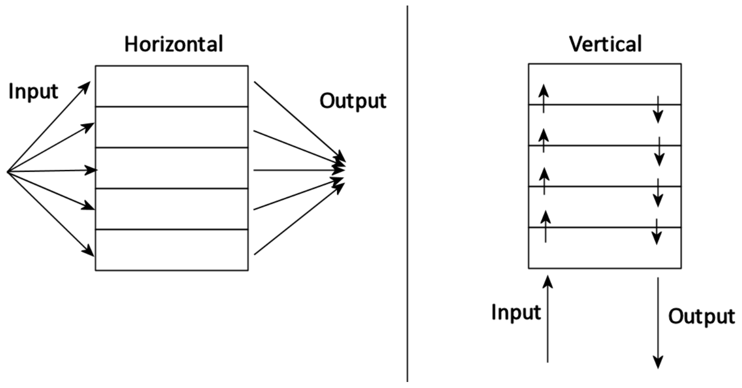

Figure 2.1: Subsumption Agent Architecture

Each level takes an input from the sensors and has an output. This output is either an action or an input to another AFSM on the same level. The state machines can also be inhibited or suppressed. If an AFSM is suppressed then the input to the AFSM is overridden. If it is inhibited then the output is suppressed. This simple set up allows one level of behaviour to override another so as to produce coherent behaviour.

The Subsumption Agent Architecture, unlike the deliberative architecture, requires no representation of the world (Luck et al. 2003). They can be fast since there is no planning or symbolic manipulation to be done. The environmental model does not need to be read to decide on an action or rewritten in response to a change in the environment. The AFSMs can be executed in parallel as they are all independent of one another.

It became clear by the early 1990s that the reactive architecture, despite its simplicity, was well suited to certain agent tasks (Jennings et al. 1998). There are, however, limitations to the reactive architecture. Since there is no stored representation of the environment, there must be sufficient information available in the environment to allow the agent to determine an acceptable output and since they do not store history and cannot take into account information that is not available in the environment, they must take a short-term view.

With the Reactive and Subsumption agent architectures covered, the Belief-Desire-Intention is discussed in the next section

2.3.3 Belief-Desire-Intention Agents

The BDI architecture, as mentioned before, is based on a theory of rational action in humans and on the tradition of practical reasoning (M. Bratman 1987).

Practical reasoning is reasoning about actions. Bratman describes it as the process of weighing the considerations for and against various options, based on desires and current beliefs about the environment (M. E. Bratman 1990).

The theory of practical reasoning focuses mainly on intentions and the part that these play in reasoning and in constraining the actions that an agent must consider. Actions that are incompatible with the current intention can be discarded with no further consideration necessary.

As a practical example of this, suppose one were to have an intention to drive from Johannesburg to Pretoria. Actions which involve booking flights or checking the swell off Port Elizabeth can be discarded, as they do not in any way assist in achieving the current intention.

The BDI architecture has three main components (Wooldridge 1995). It has beliefs about the environment, desires that the agent is attempting to achieve and intentions that it is currently working to bring about.

The beliefs which an agent holds contain all the information that the agent has about its environment. Sometimes the beliefs will also contain information about the agent's internal state. The beliefs may contain full or partial information on the environment, depending on whether the environment is fully observable or partially observable (Russell and Norvig 2010).

The desires are the goals that it currently has, that is the tasks that it is attempting to achieve. The desires can be active goals that could become intentions. They can be inactive desires if they are not achievable under the current set of beliefs or current state of the environment. The process of generating the desires is the core of the reasoning abilities of the agent.

Intentions are desires that the agent has committed to achieving. Intentions must be possible to achieve with the current beliefs. The agent must not commit to achieving a goal that it cannot achieve under the currently known conditions.

Intentions influence the beliefs and in turn should influence future reasoning. If an agent intends to achieve something, it should hold a belief that the action will be performed and the intention achieved. Hence it can develop future plans based on intentions that have not been achieved.

The requirement for intentions to be rational is one of the key features of the BDI Agent. For intentions to be considered rational, they must meet several criteria (Wooldridge 2000).

- Intentions must be achievable. It is not rational to intend to achieve a goal that cannot be achieved, either due to the state of the environment or to limitations of the agent. If an intention cannot be achieved due to the current state of the environment, then it must be discarded. It is possible that the intention may be generates at a later time, from the desires, if the state of the environment at that later time allows for the intention to be achieved.

- Intentions must lead to action. It is not rational to hold an intention and never to act upon it.

- Intentions must persist. Intentions should be held until they are achieved or are no longer achievable.

- Intentions must be discarded if they cannot be achieved. It is not rational to retain an intention that can no longer be achieved either due to the actions of the agent or due to environment changes.

The main functions that operate on the components of the BDI architecture are the option generation function, the filtering function and the action selection function. These are shown in Figure 2.2.

The option-generation function takes the current beliefs and desires and formulates a new list of desires. The option-generation function is the core of the reasoning capability of the BDI agent. The resulting desires generated by the option-generation function must be consistent with the current beliefs.

The filtering function takes the current beliefs, the current desires and the current intentions and forms a new set of intentions. The new set of intentions will be based on the previously held set of intentions and on changes that have occurred in the environment since the previous intentions were generated. Intentions that can no longer be achieved, intentions that have too high a cost and intentions that have already been achieved will be dropped.

The action-selection function takes the current list of intentions and selects one option to pursue based upon the current state of the environment and upon the plans that it has in its plan library.

The option-generation, filtering and action-selection functions are collectively known as the deliberation phase (Wooldridge 1995). Below is a high-level description of the deliberation phase of a BDI agent:

1. Updating of Beliefs: The agent will update its set of beliefs based the current beliefs that it holds and on what it can observe in its environment. The agent may also update beliefs based on the achievability of desires or intentions that it has (Rao and Georgeff 1995).

2. Updating of Desires: The desires will be updated based on the current desires and intentions and on the new set of beliefs.

3. Updating of Intentions: The agent will select new intentions based on the current intentions and desires. These intentions will be ones that, based on the state of the environment, it believes it can achieve.

4. Select Action: In this phase the agent takes the set of intentions, i.e. goals that it has committed to achieving and, based on the current state of the environment, selects a plan that it will use to achieve said intention.

This sequence is illustrated in Figure 2.2.

The BDI architecture uses a library of built-in plans. These plans store the implementation details for the selected intentions (Wooldridge 1995). This gives the agent the ability to select from a number of possible plans based on the current set of beliefs. Rao and Georgeff (1995) described this plan library as specifications of the options that the agent has for achieving

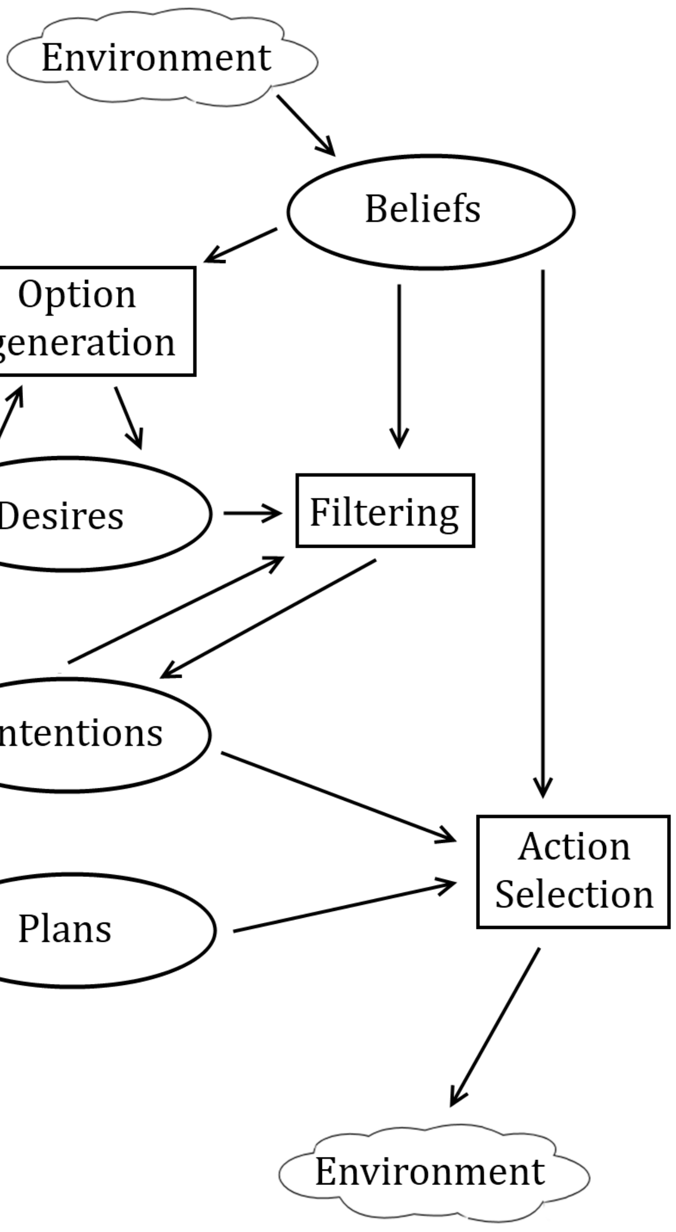

Figure 2.2: BDI Agent Deliberation Phase

its desires and M. E. Bratman et al. (1991) describe the plan library as a set of recipes for achieving desired outcomes.

The reason this plan library exists is because agents are considered to be resource-bounded. The existence of a known set of options that have an intended effect in a situation, defined by a set of beliefs, allows the agent to reduce the amount of costly reasoning that it must do. The purpose of a plan library is three-fold (M. E. Bratman et al. 1991):

- Drive the means-end reasoning that the agent engages in. If an agent has no plans covering a set of beliefs and desires then the agent should not consider those desires when formulating intentions as it will have no ability to achieve that desire.

- Provide constraints on the options that the agent needs to consider on any execution cycle, as it can avoid considering options that it has no available plan to achieve.

- Influence the beliefs that the agent holds. As an example an agent that only has plans involving the order in which to visit a number of shops during a short period during the day does not need to hold or evaluate beliefs regarding the positions of the planets. Those beliefs could be useful for a different set of plans regarding the optimal time to go stargazing.

This use of a plan library however poses a potential problem with the BDI agent architecture. BDI agents are highly adaptive to changing environments (Silva and Padgham 2004). The hardcoded plan library, combined with the inability to change those plans, can hinder the agent in dynamic environments where the environment can change from that which the agent designers anticipated. The agent can only cope with situations that it has a partial or complete plan for. Partial plans are plans which the agent can combine to form a complete plan (see Partial Order Plans on Page 32).

BDI agents with a hardcoded plan library have no capability to do any form of look-ahead or deliberative planning (Subagdja et al. 2009), nor do they have the capability to deal with scenarios that were not anticipated when the plan library was populated (Meneguzzi et al. 2004). The agents trade off correctness of actions for performance in that they do not have the computational overhead of deliberative planning (Walczak et al. 2006).

In an effort to alleviate this limitation there has been a significant amount of research done on integrating various forms of deliberative planning into agents. This deliberative planning provides an alternative to the reactive planning that the BDI agent architecture provides. These investigations along with some details on general planning will be discussed in the next section.

2.4 Planning

Planning can be said to be the process of constructing a sequence of actions that will satisfy a specified goal, given a specific initial state of the environment and a list of possible actions that can be performed (Weld 1999). Russell and Norvig (2010) describe a planner as a program that searches for a solution. Basic representations used by most classic planners are the STRIPS-like languages.

A planning problem needs states, goals and actions. States describe the current world, or the portion of the world relevant to the planning problem. These could be equated to BDI agent beliefs. Goals describes the desired state of the world. These would equate to BDI agent intentions. Actions have pre-conditions and effects. Pre-conditions are states of the world that must be true before the action can be executed. Effects are descriptions of how the state of the world changes as a result of the action (Russell and Norvig 2010).

The simplest planner is the state-space search. There are two main approaches to a state-space search, namely forward or progression search, and backwards or regression search.

State-space represents a set of states that a world can be in. The set of states forms a directed graph where states are connected by actions. An action connects two states if, by applying the action, the world changes from one state to the other. It is worth noting that this form of planning is essentially a search or optimisation problem. Search algorithms such as A* are complete planning algorithms when applied to a state space (LaValle 2006).

A forward state-space search starts with the initial conditions and searches across all possible states to find a goal state. Forward state-space searches can be inefficient if a large number of the states that it searches through are irrelevant to achieving the goal. This is because it may not be apparent initially which actions are applicable for a particular problem. A large search space, which is common for planning problems, can result in inefficient searches. It is possible, with good heuristics, for forward state-space searches to be efficient (LaValle 2006).

By contrast, a backward state-space search starts with the desired goal and works backwards to the starting conditions. This can be more efficient than a forward search, as only actions that are relevant to the goal need to be considered. To be able to perform a backward search, there must be a known manner to regress from one state to a prior state. Not all planning problems allow for this regression (LaValle 2006).

Forward and backward state space searches both construct what is known as a fully ordered plan. A fully ordered plan is a plan that consists of a sequence of actions. An alternative form of plan is the partially ordered plan.

Partial order planning allows for the problem to be broken down into sub-problems and plans created or selected for the sub-problems. The sub-plans can then be combined, potentially in a number of different orders, subject to the pre-conditions of the sub-plans.

Classical planners tend to make a number of assumptions. Common assumptions are that the world is static, that the planner has full knowledge of the observable environment and that only the agent is changing the environment (Rao 1997; Kambhampati et al. 1995). When agents are situated in a dynamic world, or those agents only have partial visibility, those assumptions no longer match reality. Invalid assumptions may lead to failures in planning. These failures include failure to generate the plan, and generated plans that do not work.

More recent planning approaches, such as decision-theoretic planning extend the classical planners and are capable of planning under uncertain conditions and handling undesirable outcomes and side effects (Blythe 1999).

Two of the planners that have been used with autonomous agents and will be examined here are Hierarchical Task Networks (HTNs) and Graph Planning.

HTNs operate by decomposing complex tasks into networks of more primitive actions with constraints joining them. These primitive actions are directly implementable. The methods of decomposing actions are stored in a plan library. When applied to a high level action, the result of the decomposition is a partially ordered plan comprised of primitive actions that can be executed (Russell and Norvig 2010). Essentially, a HTN can be seen as a search through a constrained space of ways of getting things done, the knowledge about the decompositions being the constraints upon the space.

It is entirely possible a task to have multiple decompositions, each having different pre-conditions that decide which decomposition will be applied. The fewer decompositions that are required for a high level action to be transformed into executable primitives, the more efficient the HTN is (Russell and Norvig 2010).

Graph planning, developed by Blum and Furst (1997), involves converting the planning problem into a graph that can then be navigated. The graph consists of a sequence of levels that represent steps in time. Each level has a set of literals, the state of the environment, and a set of actions that could have their pre-conditions satisfied at that step.

The planning graph starts with the initial state of the environment expressed as a list of literals. Into that are added all the actions whose preconditions are met in the initial state of the environment. Each action is connected to the literals that form its prerequisites. Additional "persistence actions" are added, indicating a literal that persists unchanged to the next level.

The second level of literals contains all the literals that could potentially be the result of any subset of the actions in the previous level. Each of the literals is connected to the actions that caused it. The next set of actions is then the actions whose pre-requisites are satisfied by the new set of literals. Both the action levels and literal levels contain mutual exclusions, designating actions or literals that cannot both occur.

The graph can be expanded until it levels off. Levelling off is when two subsequent levels have exactly the same literals. Once a planning graph has levelled off, any further levels will remain the same and hence there is no point in expanding further. Figure 2.3 shows a fully expanded planning graph for the problem of having your cake and eating it (Russell and Norvig 2010).

The planning graph can be used both for estimating the cost of achiev-

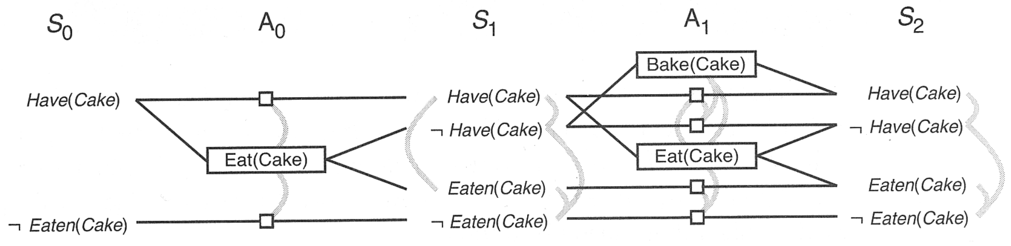

Figure 2.3: Completed planning graph

ing any particular set of literals and for extracting a plan. The cost estimations can then be used by partial order or state-space planners. The GraphPlan algorithm is one that can extract plans directly from the planning graph (Russell and Norvig 2010).

GraphPlan works by adding levels to the planning graph until it levels out and the number of unachievable goals remains static across two of more levels, or until all goal conditions are present in the literals without any mutual exclusions between them. Once that occurs a plan can be extracted by searching backwards up the planning graph (Russell and Norvig 2010).

The full mathematical treatment of the expansion of a planning graph is given in (Ghallab et al. 2004).

In this section the theory of planning and how planning is related to search was discussed. Two planning algorithms were examined in detail. In the next section the integration of planning into agents will be examined using a number of examples from literature.

2.5 Planning and Autonomous Agents

There has been a great deal of research done on the integration of planning mechanisms into autonomous agents. This includes both a number of different planning mechanisms and a number of different agent architectures.

De Silva and Padgham (2004; 2005) wrote a number of papers on their investigation into integrating a Hierarchical Task Network (HTN) planner into a BDI agent.

Their aim was to allow the programmer of the agent to specify places where the HTN planner would be invoked or specific events that would cause the planner to be called. The planner would produce a partially-ordered sequence of primitive actions that the BDI agent would then execute. This allows the developer of the agent to decide which portions of the activities which the agent will be taking would be best served with the traditional reactive planning of the BDI architecture and which requires the additional overhead of the HTN planner.

Meneguzzi et al. (2004) investigated the addition of propositional planning to BDI Agents by using GraphPlan as an external planning algorithm in combination with an extended BDI architecture. They referred to this extended BDI architecture as X-BDI. They added a propositional planning phase to the execution loop placed after the selection of intentions. Unlike in the work done by De Silva and Padgham, the planner would be invoked automatically during every deliberation cycle, whether or not any planning was necessary. This allowed the agent to solve a problem that the unmodified X-BDI agent could not, however it could be computationally inefficient as the planner is invoked automatically, not just when necessary.

Walczak et al. (2006) augmented a BDI agent with a state-based planner. They investigated both the requirements for the planner and its integration with the BDI agent. They distinguished between two main options for the integration of a planner into a BDI agent. The first option is for the planner to generate long term plans and delegate the execution of those plans to a BDI subsystem; the second for a relatively simple planner that is invoked by the BDI agent as necessary to generate short term plans.

The first of those options has a disadvantage when executing in a dynamic environment. The longer the planner spends planning, the greater the chance that the environment would have changed by the time the planner completes. The change of the environment may have invalidated the generated plan. The second option is less susceptible to this issue as its planning cycle is shorter giving less opportunity for the environment to have significantly changed during planning. Susceptibility to the problem of a changing environment depends on how fast that environment changes relative to the time taken for the planner to execute.

Walczak et al. (2006) chose with the second option, integrating a state-based planner at a level below that of the BDI agent itself and allowing the BDI deliberation cycle to choose to invoke the planner rather than selecting a plan from the plan library. This is the option chosen in this dissertation as well. It allows the means-end reasoning to work where possible, keeping the agent responsive and fast and invoking the more complex and slow planning component only when the means-end reasoning fails to cope.

In each of these, the planning engine, when invoked, creates new plans from scratch. The planning process ignores the current set of plans and generate completely new ones. The plan library, however, contains a number of plans for the agent. It may in some cases be more efficient to modify the existing plans than to generate new plans from scratch.

A theoretical study by Nebel and Koehler (1992) came to the conclusion that it is computationally more expensive to modify plans than to generate them. They analysed the complexity of plan generation compared with plan modification using the propositional STRIPES planning framework. They defined plan modification as the process of taking a plan from a library, potentially making modifications, and then applying that plan to the current problem. Given the chosen planning framework, they produced a mathematical proof that shows that plan modification is at least as complex as plan generation. It is however not certain if the result extends to other frameworks and methods.

There are existing systems that implement a form of plan modification. The 3APL agent programming language provides for plan revision rules that can modify plans based on certain beliefs that the agent has. These rules are applied during the deliberation cycle of the agent, after selecting a plan and before executing it. The plan revision in this case does not change a stored plan; it changes which subroutines the plans call (Ten Hoeve et al. 2003; Riemsdijk et al. 2004).

The plans in 3APL are abstract in nature; they specify what must be done and the plan revision rules define how it will be done. This allows for flexibility. While the plan specifies what must be done, it is only at deliberation time that it is decided how it will be done. The plan revision rules also allow for alternative actions if the specified action fails.

The XFRM planner, designed for robots in changing, partially known environments, optimised existing plans based on the observations which the robot makes of the environment (Beetz and McDermott 1994). This is achieved by generating sample execution results for the plan, based on the current environments. The results of the samples are used to determine how good the plan is. The plan is revised based on those outcomes by switching out sections of the plan based upon the reason for the failure. Beetz and McDermott found that while this increased the time required to perform a specific set of actions, the success rate and effectiveness was higher.

Boella and Damiano (2002) investigated the use of a plan-modification strategy with reactive agents. In their study, they designed a system that would decompose a plan into partial plans. These partial plans could then be switched out for others that were more appropriate to the current state of the environment.

2.6 Conclusion

In this chapter the history of autonomous agents was examined, looking at the different agent architectures which were developed over time.

The BDI architecture was discussed in detail, focusing on the plan library component. The reasons for the existence of the plan library were discussed in light of the requirements imposed by the resource-bounded nature of agents. The disadvantages of using a static plan library were discussed in light of the operation of the agent within a changing environment.

From there, the subject of planning was examined, looking at state-space planners, HTNs and the GraphPlan algorithm. Finally, the integration of planning algorithms into BDI agents was discussed and several research studies in that direction were examined.

In the next chapter the theoretical base for GAs will be discussed, as well as their use in planning problems and existing uses within agents.

Chapter 3

Genetic Algorithms

In this section the literature on Genetic Algorithms (GAs) is surveyed by examining the history of Evolutionary Computing and GAs in particular. The GA is examined in detail, looking at the components and steps of the execution cycle.

Finally a number of studies on the use of GAs and other forms of evolutionary computation with Autonomous Agents will be examined and discussed.

3.1 Background

Evolutionary computation collectively describes a number of computing techniques based on natural evolution. The techniques use various computational models of evolutionary processes to solve search and optimisation type problems.

A variety of different evolutionary algorithms exist, with varying purposes and goals. Evolutionary Strategies were originally designed to automatically vary parameters of experiments. Genetic Programming was designed to evolve computer programs. These and other variations use many of the same basic techniques for breeding and evolving potential solutions.

Evolutionary computation can be traced back as far as the 1950s, with the work of Bremermann, Friedberg, Box and others (Back et al. 1997). One of the forms of evolutionary computing, and the one of primary relevance to this dissertation, is Genetic Algorithms.

Genetic Algorithms were first devised by Holland (1962; 1975) as an abstraction of biological evolution. They were later expanded by Bagley (1967). They are effective, robust search and optimisation algorithms based on the biological principles of evolution and survival of the fittest. They have been applied to problems in many areas, including optimisation, economic modelling, ecological modelling, learning and social system modelling (Mitchell and Forrest 1994).

In a traditional GA, it is assumed that a potential solution to a problem can be mapped to a number of parameters. A complete set of values for these parameters is known as an individual. The parameters are then encoded to allow the genetic operators to operate on them. The representation must satisfy two conditions to be useful:

- The complete set of all possible values for the parameters has to cover all of the solution space, or the portion of interest to the problem if the solution space is infinite.
- The application of the genetic operators to a highly performing individual is likely to produce an individual of comparable or better performance (Floreano and Mattiussi 2008, pp. 26–30).

In GAs the representation is usually a fixed-length binary string. There has been work done on alternative, higher cardinality alphabets (Beasley et al. 1993a).

The parameters of the problem are each mapped to a binary string, referred to as a gene. The genes are then concatenated to form a single binary string. This is called a chromosome.

Despite the design of the representation being a step of utmost importance to the effectiveness of the evolutionary algorithm, little theoretical work has been done on designing effective representations (Back et al. 1997).

3.2 Execution cycle

In this section the execution cycle of a GA will be described and the individual selection and fitness concepts discussed.

When a GA begins execution it needs an initial population of individuals that it can operate on. This starting population can be randomly generated or it can be based on a starting condition of the particular problem. The number of individuals is decided on by the specifics of the problem. For example, if a GA was to be used to obtain solutions for the travelling salesman problem, the initial population could contain a set of random routes between the cities defined for the problem.

Each individual in the starting population is assigned a fitness score. This is a measure of how close to the optimal solution of the problem the individual is. The fitness function is designed so that individuals that are closer to the optimal solution have higher fitness values. The fitness score for an individual is determined by running the fitness function defined for the problem upon the individual. Fitness functions will be discussed later in this chapter.

A GA will execute for either a pre-determined number of generations or until the improvement in fitness between the generations falls below a predefined threshold, i.e. very little improvement is seen from one generation to the next. If the improvement in fitness falls below a certain level, the population is said to have converged. At that point the individual with the highest fitness would be chosen as a "good-enough" solution. GAs do not guarantee that the optimal solution will be found.

The execution cycle of the GA is shown in Figure 3.1.

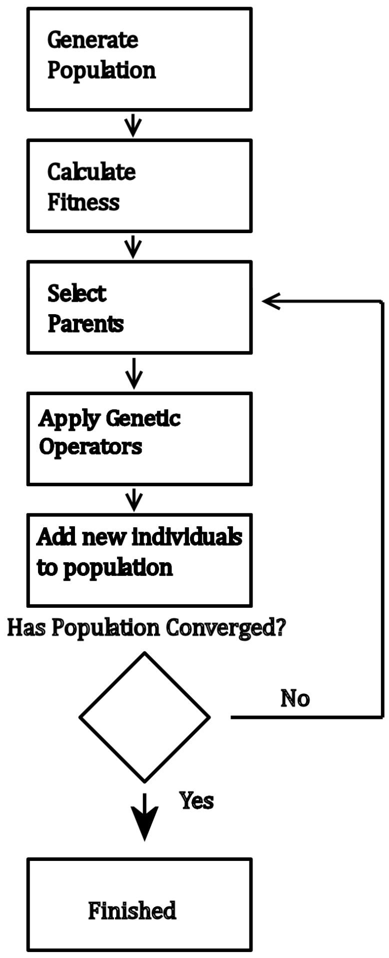

Figure 3.1: GA Execution Cycle

The process of selecting which individuals will be selected to produce the next generation is based on the fitness function defined for the problem. The value that an individual obtains from the fitness function determines how likely it is that the individual will be selected as a parent, and if it is, how many offspring it will have. The fitter the individual, the more offspring it will generally have. The specifics of how and which individuals are selected as parents and how many offspring they have varies from one

GA implementation to another. Some options will be discussed later in this chapter.

Depending on the specific options used for the GA, the individuals created by the genetic operators may replace their parents, becoming the new population or may be added to the population alongside their parents. This process is then repeated for a number of generations. In theory this produces a higher proportion of fit individuals in later generations (Beasley et al. 1993b).

It is possible for a later generation to have a lower maximum fitness than a previous generation. There are variations of the GA that seek to address problems such as these with variations on the selection of individuals to make up a population. For example only replacing a parent with its offspring if the offspring has a higher fitness (Beasley et al. 1993a).

A population is said to converge when, over successive generations, the fitness of the individuals tends towards a single value. A population converging is considered a success in GAs. A gene is said to have converged when 95% of the population share the same fitness (De Jong 1975).

This convergence is accompanied by a loss of genetic diversity. This may be acceptable in most scenarios. If using GAs to repeatedly identify optimal solutions in a changing environment when the fitness function is a time-dependent function it may be more of a problem. The loss of genetic diversity hinders the ability of the GA to react to changes. One possible solution is to not discard the low fitness individuals from the population. This would keep the genetic diversity high and allow the GA to operate even in changing environments (Gaspar and Collard 1999). Another option would be to increase the mutation rate so that mutations could replace lost genetic diversity.

3.3 Genetic Operators

The two genetic operators that are mainly used in GAs are Crossover and Mutation. In this section the two main genetic operators will be discussed and their purpose explained.

3.3.1 Crossover

Crossover, also called recombination, is the process of combining the information from two or more parents to produce new individuals that have some of the characteristics of their parents. The theory is that the offspring may have favourable traits, in terms of the fitness function, from both parents, especially if the evolutionary algorithm used only allows individuals with high fitness levels to become parents (Back et al. 1997; Beasley et al. 1993b).

The chromosomes of two individuals are split at a single point. The two resulting portions of each individual, called the head and the tail, are switched to form two new individuals. The point at which the split is done is usually chosen at random.

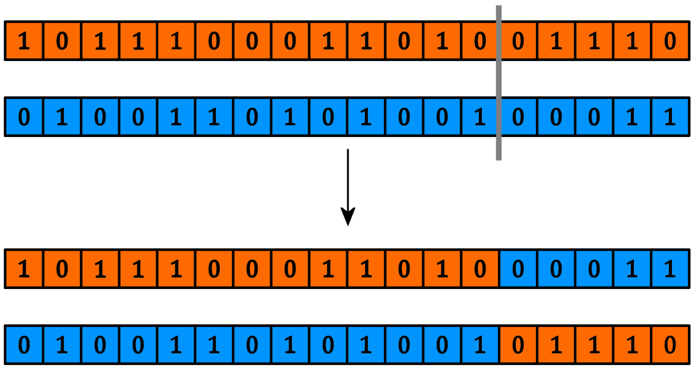

Figure 3.2: Simple crossover

The most common crossover is the single-point crossover, shown in Figure 3.2. There are other forms of crossover including multi-point crossover and uniform crossover (Beasley et al. 1993a).

With multi-point crossover two or more points are chosen at which to split the string and the resulting portions crossed over. With uniform crossover a bitmask determines which bits will be taken from which parent to form the new individual. Multi-point crossover and uniform crossover are illustrated in Figures 3.3 and 3.4.

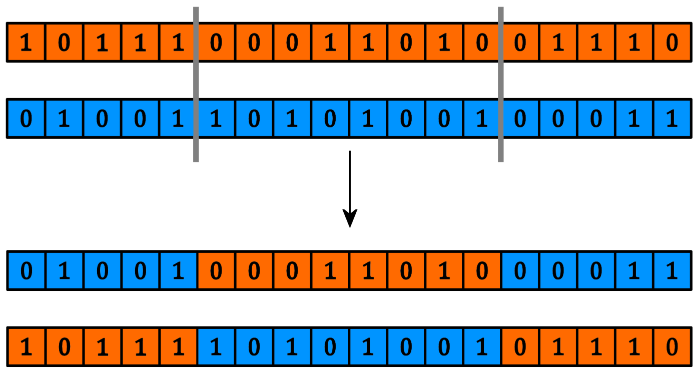

Figure 3.3: Multi-point crossover

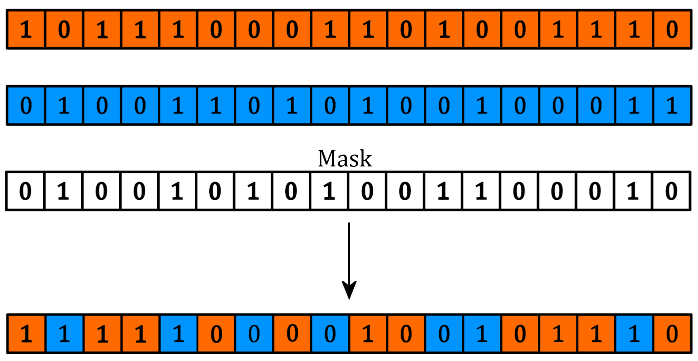

Figure 3.4: Uniform crossover

An analysis of the effects of different forms of crossover was done by De Jong and Spears (1991). They concluded that in larger populations less disruptive forms of crossover, such as single and double point crossovers, work well, while in smaller populations there is some benefit to using multi-point crossover with more than two points or uniform crossover as it helps overcome the limitations of genetic diversity in smaller populations.

3.3.2 Mutation

Mutation is the other major genetic operator used in GAs. The main purpose of mutation is to add novel individuals to the population, that is individuals that could not be created by recombination (Back et al. 1997). In natural evolution, mutation is the instrument of change, creating individuals that could not be created by crossover. Mutation has a similar role in GAs.

Mutations are usually applied to a very small percentage of the individuals, typically around 0.1% (Beasley et al. 1993b). A study by Hesser and Männer (1991) found that the optimum mutation rate depends on the population size. The larger the population, the smaller the rate of mutation should be. Stanhope and Daida (1998) investigated crossover and mutation rates in dynamic environments and showed that for a faster changing environment, a higher mutation rate can result in a more efficient GA.

With a binary string, the simplest mutation is a switch of one bit of the string somewhere along its length. Both the individual selected for the mutation and the position within the binary string are typically chosen at random. An example of a single-bit mutation is shown in Figure 3.5.

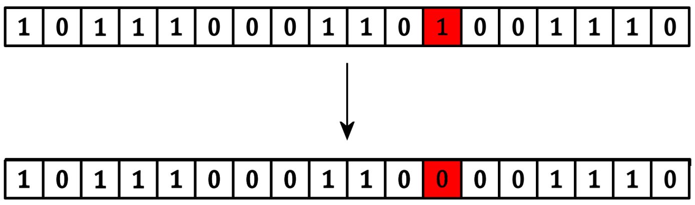

Figure 3.5: Mutation

3.4 Encoding and Fitness Function

Two of the most crucial choices around implementing a GA are the design of the representation, or encoding, and the design of the fitness function. The encoding is the method by which the problem is converted into a binary string suitable for the genetic operators to be applied to. A good encoding will give similar individuals a similar encoded form so that they are nearby in the solution space (Floreano and Mattiussi 2008).

A poor representation can make the problem of finding an optimal solution much harder. For example, a representation where two similar chromosomes map to vastly different points on the solution space can result in the crossover of two high-fitness individuals producing a new individual that represents a completely different portion of the solution space and is of much poorer fitness. This hinders the ability of a GA to narrow in on the optimal areas of a solution space (Moscato 1989).

The design of the fitness function has two goals. The function must return a deterministic value for each individual in the population indicating how close to optimal that individual is in terms of potential solutions to the problem. The function should use as few resources as possible to execute as its execution time is one of the main factors that affect the performance of a GA.

An inefficient fitness function can result in excessive execution times for the GA. In a standard GA design, the fitness function is executed at least once per individual per generation. Hence we can say that the execution time of the GA, t_{GA}, is bounded below by the sum of the times of all the executions of the fitness function, as described in Equation 3.1, where G is the total number of generations, P is the population size and t_f is the execution time of the fitness function.

$$t_{GA} = \sum_{i=0}^{G}(Pt_f) \tag{3.1}$$

Hence, purely from an execution time perspective, smaller populations are more efficient. The counter-argument to this is that smaller populations have less genetic diversity. Lower genetic diversity within a population can be partially overcome by having higher mutation rates (Hesser and Männer 1991) or by using crossover forms such as uniform or multipoint crossover (De Jong and Spears 1991).

In this section the execution cycle and components of a GA were described. In the next section the uses of GAs for planning purposes will be examined. The kinds of problems they are suitable for will be described and a number of papers on the use of evolutionary computing for planning will be discussed.

3.5 Genetic Algorithms and Planning

Genetic Algorithms are at their core an optimisation technique. One of the key differences between GAs and other commonly used search algorithms, such as random walk, hill-climbing, gradient descent or simulated annealing, is that GAs are population based. Instead of searching the entire solution space one solution at a time they focus towards optimal choices, essentially performing a directed search. They are not as susceptible to the problem of local maxima as hill climbing techniques and they tend to be more efficient than random walk techniques (Keane 2001; Goldberg 1989).

As a form of optimisation algorithm, GAs can be used in planning problems. Planning can be expressed as a specialised form of optimisation: that is, seeking the best series of actions to achieve some goal. LaValle (2006) describes a variety of search algorithms as ways to solve simple planning problems. He defines simple planning problems as ones where the possible states of the planning problem can be formulated as a finite or infinite graph. An additional component of the definition is that the search algorithm is systematic, able to evaluate all states or, in the case of an infinite graph, able to evaluate in finite time that there is no solution.

To achieve the aforementioned conditions requires:

1. The planning problem can be represented by a series of values which will become the genes.

2. The optimality of a potential solution can be computed, which is necessary for the fitness function.

Ahuactzin et al. (1991) used GAs to solve the path-planning problem for both a robotic arm with two degrees of freedom and a mobile robot within an environment containing moving obstacles. By expressing the path planning problem as an optimisation problem it became a problem perfectly suited to GAs. Their solution was both correct, in that it found acceptable paths, and of sufficient speed for the desired application.

Another use of GAs for motion planning was by Sugihara and Smith (1997). They utilised a GA to do both path and trajectory planning for an autonomous mobile robot. They investigated both off-line and online planning options. They encoded the paths as pairs of direction and distance in the x and y directions, thus enabling them to use fixed length binary strings for the GA. They found that the GA identified an optimal path in the majority of their tests.

3.6 Evolutionary Computing and Agents

There has also been prior work done on the use of GAs, or other forms of evolutionary computing, to evolve agent architecture or behaviour.

Calderoni, Marcenac, and Courdier (1998) used Genetic Programming in order to evolve dynamic behaviour in agents. Each evolved behaviour was then partially evaluated against the current state of the environment to determine the relevance and usefulness of the evolved behaviour. Their foraging robot experiment showed a promising increase in the efficiency of the robots over time as compared to hardcoded behaviour. This allowed the robots to adapt to a dynamic environment.

Dellaert and Beer (n.d.) did similar research into evolving complete agents, although instead of evolving the behaviour of the agent, they evolved a genotype that a development process could then grow into an agent. They investigated two different models, one simplified and one closely following biological processes. With both the complex and the simple models, they had notable success in evolving agents to execute complex tasks in a static environment. With the complex model they needed to start out with a hand-crafted genome as they were unable to evolve an agent from scratch. They were able to evolve an agent from scratch with the simplified model.

Nonas and Poulovassilis (1998) applied GAs to a variant of the BDI agent architecture called Active Databases. Active Databases are databases that include an event-driven system to respond to internal and external conditions (Paton and Díaz 1999).

In their experiments they used a simulated network consisting of nodes with connections between them. This is superficially similar to the network used in this dissertation however designed for a different purpose. The network used in this dissertation is a construct to send messages across. The network is a connected set of nodes that provide a number of services. Each node has a single service it could offer and an agent running on that node. The agent's goal is to respond to a requested set of services as efficiently as possible, using nearby nodes to provide the services which it does not provide itself.

In the study, the GA operated as part of the deliberative cycle. It was capable of selecting intentions. This is different from the work in this dissertation, where the GA is used to alter the plans stored in the plan library.

3.7 GAs and network routing

The environment chosen in this dissertation, for the testing of the GA plan modification component, is that of a simulated communication network. The purpose of this dissertation is not however to provide a new routing algorithm for networks, nor to compare results with existing routing algorithms. The common routing algorithms are discussed below for completeness.

The shortest path algorithms are well regarded for network routing and have been used for many years. The commonly used algorithms for distributed networks are the Bellman-Ford algorithm and Dijkstra's algorithm (Medhi and Ramasamy 2010). These are both shortest path routing algorithms.

Despite the popularity of shortest path algorithms, GAs have also been used for network routing.

Ahn and Ramakrishna (2002) use a GA for shortest path routing in an ad-hoc wireless network. They define the chromosome representation to be the sequence of nodes through which the routing path passes. As such, their chromosome is of variable length. Their fitness function calculates the cost of the network path and represents the fitness of each individual as the inverse of that cost. They use a variation on the normal crossover where two individuals can only be selected for crossover if they have a partial route in common. Additionally the crossover point can only be at the boundaries of the genes. They showed that their GA converges to the route found by Dijkstra's algorithm and has a lower failure ratio on ad-hoc networks than the GA solutions that they compared against.

Nagib and Ali (2010) used GA in a network routing protocol. Their GA has performance comparable with Dijkstra's algorithm while returning the same result. The encoding they use is the same as that used by Ahn and Ramakrishna use.

3.8 Conclusion

In this chapter the history and evolution of GAs have been discussed. Their components and execution cycle were discussed along with the different genetic operators which they use.

The use of GAs within planning and within agents were examined, giving details of how GAs, which are at their core an optimisation or search technique can be used in planning problems and what requirements there are for GAs to be used in planning. Additionally previous studies which used GAs for planning were described.

The use of GAs within network routing problems was also examined. While this dissertation is not concerned with new network routing algorithms, the environment used is one of a simulated communications network and hence a brief look at how GAs have been used for network routing is required.

In the next chapter the design for the experiments will be described, including the environment, the agents used and the method for integrating the GA plan modification mechanism into the BDI Agent.

Chapter 4

Research Design

In this chapter the design of the simulated network environment and the two BDI agents used for the experiments are discussed. The two agents are referred to as the Messaging Agent (MA) and the Environment Agent (EA). The MA is the agent which is modified to include the GA plan modification mechanism. The EA is the agent which operates on the environment to enact the changes to it.

The design of the GA plan modification method is described in detail, along with the method of integrating it into the execution cycle of the BDI Agent.

4.1 Methodology

The methodology chosen for this dissertation is experimentation. The experimentation methodology is a form of empirical research, involving trials and tests.

Mouton (2001) describes experimental research as research which answers causal questions. It operates on a small number of cases in highly controlled conditions.

Experimental research in Computer Science has three possible goals (Olivier

2009):

1. To explore an area in hopes of finding something interesting.

2. To test a theory.

3. To prove a theory.

The first of those, exploratory experimentation, involves running experiments within an area to see whether further investigation is warranted. The second revolves around a limited set of experiments to see whether a theory holds true in specific cases. The third is concerned with investigating whether there are any cases where the theory does not hold true.

Tichy (1998) states that experimentation is used for testing theories against reality and for exploring areas where theory does not yet reach . He also mentions that when testing theories against reality, the goal is to see if there are any areas where the theory fails.

The experiments done in this dissertation fall into the category of a limited set of experiments to see whether a theory holds true. The goal for the experiment described in this dissertation is to discover whether, in a specific scenario, GAs can be a feasible plan modification mechanism. This is done by running a set of experiments in controlled conditions to determine whether the theory holds.

The experimental methodology is appropriate for this study as the goal of the research is to determine whether or not the inclusion of a GA plan modification component can improve performance of a BDI agent operating in a dynamic environment. Specifically this is the testing of a theory.

The experimental methodology allows for the performance of the BDI agent to be evaluated both without and with the GA plan modification component and two sets of data evaluated and compared.

4.2 Simulation design

In this section the design of the experimental system will be discussed along with the motivation behind that design. Emphasis will be given to the GA planning component and how it integrates with the BDI agent.

The system created for the experiments, referred to as the System henceforth, consists of a simulated communications network across which messages of various priorities and sizes are transmitted. The network environment is dynamic. The latency on the nodes and links changes over the course of the experiment. In addition nodes and links can be disabled or enabled.

The programming language that the prototype is developed in is Java. The Java language is used because of the freely available agent and GA libraries for it. Examples are given below.

BDI Agent Frameworks, as listed in (Mascardi et al. 2005) include:

- JACK (Busetta et al. 1999)
- JAM (Huber 1999)
- Jadex (Pokahr et al. 2003)
- 3APL (Ten Hoeve et al. 2003)

Genetic Algorithm Frameworks include:

- JGAP (Meffert et al. n.d.)
- ECJ (Luke et al. n.d.)
- JCLEC (Ventura et al. 2008)

The agent framework chosen for this experimental system is Jadex (Pokahr et al. 2003), a BDI agent framework built on top of the Jade (Bellifemine et al. 2001) agent library. The GA library used is JGAP (Meffert et al. n.d.).

In Jadex, an agent definition file, in the XML format, defines the beliefs, goals and plans of the agent. The beliefs are defined as Java data types, either individual values or collections of values. These can be built-in types or custom classes. The goals contain conditions that trigger the activation and deactivation of the goal. These conditions are based on the beliefs. The plans reference Java classes and each goal can have one or more plan associated with it. A simplified agent definition is shown in Listing 4.1.

```
<agent xmlns= "http:\\jadex.sourceforge.net/jadex"
        xmlns:xsi= "http:\\www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation= "http:\\jadex.sourceforge.net/jadex/jadex-0.96.xsd"
        name= "EnvironmentAgent"
        package= "Experiment">

        <imports>
                <import>jadex.runtime.*</import>
                <import>jadex.util.*</import>
                <import>jadex.*</import>
                <import>Java.util.Date</import>
        </imports>
        <beliefs>
                <belief name="gui" class="NetworkGUI">
                        <fact>
                                new NetworkGUI(\$agent.getExternalAccess())
                        </fact>
                </belief>
                <belief name="NetworkWidth" class="int"
                        exported="true"/>
                <belief name="MaxLinks" class="int"
                        exported="true"/>
                <belief name="GAStatus" class="int"
                        exported="true"/>
        </beliefs>

        <goals>
                <performgoal name="performUpdateCounter"
                        retry="true" exclude="never">
                </performgoal>
                <performgoal name="performChangeEnvironment">
                </performgoal>
        </goals>

        <plans>
                <plan name="UpdateCounter">
                        <body class="PlanUpdateCounter"/>
                        <trigger>
                                <goal ref="performUpdateCounter"/>
                        </trigger>
                </plan>
                <plan name="ChangeEnvironment">
                        <body class="PlanChangeEnvironment"/>
                        <trigger>
                                <goal ref="performChangeEnvironment"/>
                        </trigger>
                </plan>
        </plans>
</agent>
```

Listing 4.1: A simplified Jadex agent definition file

The agent described in Listing 4.1 has four beliefs, one defined as a custom Java class and three as integer values. It has two goals, each goal having a single plan that can be used to achieve that goal.

The experimental system consists of two main parts, the Environment Agent (EA) and the Messaging Agent (MA). The EA simulates nature and external factors and changes the characteristics of the network over time. It is also responsible for generating messages for the MA to send.

The job of the MA is to deliver messages across the network, from the source node S to the destination node D.

4.2.1 Simulated Network Design

The simulated network has a start node S and destination node D, connected to a grid of $N \times N$ interconnected nodes in a directed graph. The network is arranged into N rows and N columns. Node $(2, 3)$ represents the node with $x = 2$ and $y = 3$, as highlighted in Figure 4.1. The numbering starts at 0. In a real network the nodes would be routers, switches or servers. The links which connect the nodes represent network connections.

The experiments used $N \in \{32, 64, 128\}$, corresponding to network grid sizes of 32×32, 64×64 and 128×128. An example network with $N = 8$ is shown in Figure 4.1.

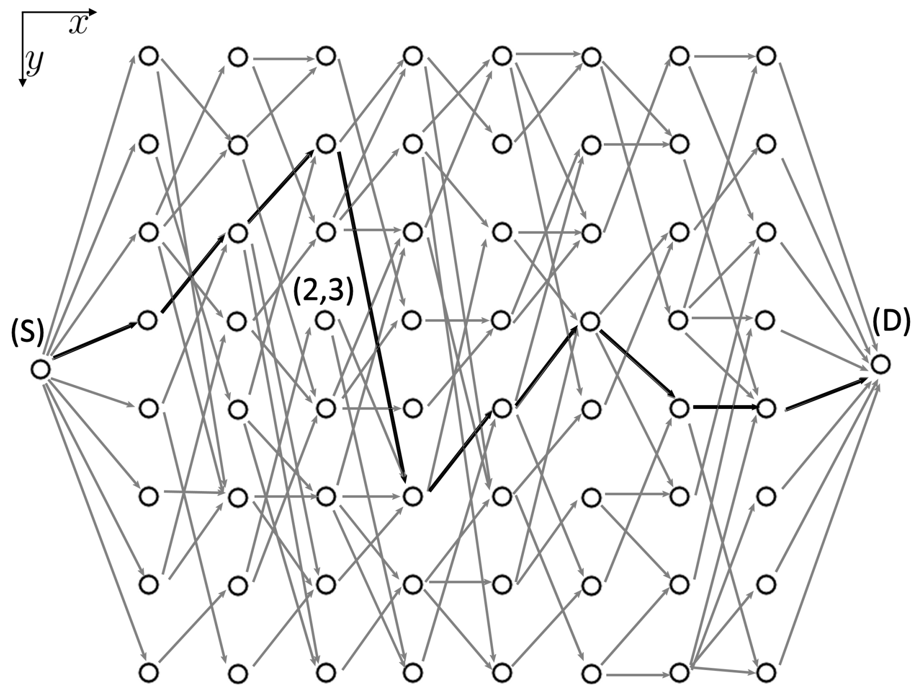

Figure 4.1: Example 8x8 grid

Each node, other than D, is linked to at least one node in the next row. There are no reverse links.

The network is not fully connected. There is a maximum number of output links, with m allowed per node. This value is smaller than the total number of nodes in the next column. The starting node S is the only node that does not have this limit. The starting node links to all nodes in the first column. This number is based upon the size of the graph. The max number of links per node is m, where $m = N/4$. This value for m

was chosen to limit the number of paths through the network. Limiting the paths means that there are fewer opportunities for the MA to route around nodes affected by the changing environment. A fully connected network would be more resilient to changes to the environment as there are more alternate routes which can be chosen.

Each outgoing link is given a number, an ordinal value, starting at 0, which uniquely identifies the link within that specific node. This is illustrated in Figure 4.2.

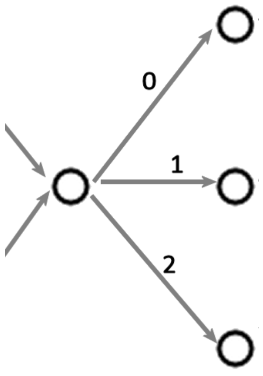

Figure 4.2: Link numberings

The nodes and links are both characterised by a Latency variable L and a Reliability variable R. Additionally each node and link can be in an enabled or disabled state. When disabled, no messages can be routed through it. The start and destination nodes cannot be disabled, neither can the links from or to the start or destination nodes.

When the network is created, the latency on the nodes is set to a random value between 0 ms and 100 ms and the latency on the links to is set to a value between 0 ms and 1000 ms as detailed in Listing 4.2. These figures were chosen so that there is scope for the values to change in both directions during the course of the experiment. All nodes and links are enabled.

```
nodeLatency = (int) Math.round(rand.nextDouble()*100);
```

```
linkLatency = (int) Math.round(rand.nextDouble()*1000);
```

Listing 4.2: Latency starting values

4.2.2 Environment Agent

A BDI Agent, defined as the EA acts upon the environment. It changes the properties of the nodes and links and potentially disables or enables them. This ensures that the picture that the MA has of the environment is not completely accurate and that any routing plan that the agent comes up with will not remain optimal.

The EA, since it is implemented as an agent, runs in parallel with the MA. This is intentional. Doing so allows for the environment to change while the MA is executing and, once the GA plan modification mechanism is added, while the GA is evolving plans.

The EA has a permanent goal which increments an internal counter then sleeps for 200 ms. The counter is solely used to determine when the environmental modification routines run. Every time that this goal runs, the EA checks a random number, generated using the `java.util.Random` class, against a constant chance to modify the environment, defined as c_O. The value for c_O is set to 0.01 in all the experiments. Hence each time the internal counter is incremented there is a 1% chance that the environment will be changed.

The routine invoked when the environment changes loops over all of the nodes in the network. For each node it generates a random number using the `java.util.Random` class and checks that generated random number against the base chance for the node to be modified, defined as c_N. The c_N is hardcoded and based upon the size of the network.

For a network with $N = 32$, $c_N = 0.6$. For a network with $N = 64$, $c_N = 0.4$. For a network with $N = 128$, $c_N = 0.2$.

To summarise, when the environment change routine runs, each node in the network has a chance equal to c_N to be modified at that time.

These values were chosen based on initial tests into the effects of the changing environment on the performance of the MA. The values were chosen so that the degradation of the performance of the MA as the environment changed was similar across the three network sizes. The second reason was to ensure that the failure rate on the larger networks did not increase, during the test, to the point where useful data could not be obtained.

If a node passes the random check, the latency of the node is modified, as per Listing 4.3, lines 1 and 2.

The EA then generates a second random number and compares that random number with the value of R for the node to see if the node should be disabled or enabled, as shown in Listing 4.3, lines 4 to 12.

```
nodeLatency += (int) Math.round((rand.nextDouble-0.5)*80);
nodeLatency = Math.max(nodeLatency, 0);

chanceOfDisable = rand.nextDouble*1000;

if(!currentNode.isEnabled() && chanceOfDisable < currentNode.Reliability())
{
    currentNode.enableNode();
}
if(currentNode.isEnabled() && chanceOfDisable > currentNode.Reliability()) {
    currentNode.disableNode();
}
```

Listing 4.3: Node changes

The changing environment means that the plans included in the agent may not be appropriate at any given time and hence some form of plan modification or complete plan recreation will be necessary to keep the operations efficient.

4.2.3 Messaging Agent

A BDI Agent, which will be referred to as the MA, has the job of sending messages across the network from the source node S to the destination node D. The goal of the MA is to deliver messages as quickly as possible.

This goal is hindered by the EA which intermittently alters the characteristics of nodes and links, possibly resulting in messages being discarded or delayed. If the EA disables a node or link, messages sent to that node or sent across that link are discarded and that message delivery is considered a failure. If the EA increases the latency on a node or link, messages sent to that node or across that link take longer to be delivered.

The MA cannot see the entire network, hence it cannot tell the status of the entire network at any point in time. The partially observable environment means that the MA cannot work out the perfect route at the time of sending a message and must rely on information that was obtained from previously sent messages.

There is a single agent delivering messages. It would certainly be possible to have multiple agents, each with their own plan libraries and each with different configurations of the GA; however that will be left for future work.

The MA has access to a collection of plans, in the form of the BDI Agent architecture plan library. These plans describe how to achieve the goals which the agent has; that is to route the messages in order to deliver them as quickly as possible.

Initially, this set of plans is static and all that the agent is able to do is choose between them. This is the normal behaviour for a BDI agent and will give a baseline for the performance of the agent. Within the context of the messaging simulation, the performance is based upon how long it takes for a message to be delivered. Shorter delivery times means higher performance.

The time taken to deliver a message (t_{msg}) is the total latency of all the nodes and links that the message passes through and can be expressed as shown in Equation 4.1, where L_n is the latency of the node and L_l is the latency of the link.

$$t_{msg} = \sum_{i=1}^{N} L_{n_i} + L_{l_i} \tag{4.1}$$

Once this baseline is established, the GA plan modification component will be added and the performance again examined. The difference between the performance without the GA and the performance with the GA will be the measure for determining how successful the GA plan modification mechanism is.

To further test the hypothesis, additional tests will be run with different settings for the GA and the environment. The full list of all experiment configurations is detailed in the next chapter.

4.2.4 Plan design

In Jadex plans are implemented as Java classes. The class inherits from the `jadex.runtime.Plan` class and must, at a minimum, implement the `body()` method. This method is the core of the plan; the actual actions that will be executed by the agent to achieve its goals.

To facilitate the evolution of these plans, they are abstracted as a delimited string and stored within a plan cache object. The execution of a plan involves decoding the string and following the directives contained within. Each element corresponds to a specific block of code; in this case defined as a Java function.

As an example, a plan to obtain a cup of coffee could be specified as:

`fillKettle; boilKettle; fetchCup; addCoffeeToCup; addSugarToCup; addWaterToCup; addMilkToCup; stirCup`

An alternate plan to achieve the same goal could be:

`goToCar; driveToNearestSteers; buyCoffee; driveHome`

Each of those elements of the string (`fillKettle`, `buyCoffee`, etc.) would then map to a Java function that would implement the logic for that small fragment. This way the plan can be evolved out of small fragments of code. There will be combinations of elements that will make no sense together. It will be one of the jobs of the fitness function to weed these combinations out by assigning them a low fitness.

```
public void body() {
        CachedPlan executionPlan = Cache.FetchActivePlan();
        boolean success = executionPlan.Execute();

        if (!success) {
                \\ fail the goal.
        }
}
```

Listing 4.4: Plan Class definition

With the plan stored as such strings, the Java Plan classes can be reduced to code shown in Listing 4.4.

The Execute method of the plan would be responsible for breaking up the string representation of the plan, interpreting each portion and taking the appropriate actions in order to execute it.

Consider the simplified network of 64 nodes, plus start and destination, in an 8×8 grid shown in Figure 4.1.

With that network structure a plan for the first option discussed above appears as follows: `Link3; Link0; Link0; Link1;Link1; Link1; Link1; Link0; Link0`

This plan is read as: from the starting node use link 3, from the node in Row $x = 1$, use link 0 and so on. The route described by this plan is shown in bold in Figure 4.1.

Given this plan form, the `Execute(plan)` method converts the plan to its string representation, splits the string into an array and then executes each element of the plan sequentially, as shown in Listing 4.5.

```
private boolean Execute(CachedPlan executionPlan) {

        Boolean success = false;
        String[] StrArray = executionPlan.getStringRepresentation().split(";");
        for (int i=0; i\textless StrArray.length; i++) {
                if (StrArray[i] == "dispatchMessageLink0")
                        success = dispatchMessage(0);

                if (StrArray[i] == "dispatchMessageLink1")
                        success = dispatchMessage(1);

                if (StrArray[i] == "dispatchMessageLink2")
                        success = dispatchMessage(2);

                if (!success)
                        break;

        }
        return success;
}

```

```
public Boolean dispatchMessage(int LinkNo) {
        Node currentNode = msg.getCurrentLocation();
        \\ call dispatch on the node
        return currentNode.dispatch(msg, LinkNo);
}
```

Listing 4.5: Execute method of the Plan class

In this section the design of the plan has been discussed as well as how the plan is executed by the Jadex agent. In the next section, the design of the GA plan modification component is discussed showing how the plan design facilitates the operation of the GA on the plan.

4.2.5 Plan Library Design

The plan library is implemented as a Java `HashSet`.

When the network is generated, the plan library is populated with a starting set of plans. The initial set consists of K_{init} plans where $K_{init} = 4N$. For the example 8×8 network shown in Figure 4.1, $K_{init} = 32$. These plans are generated at random. The code which generates the starting set of plans is shown in Listing 4.6.

```
private void generatePlanCache() {
        Cache.PlanCache = new HashSet<CachedPlan>();

        for (int i=0; i<Environment.NetworkSizeNodesPerRow*4;i++) {
                PlanCache.add(generateRandomPlan());
        }

        initialSize = PlanCache.size();
}

private CachedPlan generateRandomPlan() {
        CachedPlan randomPlan;
        int transmissionDelay, minBandwidth;
        String initialPlanStringRef = "";
        Node CurrentNode;
        int i;

        i = (int) Math.random()*Environment.NetworkSizeNodesPerRow;

        initialPlanStringRef = "dispatchMessageLink" + i + ";";
        transmissionDelay = 0;
        minBandwidth = Integer.MAX_VALUE;
        CurrentNode = Environment.Network.get("Node0x" + Integer.toString(i));
        while (!CurrentNode.IsDestinationNode()) {
                i = (int) (Math.random()*CurrentNode.GetAllConnections().size());
                initialPlanStringRef = initialPlanStringRef +
                        "dispatchMessageLink" + i + ";";
                transmissionDelay = transmissionDelay +
                        CurrentNode.totalLatency() +
                        CurrentNode.getLink(i).transmissionDelay;
                minBandwidth = Math.min(minBandwidth,
                        CurrentNode.availableBandwidth());
                        CurrentNode = CurrentNode.getLink(i).Destination;
        }

        randomPlan = new CachedPlan(0);
        randomPlan.setStringRepresentation(initialPlanStringRef);
        randomPlan.updateStats(transmissionDelay,minBandwidth,true);

        return randomPlan;
}
```

Listing 4.6: Generation of initial set of plans

The plans generated at the start is not a complete set of all possible plans. The network contains between $N^2(N-1)$ and $mN^2(N-1)$ possible paths, where N is the number of nodes per row and m is the maximum number of links per node.

The number of possible paths is far higher than the number of plans. Having plans for every possible route is possible in this scenario but would not be in general. Even in this scenario, having a plan for every possible route would make choosing a plan from all the options a non-trivial operation.

4.3 Genetic Algorithm Design

In this section the design of the GA and how it integrates with the plan design will be explored. The encoding and fitness function will be discussed in detail.

4.3.1 Encoding

It is necessary to encode the delimited string, described in the Section 4.2.4, so that a standard GA can operate on in. This is done by taking the ordinal value of the link to send the message on, expressed in binary, as the binary representation. If the plan is `Link4; Link0; Link0; Link1; Link1; Link1; Link1; Link0; Link0`, the encoded plan would then be `100000000001001001001000000`. This would describe the route in bold in Figure 4.1.

In the case of the sample network shown in Figure 4.1, the maximum number of links possible from a single node is 8. The start node (S) alone can have that many. As such, a 3-bit string is necessary and sufficient to encode all possible options. For larger networks such as the network with $N = 128$ used in the experiments, a longer string would be required to hold all the possible values. An $N = 32$ network requires a 5-bit string, an $N = 64$ network requires a 6-bit string and an $N = 128$ network requires a 7-bit string.

The 3-bit string for each link allows for up to 8 links from a single node. This is more than any node other than the start node can have. In these cases, illegal plans, i.e. ones that reference links that do not exist, are weeded out by the fitness function. It does this by assigning any plan with such illegal elements a fitness of 0.

4.3.2 Fitness function

The job of the fitness function is to give the individuals in the population a value that allows them to be ranked in order of effectiveness. With the dynamic environment that this GA acts in, the fitness function cannot be a static function. It must be based on the state of the environment at the time that the GA runs.

There is more than one possible design for a fitness function for the configuration described so far.

The simplest is a fitness function that simply tests each plan on the real network or on a cached simulacrum of the network if that is not possible. The fitness function tests a route to obtain the latency and validity of a route. A route is considered invalid if any of the nodes or links it uses are in a disabled state. This is the option that will be used for these experiments.

In these experiments the actual network will be used because it is already a simulation. In a real network, this would not be possible and the fitness function would have to operate against a simulacrum of the network that has the last known properties of the network. Depending on how accurate those last known properties, are the plan modification may be less efficient than in the experiments described in this dissertation.

More complex fitness functions could involve the creation and maintenance of a representation of the network that includes both the last known statistics and the historical trends in order to attempt to predict future behaviour. With this, a fitness function can consider both the current state of the network and the historical behaviour. This option for a fitness function will not be investigated in this experiment, but could be considered for future work. The fitness function used in these experiments is shown in Listing 4.7

```
private double PlanCostLatency(CachedPlan plan) {

        boolean isWorking = true;
        int totalLatency = -1;
        String[] PlanRepresentation = plan.getBinaryRepresentation();
        if (!isValid(PlanRepresentation))
                isWorking = false;
        else
                totalLatency =
                        Environment.EvaluateRouteLatency(PlanRepresentation);

        double fitness;
        if (!isWorking)
                fitness = 0;
        else
                if (totalLatency < = 0)
                        fitness = 0;
                else
                        fitness = (1/(double) totalLatency)*10000;

        return fitness;
}
```

Listing 4.7: Fitness Function

The fitness, F, is constrained as $0 <= F$. $F = 0$ indicates that the evaluated route is not valid or does not exist. If the route is valid, the fitness is calculated as shown in Equation 4.2. Higher fitness values are better.

$$F = 10000.(frac1t_{msg}) \tag{4.2}$$

4.3.3 Commitment

The decision as to when the GA is invoked is of utmost importance. If the GA is invoked too frequently, the system spends all its time evolving and does no useful work. If the GA is invoked too infrequently, the system exhibits poor performance due to unnecessarily poor plans.

For the purposes of this study, a simple trigger is used for invoking the planning for the initial tests. If the MA records three failures then the GA is invoked. This value was chosen as balance between invoking the GA on every failure which may be too often and invoking it too seldom. A failure is a message not delivered because the route it uses contains a disabled link or node.

An additional test is done where the invocation of the GA was invoked on a regular basis, therefore decoupled from the outcome of the plans.

4.3.4 Retention

A changing environment is not ideal for a traditional GA. A GA typically works by retaining high performance individuals and discarding low performance individuals hence improving the overall fitness of the population. This works well for static environments which is typically where GAs are used. In a dynamic environment, that discarding of low performance individuals inhibit the long term usefulness of the GA, as individuals that show low performance at one point in time may show high performance at a later time.

One of the mechanisms that can be used to avoid this is to not discard the low-performing individuals from the population, but rather retain them in case they become useful in the future. This is the approach taken by Gaspar and Collard (1999).

In the initial design for this study, the starting population is the entire contents of the plan library. As will be discussed in Section 5.4, doing so proved impractical due to the increased execution time of the GA.

In the revised design, each time the GA is invoked its starting population is drawn from a random sample of the plans existing in the plan library. Because no plans are ever discarded from the plan library, this allows the GA to have the necessary genetic diversity to produce good plans without needing to change the basic implementation of the GA.

The size of the sample is constant over the duration of the experiment, and therefore the GA execution time also remains constant over the duration of the experiment.

4.4 Criteria

The criteria for invoking the GA can be chosen based on the exact scenario and the requirements of the BDI agent. If we take as an example a BDI agent with the goal to send messages quickly then there are a number of possible triggers for invoking the GA:

- When the message transmission time increases by more than a threshold amount.
- When the current plans fail or there have been more than a threshold number of failures since the last GA execution.
- On a schedule on the assumption that there is always a more efficient plan.
- Run asynchronously and continually, adding new plans to the plan library all the time.

In this study, the second of those options is the criteria used.

4.5 Conclusion

In this chapter the design for the experiments was described in detail. The chosen frameworks and language for the experiment were listed.

The chosen environment for the experiment, that of a simulated communications network, was detailed, its properties listed and its starting condition described.

The two BDI agents that operate within the simulated environment were described in detail. Their behaviour and goals were listed and the performance of the MA was defined.

The plan design was explained, showing how a plan can be constructed in such a way as to allow for a GA to evolve it. The design of the GA, including fitness function, encoding is explained, along with the options for invoking the GA plan modification mechanism.

In the next chapter the results of the experiments will be discussed and analysed.

Chapter 5

Results

The core question in this dissertation is whether the use of a GA to evolve the plan library of a BDI agent provides any benefit for message routing in a changing environment and, if it does, under what circumstances.

This question is answered in this chapter by examining the performance characteristics of a BDI agent in a simulated network environment and analysing the differences in performance between the tests which do not include the GA plan modification component and the tests which do make use of the GA plan modification component.

In this chapter the results of the experiments described in Chapter 4 are described and analysed.

5.1 Test configuration

The experiments are performed on three different sized networks, $N \in \{32, 64, 128\}$. For each size 100 tests are run. A number of different configurations of the GA are tested to get a broad view of how well the GA plan modification component behaves. The configurations investigated are covered in Table 5.1.

The random seeds are fixed for the specific numbered test in each envi-

Test Number	GA generations	% individuals added	GA trigger
1	25	1/8	3 failures
2	5	1/8	3 failures
3	100	1/8	3 failures
4	25	1/64	3 failures
5	25	1/8	Every 50 messages
6	25	1/8	Every 20 messages

Table 5.1: Test Configuration

ronment. In other words the first test for the $N = 32$ environment without the GA uses the same random seeds as the first test for the $N = 32$ environment with the GA enabled. The EA uses pseudo-random number streams, controlled by these random seeds to generate the random numbers which are used to modify the environment. Keeping the random seeds the same for each numbered experiment ensures that the environments change identically and allow the effectiveness of the GA plan modification mechanism to be compared as like with like. The exact details of the execution of the EA are detailed in Section 4.2.2.

5.2 Data Collection

During execution the MA records the statistics of each message sent. If the message succeeds then the total duration is recorded along with the exact path that the message took through the network. If the message fails then the partial path, the location that the message delivery failed and the reason for the delivery are recorded. These are stored in an array. When the test run has sent 1000 messages it writes the delivery log out to disk in a delimited file.

Once all 100 runs of a test are completed, the output files are imported into SQL Server for analysis. The analysis in SQL Server involves the averaging of the message durations across the 100 tests in each set as well as averaging the percentage improvement between the baseline experiment without the GA active and the experiment with GA. These averaged values are then exported into Excel for graphing.

The specific route that each message takes through the network is not relevant to the result.

The aggregated data in the form of an Excel spreadsheet as well as a SQL Server database backup containing the raw data are available from `http://gilamonster.za.net/blog/resource-files/`

5.3 Tests with no GA

The first set of experiments was with the GA inactive to form the baseline for future comparison. Hence there was no mechanism present that could modify the plans held by the agent based on the changing environment. Plans that became inefficient because of changing conditions had to be reused. The reactive planning allowed it to pick the best plan that it has, but allows no new plans to be created.

The message durations over the tests for each configuration were averaged and graphed in Figures 5.1, 5.3 and 5.5.

The averaging process for each message in the experiment involved taking the sum of the transmission time for each test across the 100 tests, discarding the values of transmission time for any of the messages that failed delivery and dividing by the number of messages successfully delivered.

The averaging is done because there is a huge amount of raw data for each experiment. Graphing each experiment run individually would make it hard to draw an overall conclusion. While the standard deviation values are high, the averaging is valid because while the individual experiment runs vary in their transmission times, the overall pattern is similar across the tests. In addition, this study is an investigation into feasibility, not a comparison with existing methods and hence the comparisons which need to be made are between experiments within the study, not with existing benchmarks.

As the network moves further from its starting configuration, so the performance of the existing plans degrades and the message delivery times, referred to as t_{msg}, increase. The initial values of t_{msg} are within the range $12s \leq t_{msg} \leq 15.5s$. The final values of t_{msg} are within the range $15s \leq t_{msg} \leq 24s$. The spread of starting and ending values for t_{msg} is due to the random configuration of the network in each test. The latency values, L, vary independently from one run to the next. As such the standard deviation, SD, of the message delivery times is high. For the averaged data shown in Figure 5.1, $0.8 \leq SD \leq 2.5$.

The average value of t_{msg} across the 100 tests will be referred to as t_{avg}.

Initially the increase in t_{avg} is relatively steep, indicated on Figure 5.1 as Phase 1. After this, in Phase 2, the rate of change decreases. The values of that increase in seconds per message are given in Table 5.2.

	Rate of change in seconds per message
Phase 1	0.005
Phase 2	0.0015

Table 5.2: Rate of change, $N = 32$, no GA

To see the specifics of an individual test run, without the averaging, the values for five individual runs are graphed in Figure 5.2. The gaps in the lines are where messages failed delivery. If a delivery failed t_{msg} cannot be

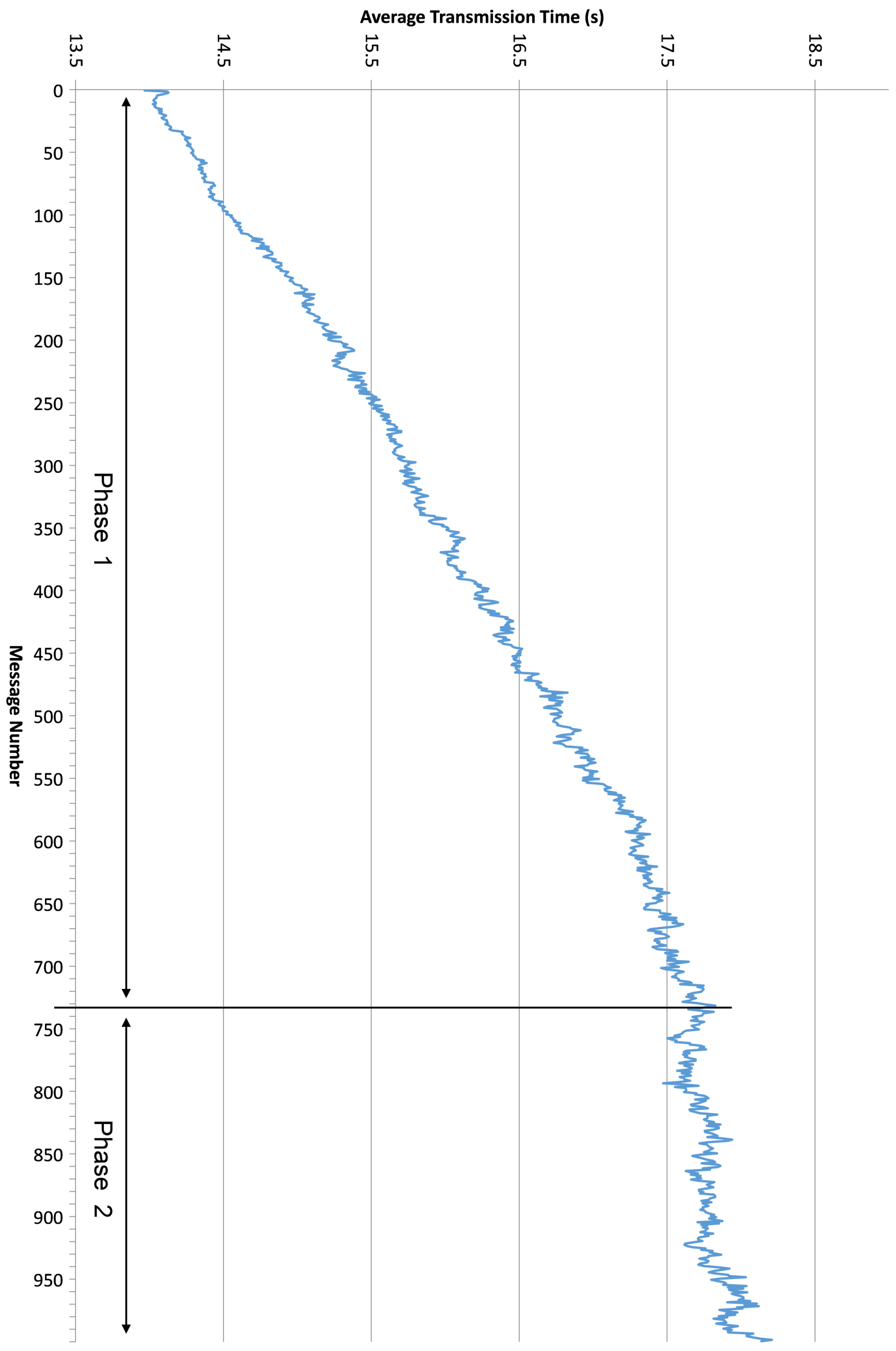

Figure 5.1: Average durations, $N = 32$ network

recorded as the message was not delivered. It is left blank, leading to gaps in the graph lines.

t_{msg} for the individual messages in each of the five selected tests start within a small range, $13.3s \leq t_{msg} \leq 14.6s$. The values diverged as the networks were modified in different ways by the EA.

The progression with the $N = 64$ network, shown in Figure 5.3, is much the same as for the $N = 32$ network shown in Figure 5.1. It shows a steady increase in the t_{avg} as the plans that the agent has become less and less appropriate for the changing configuration of the network.

The rate of change of t_{avg} is detailed in Table 5.3. The phases mentioned in the table are indicated on Figure 5.3.

	Rate of change in seconds per message
Phase A	0.008
Phase B	0.0025

Table 5.3: Rate of change, $N = 64$, no GA

At the start of the experiment, $t_{avg} = 29s$ and at the end $t_{avg} = 35.6s$.

As with the $N = 32$ network, the standard deviation, SD, is high due to the independence of the networks in the 100 tests, falling withing the range $957 \leq SD \leq 2877$. The transmission times for five individual tests are shown in Figure 5.4.

In this case, while the increase slowed towards the end of the test, it did not slow as much as it did on the smaller network. This would make sense if the levelling out is a result of the network reaching a steady state. The $N = 64$ grid has a much larger range of possible transmission times than the $N = 32$ network has and a larger set of starting plans. This is due to the larger number of nodes and links that the $N = 64$ network has.

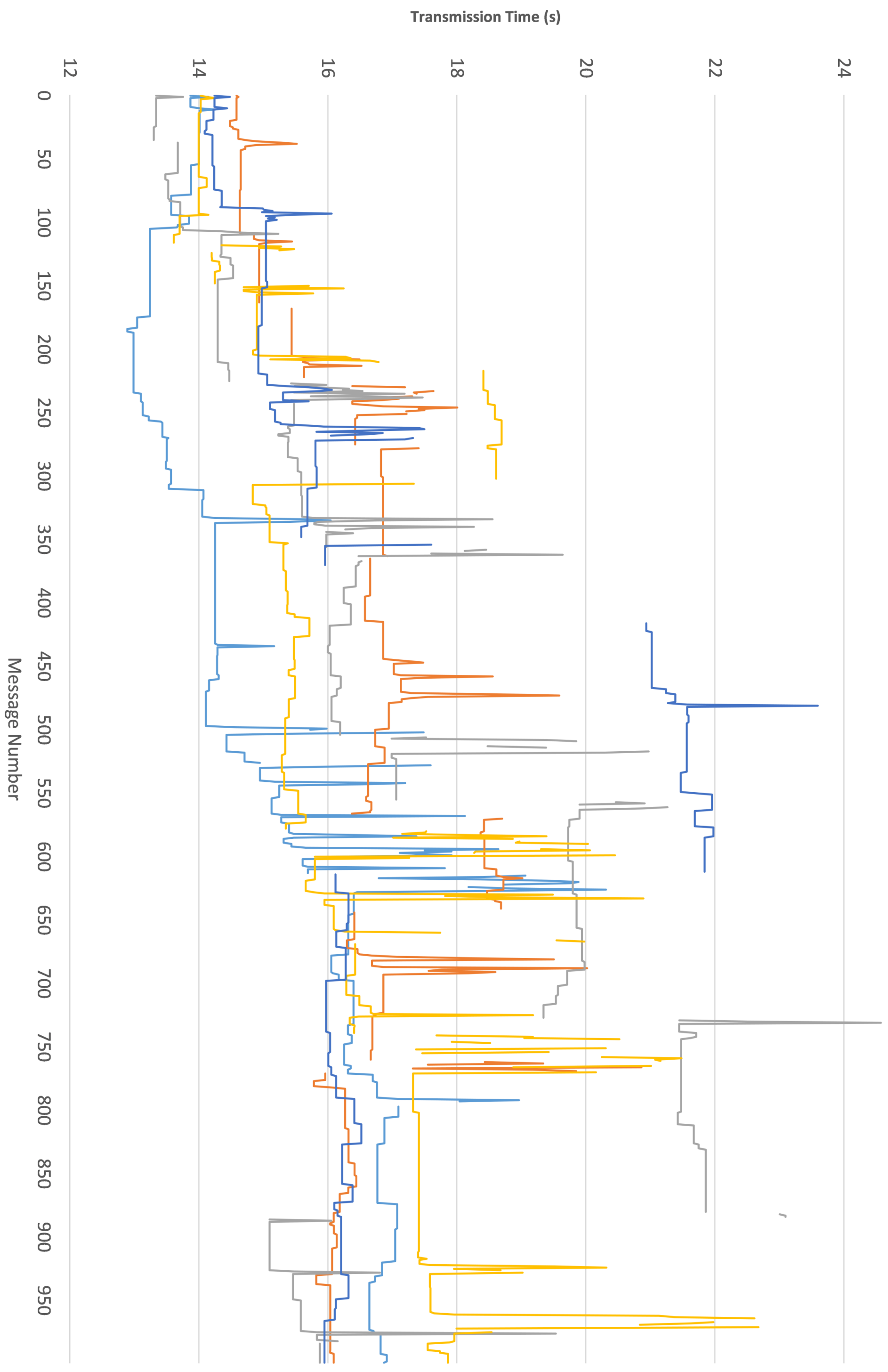

Figure 5.2: Individual durations for five tests, $N = 32$ network

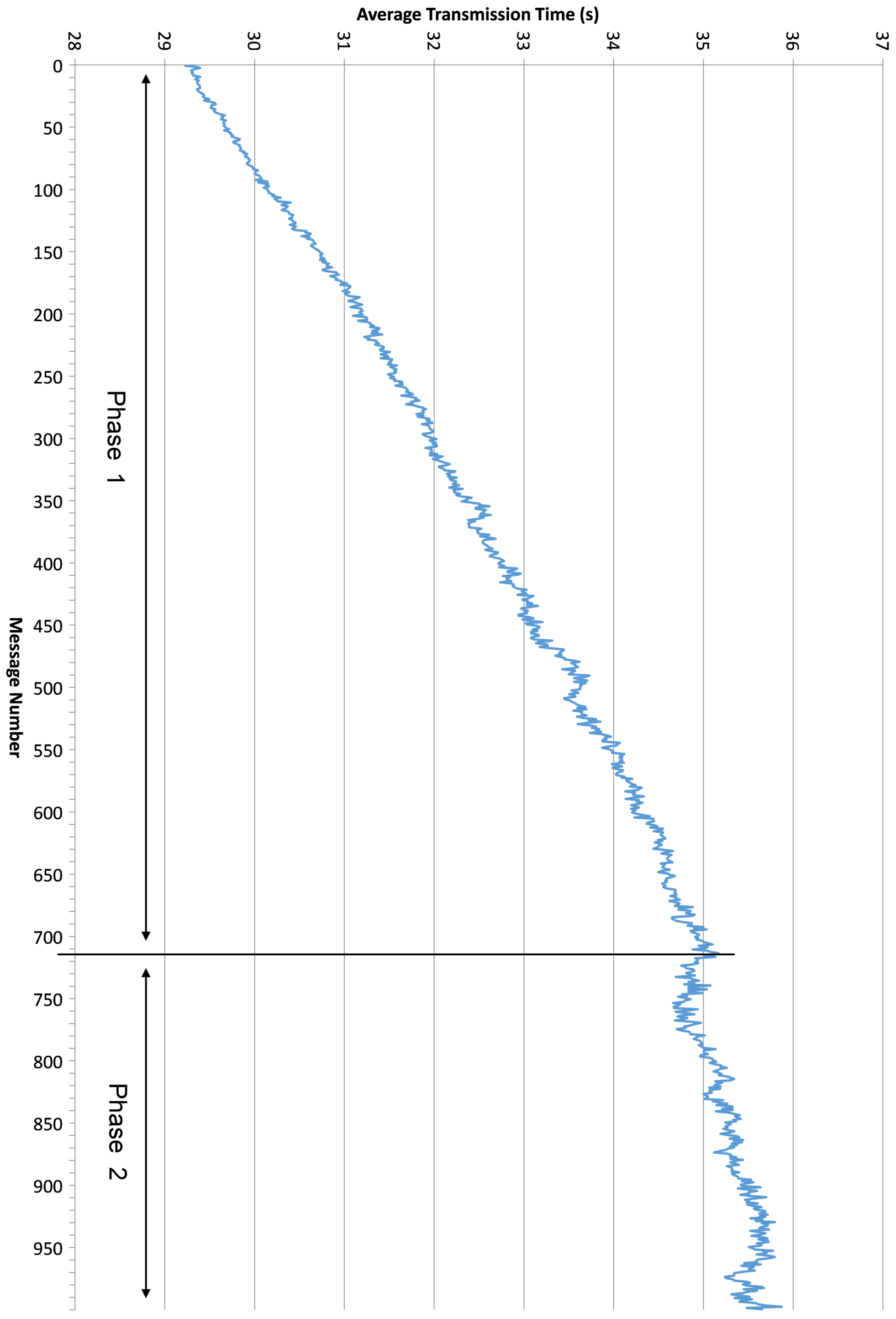

Figure 5.3: Average durations, $N = 64$ network

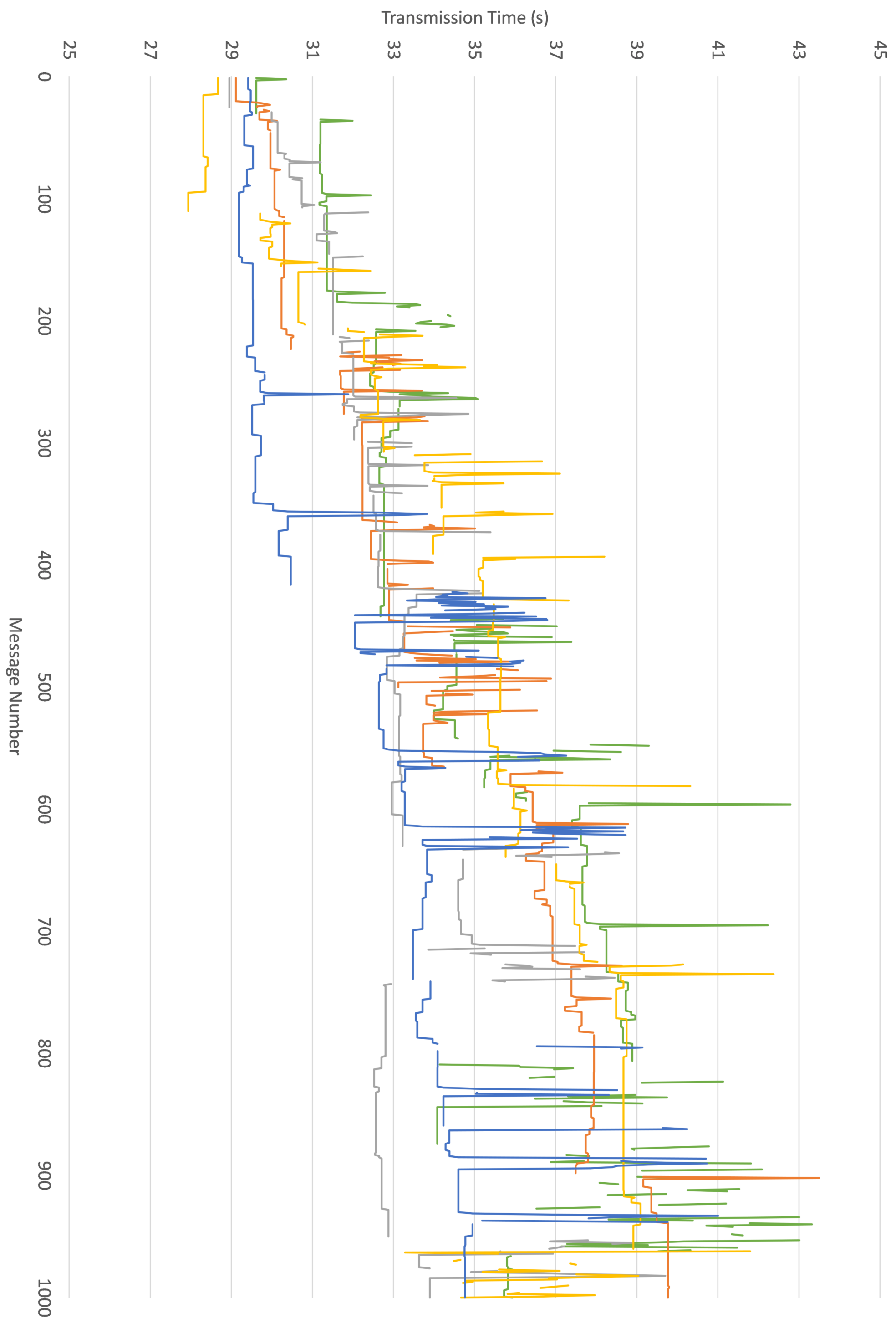

Figure 5.4: Individual durations for five tests, $N = 64$ network

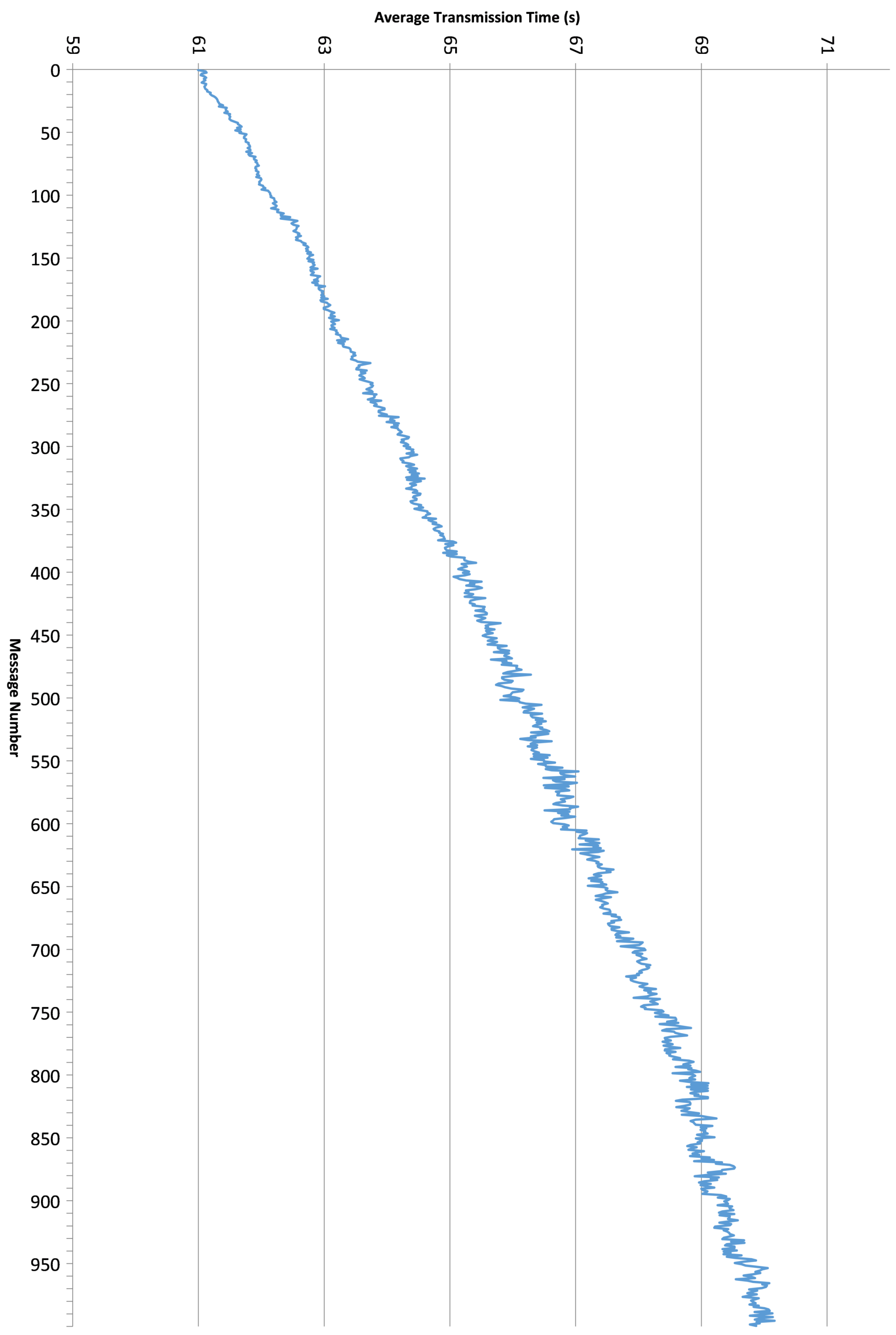

Figure 5.5: Average durations, $N = 128$ network

The results for the largest network, $N = 128$, are shown in Figure 5.5. They show the largest increase in the average network transmission time, with an almost linear increase across the course of the experiment. This increase is at a rate of 0.009 seconds per message.

5.4 First set of tests with GAs

The first experiment using the GA to evolve the plans used a single message failure as a trigger. The GA used the entire contents of the plan library as a starting population. The evolution was set to proceed for five generations and then to add the fittest $^1/_8$ of the final generation to the plan library. The plan library started with K_{init} plans, where $K_{init} = 4N$. Since each execution of the GA added the $^1/_8$ of the final generation with the highest fitness, K increased by $^1/_8$ each time the GA executed.

The growth of the plan library is shown in Figures 5.6 and 5.7.

The addition of multiple plans to the library is intended to increase the chance that a new plan replaces one that is currently failing and reduces future failures by giving the MA more options to choose from. The disadvantage of doing so is that the plan library grows faster than if a single plan is added. This requires additional resources in the form of memory to store the plan library and processing time to locate a plan within the library.

Given that the GA is operating in a changing environment, the low fitness individuals should not be permanently discarded, as was discussed in Section 4.3.4. Because the environment is constantly changing, the fitness function is time dependent. As such, an individual with a poor fitness at time t_1 may have a much better fitness at a later time t_2. If that individual is discarded when it has a low fitness, it may hinder the evolution process at a later time when that discarded genome may have a higher fitness value.

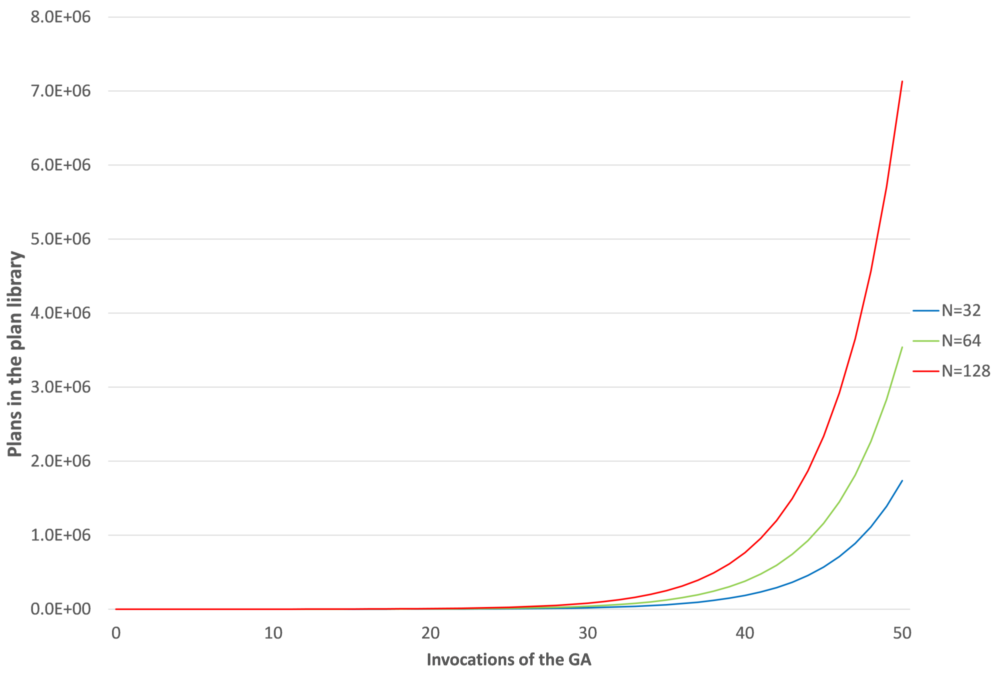

Figure 5.6: Growth of the plan library (linear)

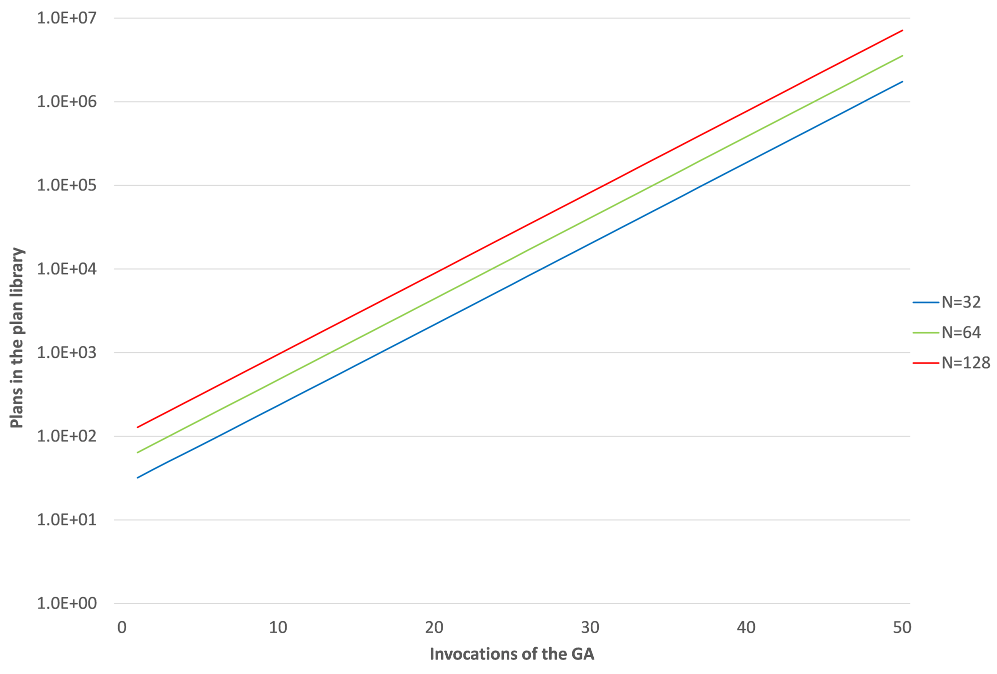

Figure 5.7: Growth of the plan library (logarithmic)

The method chosen in this dissertation is to use the default configuration for `JGAP` of keeping a fixed population during each execution of the GA. The initial population for the GA is not randomly created but is created from the plan library. The entire plan library, as it is at the time the GA execution starts, is used as the starting population. The plan library increases in size by $^1/_4$ on every execution of the GA. Hence the starting population size increases with every invocation of the GA. As was discussed in Section 3.4 the execution time of a GA is bounded below by the sum over all generations of the product of the population size (P) and execution time of the fitness function (t_f), as per Equation 5.1.

$$t_{GA} \geq \sum_{i=1}^{G}(Pt_f) \tag{5.1}$$

The increase in population produces a sharp increase in the processing time of the GA each time it is invoked. The growth of the population is given by Equation 5.2, where $P(1)$ is the starting size of the population.

$$P(I) = \frac{5}{4}P(I-1) \tag{5.2}$$

The increase in GA execution time and increase in the size of the plan library are shown in Figures 5.8 and 5.9.

The increase in execution time was exponential. This indicated that the initial design choice was infeasible. The increase in time was such that the majority of the time spent was being spent running the GA to generate new plans rather than using the generated plans.

The increase in GA time is shown in Table 5.4.

A second problem which occurred with the initial design is the time required to search the plan library for the best plan. The search method used is a linear search, an $O(n)$ algorithm. The faster the library size increased, the longer the time required to search through all increases. This

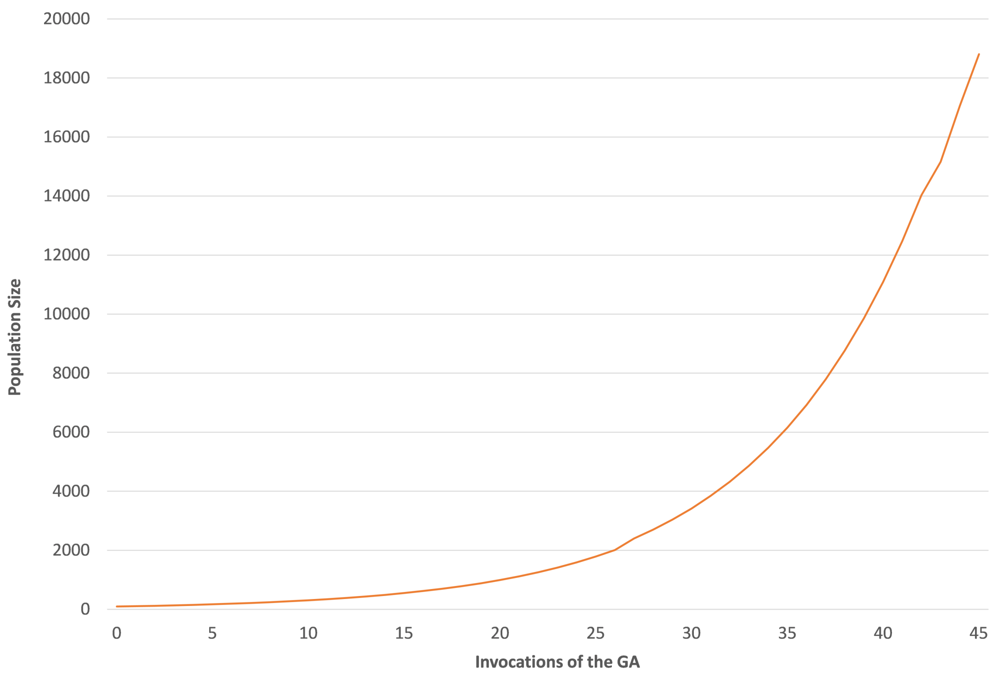

Figure 5.8: Population increase

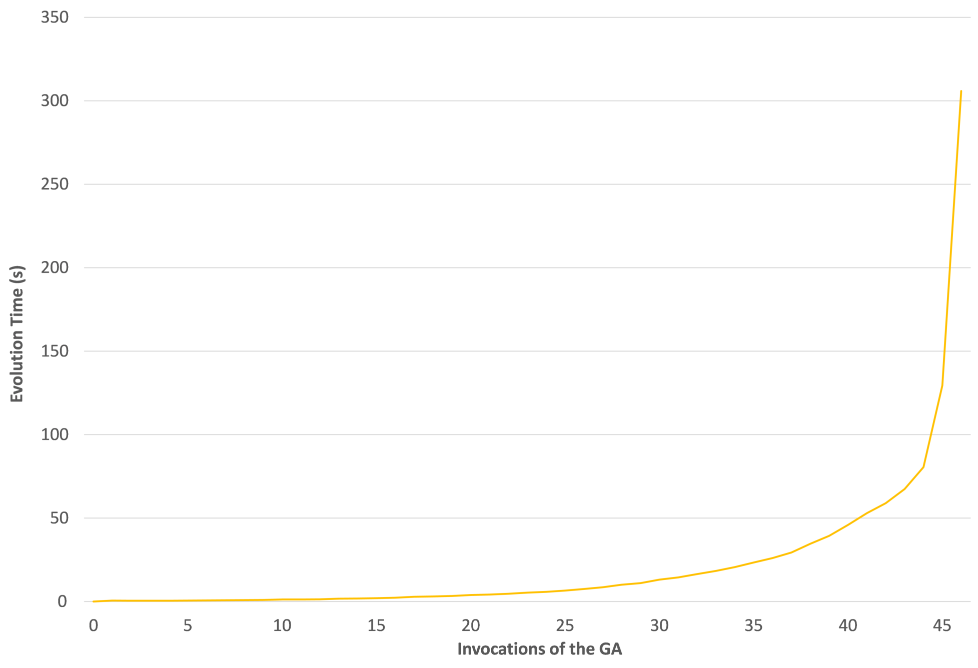

Figure 5.9: Evolution time increase

Invocations	Population size	Evolution time (ms)	Invocations	Population size	Evolution time (ms)
0	0	0	24	1413	5271
1	96	579	25	1589	6109
2	108	428	26	1787	6815
3	121	443	27	2010	7843
4	136	476	28	2261	8464
5	153	535	29	2543	9540
6	172	602	30	2860	10776
7	193	691	31	3217	12622
8	217	766	32	3619	13818
9	244	877	33	4071	15494
10	274	1106	34	4579	17694
11	308	1086	35	5151	19727
12	346	1215	36	5794	22075
13	389	1400	37	6518	24802
14	437	1608	38	7332	27899
15	491	1886	39	8248	32393
16	552	2157	40	9279	38500
17	621	2434	41	10438	42508
18	698	2652	42	11742	47414
19	785	2950	43	13209	51510
20	883	3493	44	14860	64878
21	993	3797	45	15460	78430
22	1117	4404	46	17392	133307
23	1256	4881			

Table 5.4: Increase in GA execution time due to increasing population size

would favour keeping the plan library as small as possible. An alternative would be to use a more efficient search algorithm, potentially one of the $O(log(n))$ search algorithms. Doing so would mitigate the problem of searching through the plan library, however would not help with the increase in the execution time of the GA.

The time required to find a plan was not included in the measurements of t_{msg} and so the increasing library did not affect the results described later in this chapter. In a real system using the technique outlined in this dissertation, steps to keep the plan library small would be required. These might include removing duplicate or near-duplicate plans, removing plans that failed on their previous execution or dividing the plan library into two sections, one containing plans that are currently effective and the other section containing the remaining plans. The last option would require an additional process to periodically test the effectiveness of plans. Depending on the environment, that may not be possible.

The second version of the GA design involved triggering the evolution on every n^{th} failure instead of on every failure, with $n = 5$. Since the evolution was triggered much less often, the size of the plan library increased slower and the evolution time increased proportionally slower.

This allowed the required number of messages to be sent without the evolution time becoming overwhelming as shown in Figure 5.11. However the problem of exponential growth of the starting population still existed. Had the experiment been scaled past the 1000 message mark, there would have been a point where the time required for the GA evolution cycle became unacceptably large when compared to t_{msg}.

Considering those results, it was necessary to rethink how the initial population was created each time the GA was invoked.

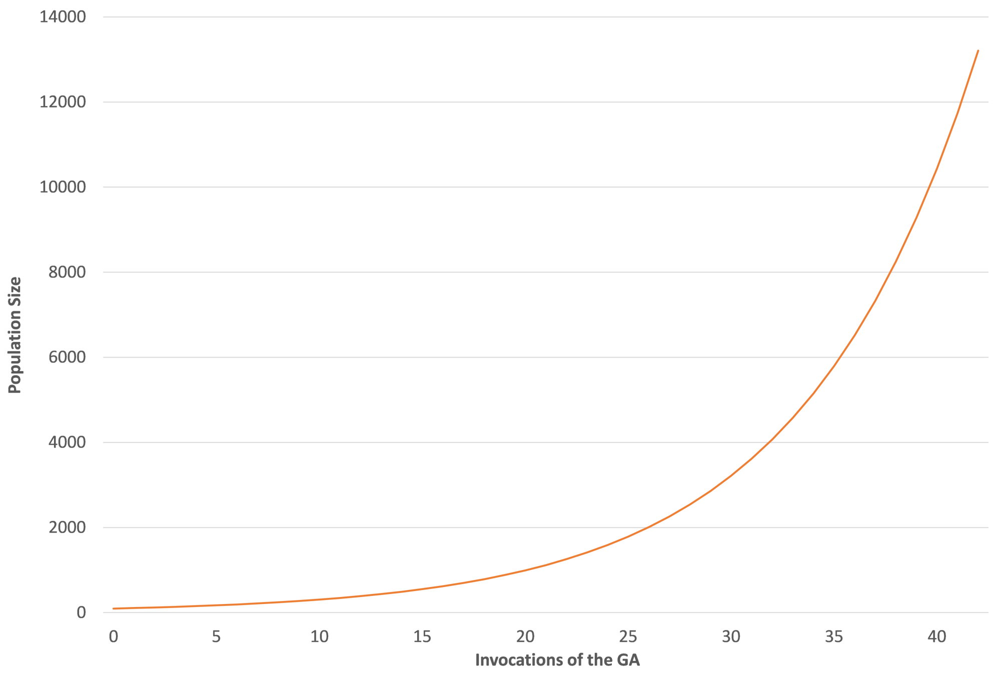

Figure 5.10: Population increase

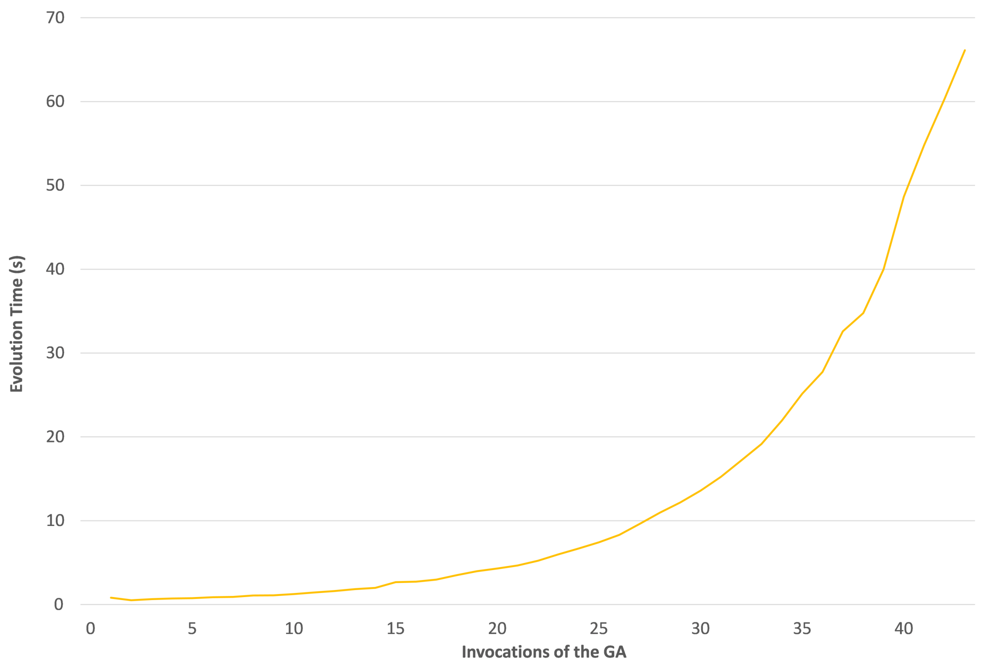

Figure 5.11: Evolution time increase

5.5 Revised design

The root cause of the rapid increase in evolution time is the exponential increase in the size of the starting population over the course of the experiment. A way of avoiding that increase in evolution time would be to fix the size of the starting population. If the starting population size is fixed, then the time required for the GA to execute a fixed number of generations is a linear function.

The concern with this approach is the potential loss of genetic diversity across the invocations of the GA, as discussed earlier in this chapter. If the starting population for any invocation of the GA is to be kept at a fixed number with the total number of plans in the plan library increasing, the plans selected to form the initial population must be selected at random from the set of all plans available to the MA in order to allow for a large genetic diversity.

It is possible to use a high mutation rate instead of a random selection of the population as the higher mutation rate would add in the necessary genetic diversity, as was mentioned in Section 3.3.2. This approach is not used here.

The revised design for the GA used the following configuration settings:

- A fixed number of individuals is used as the starting population for each invocation of the GA. This number equals the initial size of the plan cache for that size network, outlined earlier in this chapter.

- The GA is invoked on every 3^{rd} message failure. This was chosen as a balance between invoking the GA as soon as there is evidence that the current plans are no longer optimal and waiting for multiple failures and allowing messages to execute with plans known to be sub-optimal.

- At the end of each invocation of the GA, the top $^1/_8$ fittest individuals are added to the plan library. This would be $^1/_8$ of the number of individuals in the starting population. For an $N = 32$ network, this works out to 16 plans, for an $N = 64$ network, 32 plans and for the $N = 128$ network, 64 plans.

The plan library increases in size linearly, as each invocation of the GA adds the same number of plans to the library. The final size (K_f) of the plan library can be expressed as a function of the initial size (K_{init}) and the number of times which the GA is invoked (i) as shown in Equation 5.3.

$$K_f = K_{init} + (1/8 \cdot K_{init}) \cdot i \tag{5.3}$$

The starting population size for the GA is fixed and hence the execution time of the GA over the total number of generations remains constant across the duration of the experiment.

5.6 Second set of tests with GAs

The results over the hundred tests for each configuration are averaged and compared against the statistics for the tests without the GA, shown in figures 5.12, 5.13 and 5.14.

All three tests show a sharp initial reduction in t_{avg}. This is due to the GA finding more efficient plans than the initial set which the experiment began with. The extent and gradient of the initial reduction is listed in Table 5.5. Phases 1 and 2 are indicated on Figures 5.12, 5.13 and 5.14. The cut-off point betwee the phases in these and subequent graphs were identified by examining linear trend lines and seeing at what point the trend lines cross.

After that initial steep improvement, the change in t_{avg} flattens out with only moderate improvements over the course of the experiment. Dur-

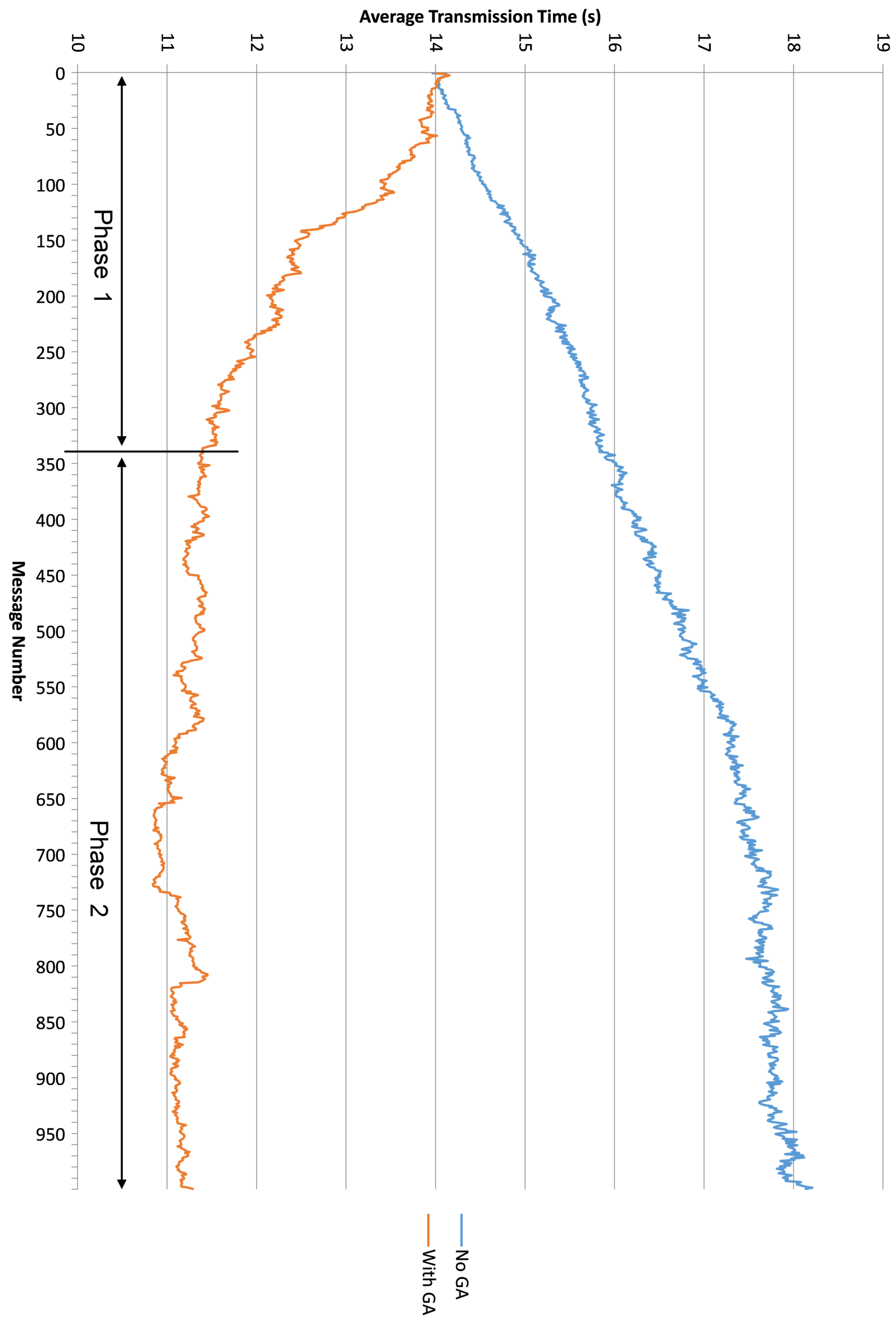

Figure 5.12: Average durations, $N = 32$ network, GA enabled

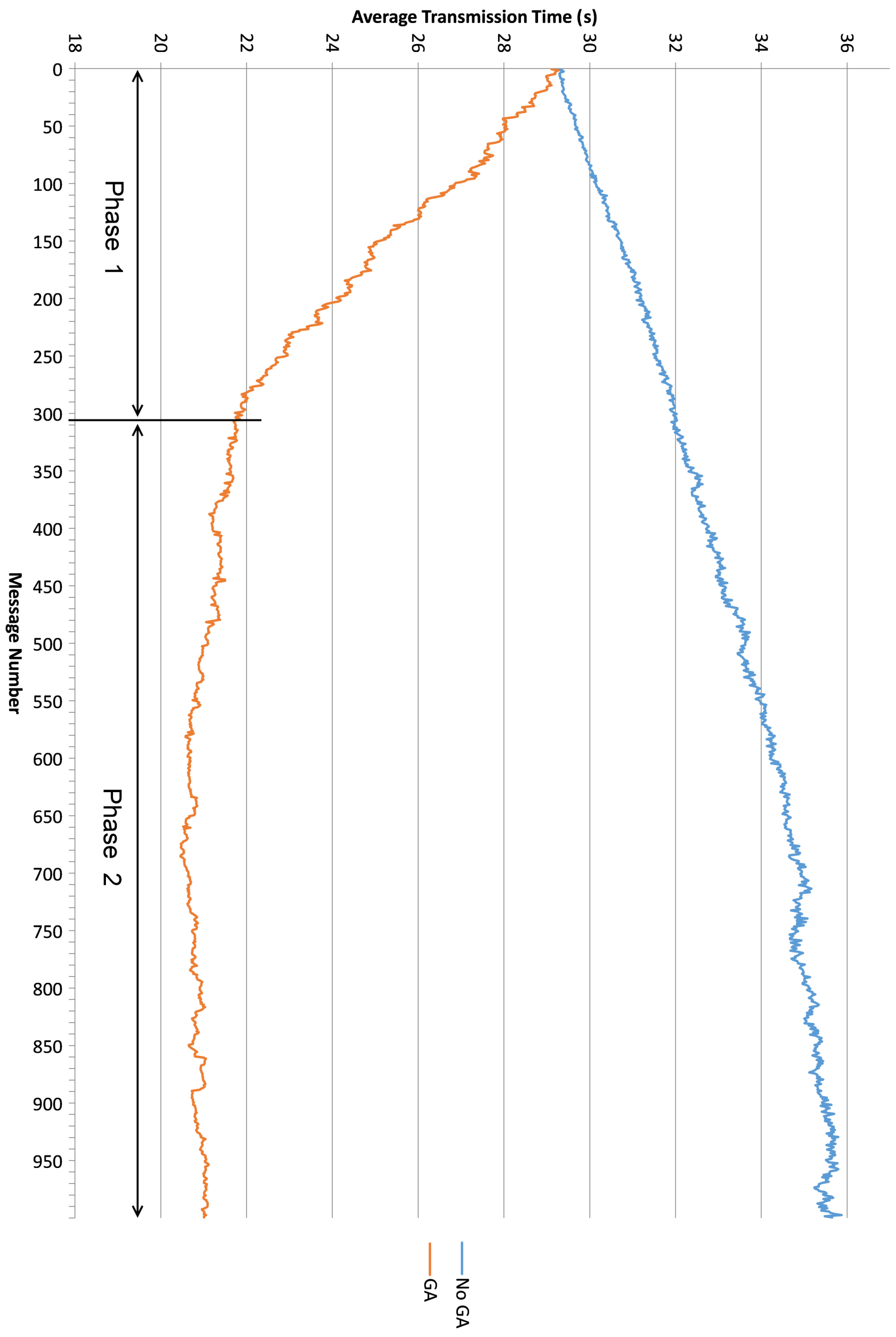

Figure 5.13: Average durations, $N = 64$ network, GA enabled

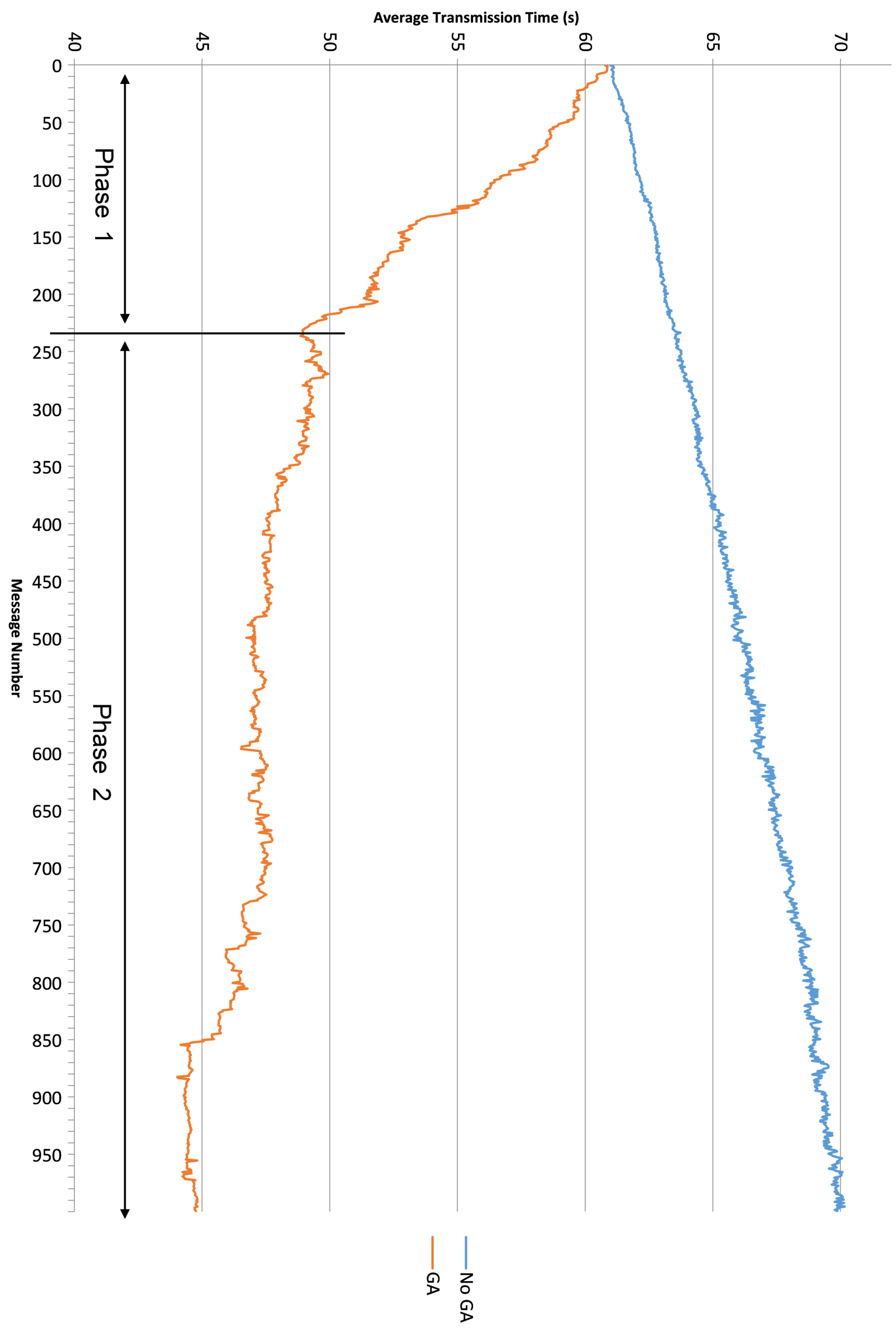

Figure 5.14: Average durations, $N = 128$ network, GA enabled

Network size	End of Phase 1	Phase 1 - rate of change	Phase 2 - rate of change
32	338	-0.0065	0
64	291	-0.025	-0.001
128	230	-0.05	-0.006

Table 5.5: Rates of change across all network sizes

ing this period the environment is still changing further from the initial configuration.

An alternate way to look at the results of these tests is to evaluate the percentage improvement that the test with the GA plan modification component showed over the default test without the GA. The percentage improvement for the three network sizes are graphed in Figures 5.15, 5.16 and 5.17.

The alternate method of analysis shows the steep initial improvement in t_{avg} followed by a slower increase that flattens out on the $N = 32$ and $N = 64$ networks. On the $N = 128$ network the increase does not flatten out.

Given that the t_{avg} of the tests without the GA present worsens over the entire course of the experiment while the t_{avg} with the GA present shows a sharp initial improvement with a slower later improvement, this suggests that the GA achieves the goal of optimising the plans to cope with a slowly changing environment. The addition of the GA plan modification component was the only change between the two tests.

5.7 Tests with different generation limits

The concern with the overhead of the GA remains valid. While the GA evolution executes, the MA has to either use outdated plans or has to stop

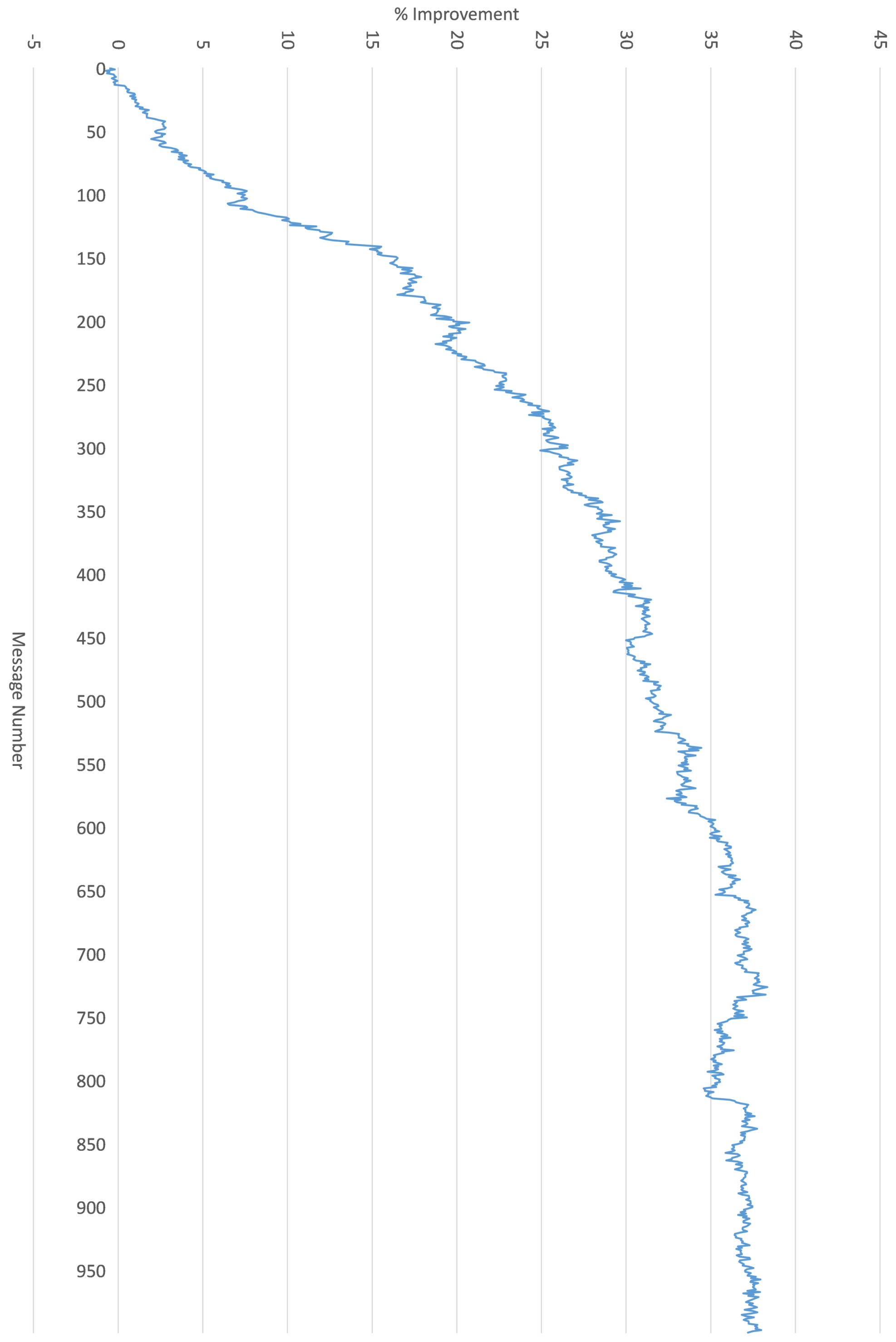

Figure 5.15: Percentage improvement over test without GA, $N = 32$ network, GA enabled

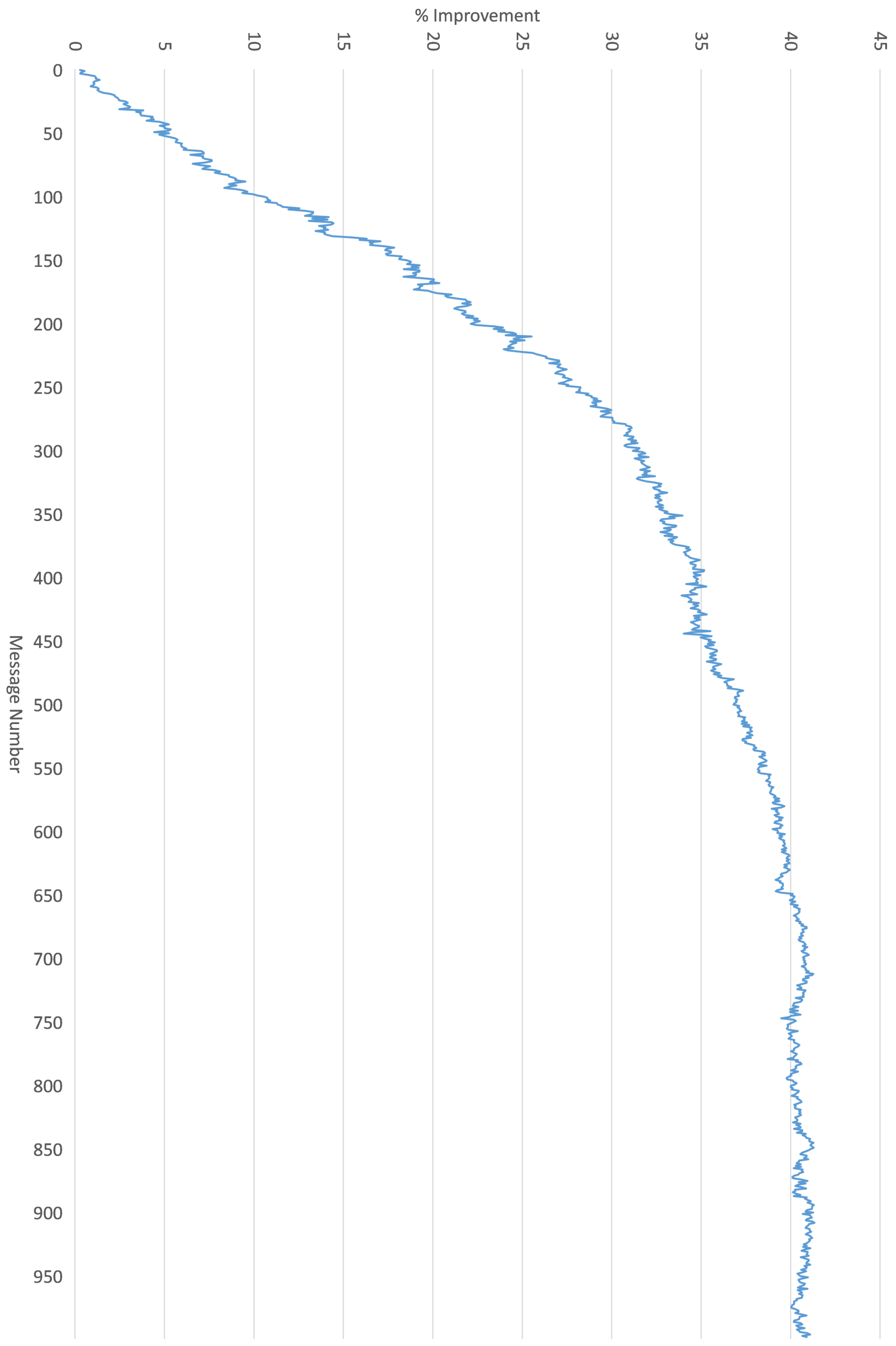

Figure 5.16: Percentage improvement over test without GA, $N = 64$ network, GA enabled

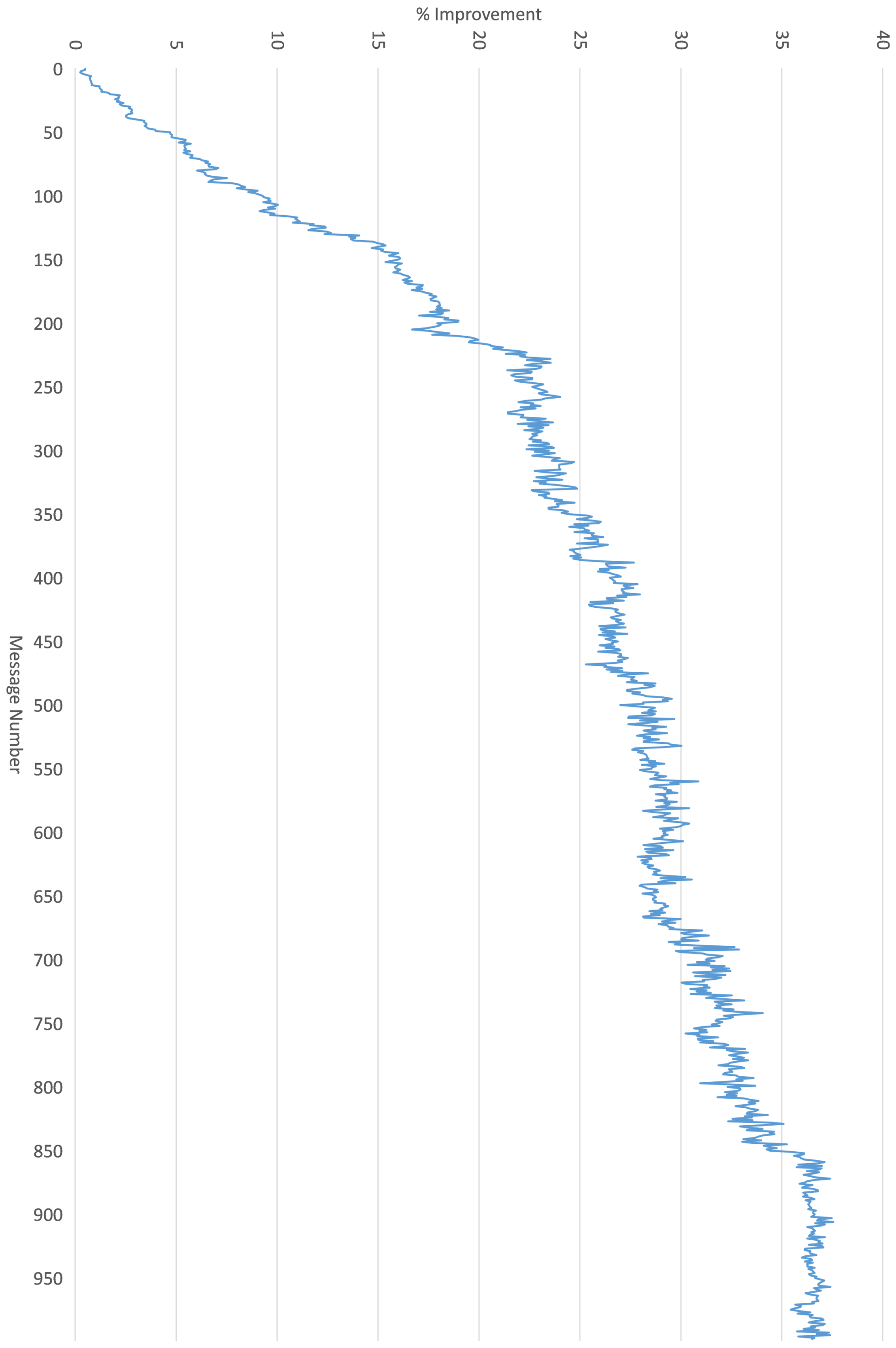

Figure 5.17: Percentage improvement over test without GA, $N = 128$ network, GA enabled

executing and wait for new plans. The longer the execution cycle of the GA takes, the more messages must be sent with plans that are known to be sub-optimal. Alternately the MA must pause and wait for the GA execution cycle to complete, thus reducing the overall rate at which it can send messages.

Given that, reducing the time spent by the GA would theoretically decrease its impact. If one starts from the assumption that the fitness function is already as efficient as possible, then the two avenues for reducing the execution time of the GA are to reduce the starting population size or to reduce the number of generations that the GA executes, as discussed in Section 3.4.

Reducing the population size would reduce the genetic diversity that the GA starts with. Considering that the initial population is a random selection of the current plans, reducing the starting population may reduce the ability of the GA to find plans suitable for the state of the environment every time it is invoked due to potentially low genetic diversity in its starting population. This option has not been investigated in this dissertation. It may be suitable for future work.

Reducing the number of generations reduces the degree to which the GA can find optimal solutions. In this particular scenario this approach may be acceptable, since all that is needed is for the GA to find new plans which perform better than the current ones, that is, plans which result in a lower t_{msg}.

Further experiments were done to investigate the effect of different numbers of generations, G, which the GA evolves for. In the first set of experiments $G = 25$. The further experiments consist of one set with $G = 5$ and one set with $G = 100$. The results of these are shown in Figures 5.18, 5.19 and 5.20.

The test with $G = 5$ showed a slower and prolonged initial improvement

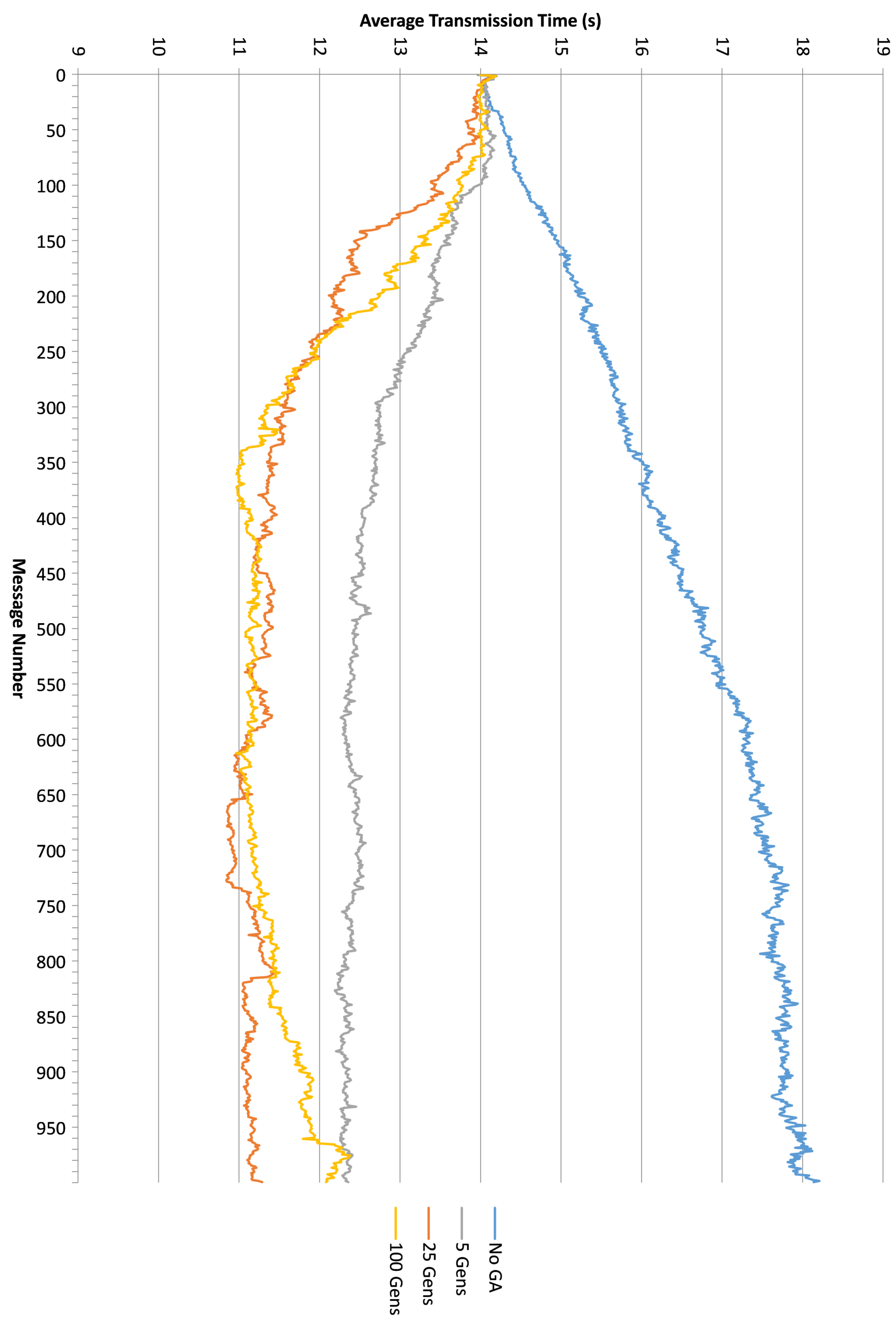

Figure 5.18: Average durations, $N = 32$ network, GA enabled with different number of generations

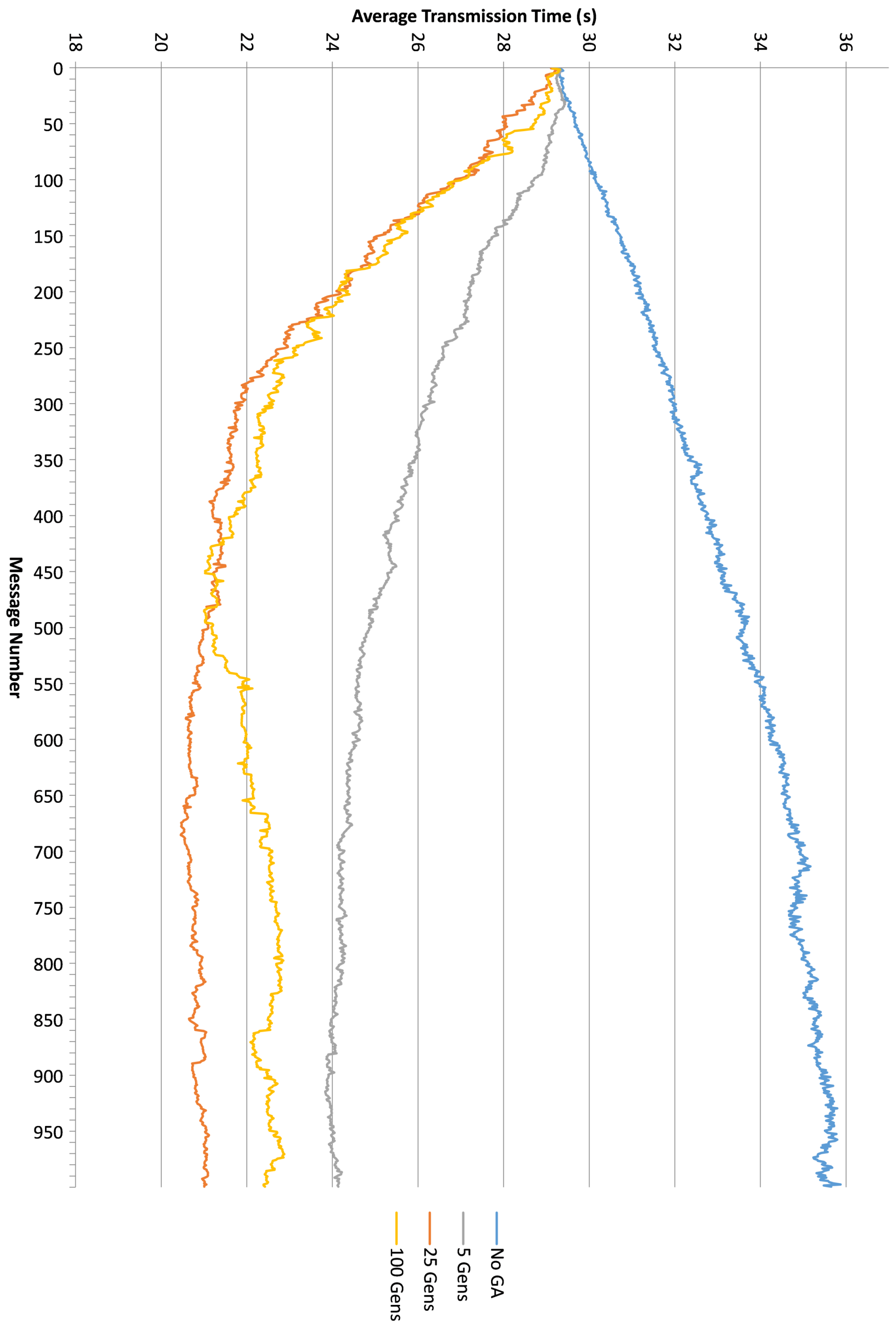

Figure 5.19: Average durations, $N = 64$ network, GA enabled with different number of generations

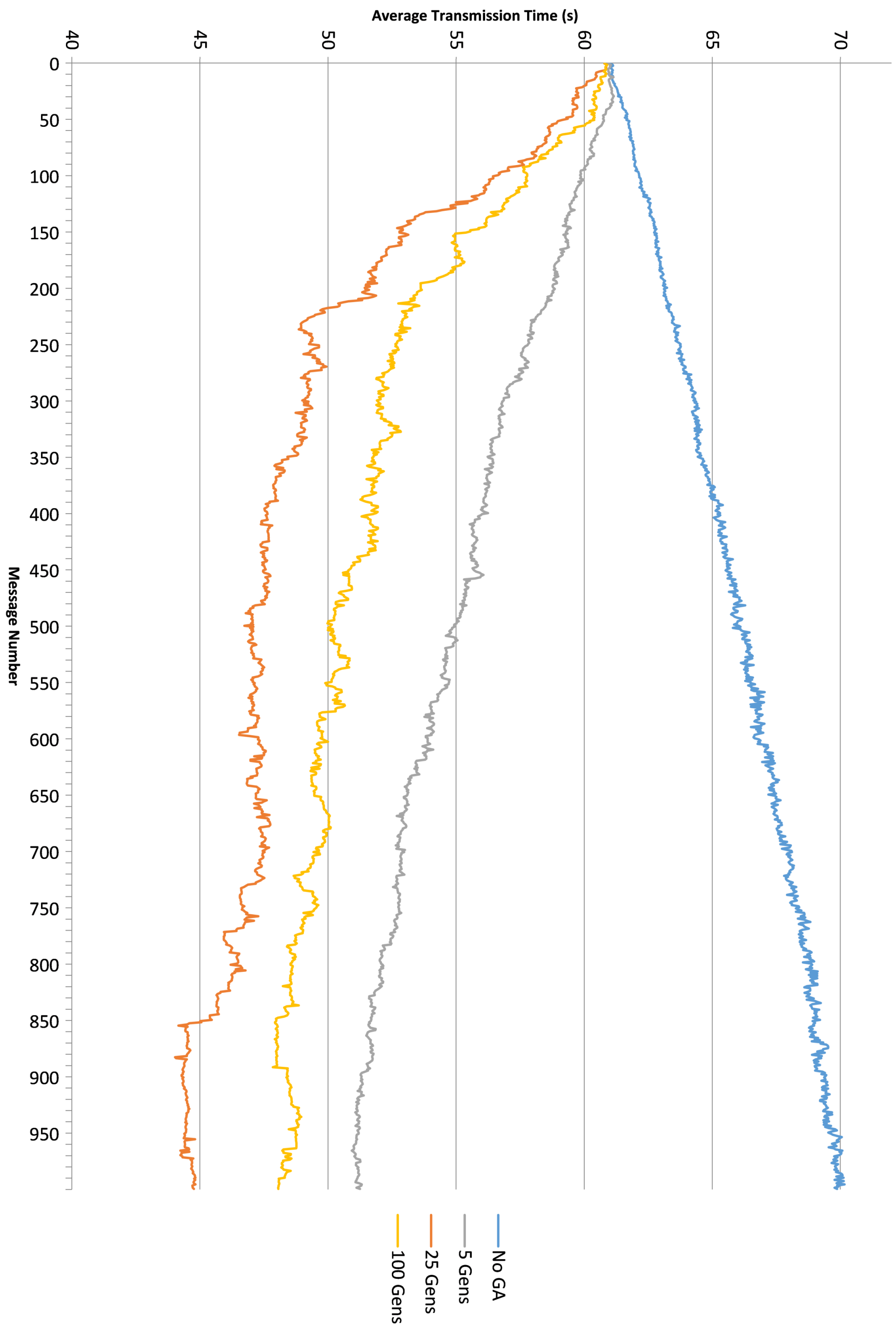

Figure 5.20: Average durations, $N = 128$ network, GA enabled with different number of generations

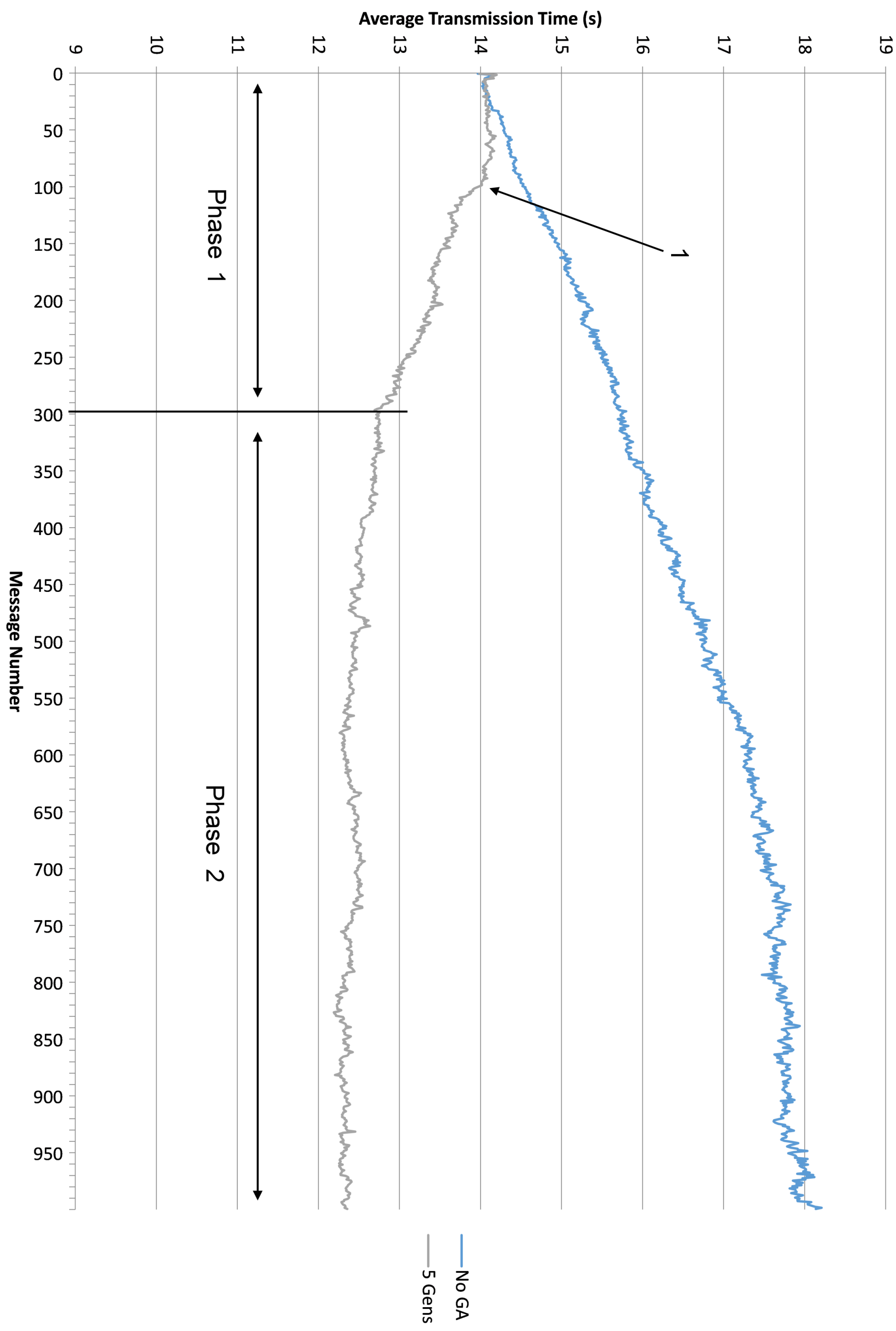

Figure 5.21: Average durations, $N = 32$ network, GA enabled with $G = 5$

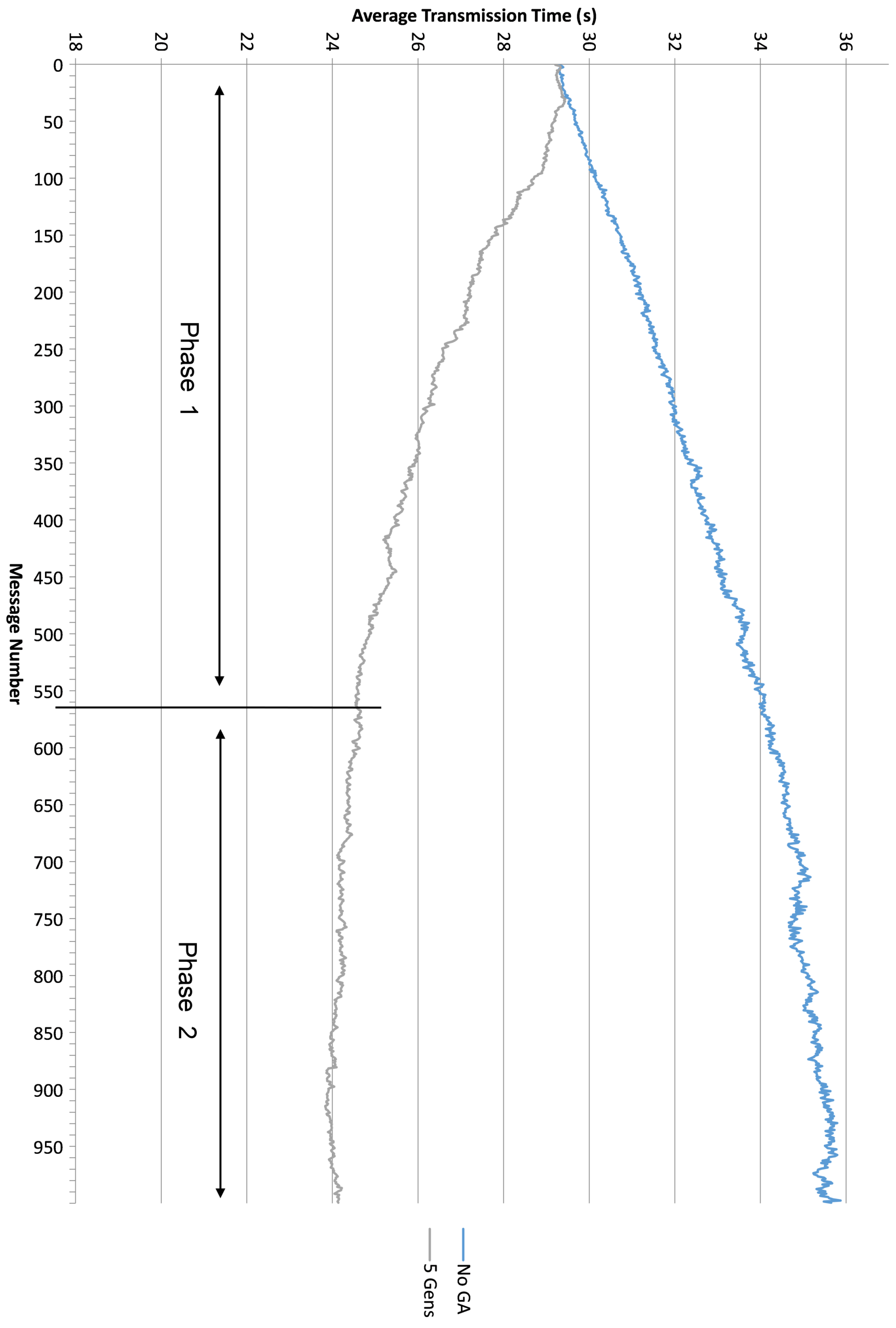

Figure 5.22: Average durations, $N = 64$ network, GA enabled with $G = 5$

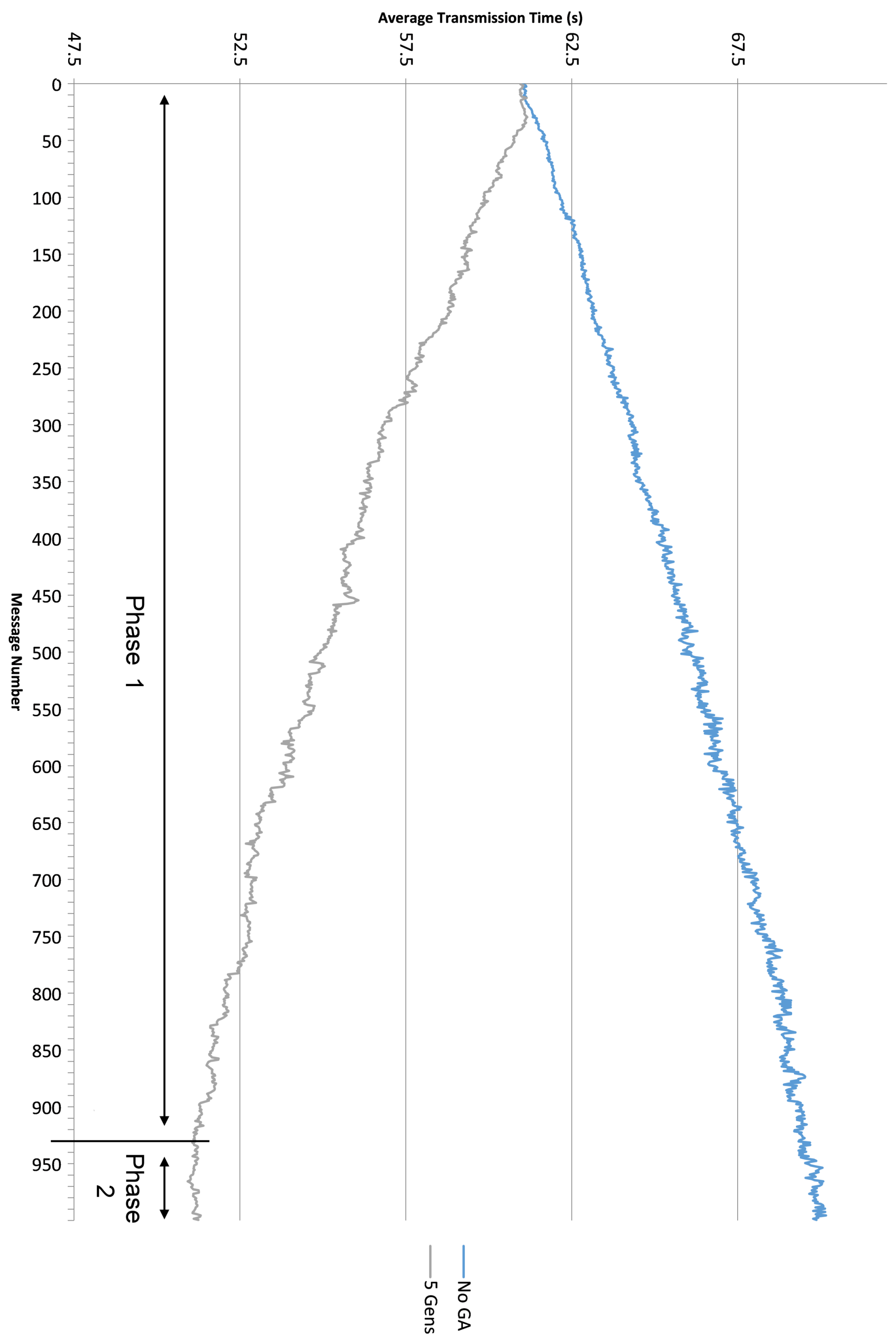

Figure 5.23: Average durations, $N = 64$ network, GA enabled with $G = 5$

Network size	Message No around which Phase 1 ends	Phase 1 - rate of change in sec/message	Phase 2 - rate of change in sec/message
32	300	-0.004	-0.0005
64	550	-0.008	-0.001
128	930	-0.01	-0.0018

Table 5.6: Rates of change across all network sizes with $G = 5$

phase, with the largest network showing a flattening only after message 900. The rate and extent of the initial reduction phase is listed in Table 5.6 and the division between the two phases is indicated on Figures 5.21, 5.22 and 5.23.

On the smallest network, the GA appears to have a delayed effect, with the average transmission time starting to drop only after message 100, indicated by point marked “1” on Figure 5.21. This is an artefact of the averaging process. On the smallest network, most of the individual tests do not have three message failures within the first 100 messages and the GA is only invoked after three message delivery failures. As such, t_{avg} only shows a steady decrease later on the graph than for the larger networks. The larger networks where the triggering three failures occur earlier, the drop in the average performance occurs earlier.

The test with $G = 100$ did not show any improvement over the 25-generation test. Instead it showed a similar performance during the early stages of the tests on the $N = 32$ and $N = 64$ networks and with a worsening of performance later. On the $N = 128$ network the $G = 100$ test showed worse performance than the $G = 25$ test.

The details are shown in Tables 5.7 and 5.8 and in Figures 5.24, 5.25 5.26, 5.27, 5.28 and 5.29.

After the initial improvement phase the test with $G = 5$ showed the

Network size	Message No around which Phase 1 ends	Phase 1 - rate of change in sec/message	Phase 2 - rate of change in sec/message
32	340	-0.0065	0
64	260	-0025	-0.001
128	230	-0.05	-0.006

Table 5.7: Rates of change across all network sizes with $G = 25$

Network size	Message No around which Phase 1 ends	Phase 1 - rate of change in sec/message	Phase 2 - rate of change in sec/message
32	340	-0.008	0.0016
64	430	-0.018	0.002
128	197	-0.036	-0.006

Table 5.8: Rates of change across all network sizes with $G = 100$

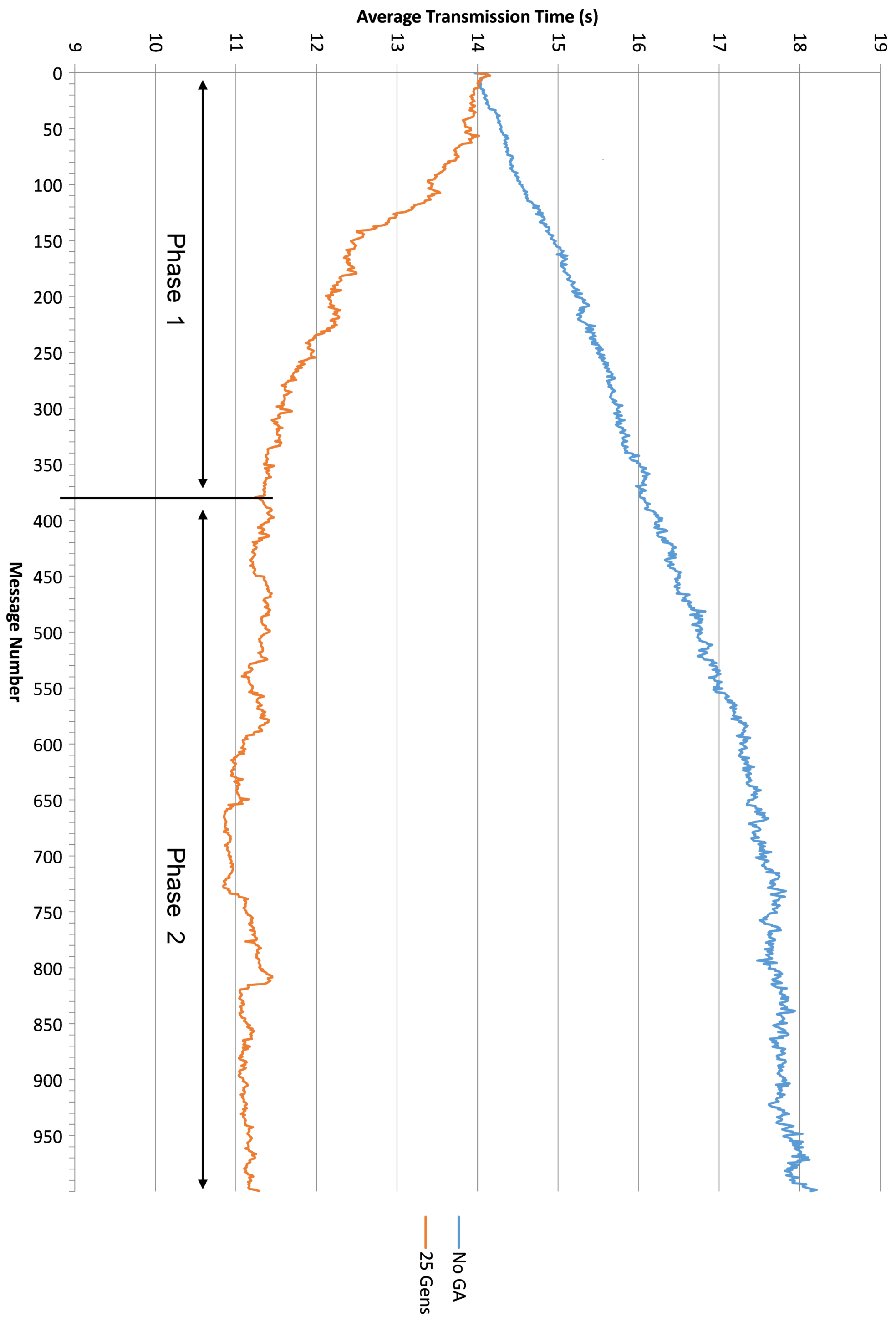

Figure 5.24: Average durations, $N = 32$ network, GA enabled with $G = 25$

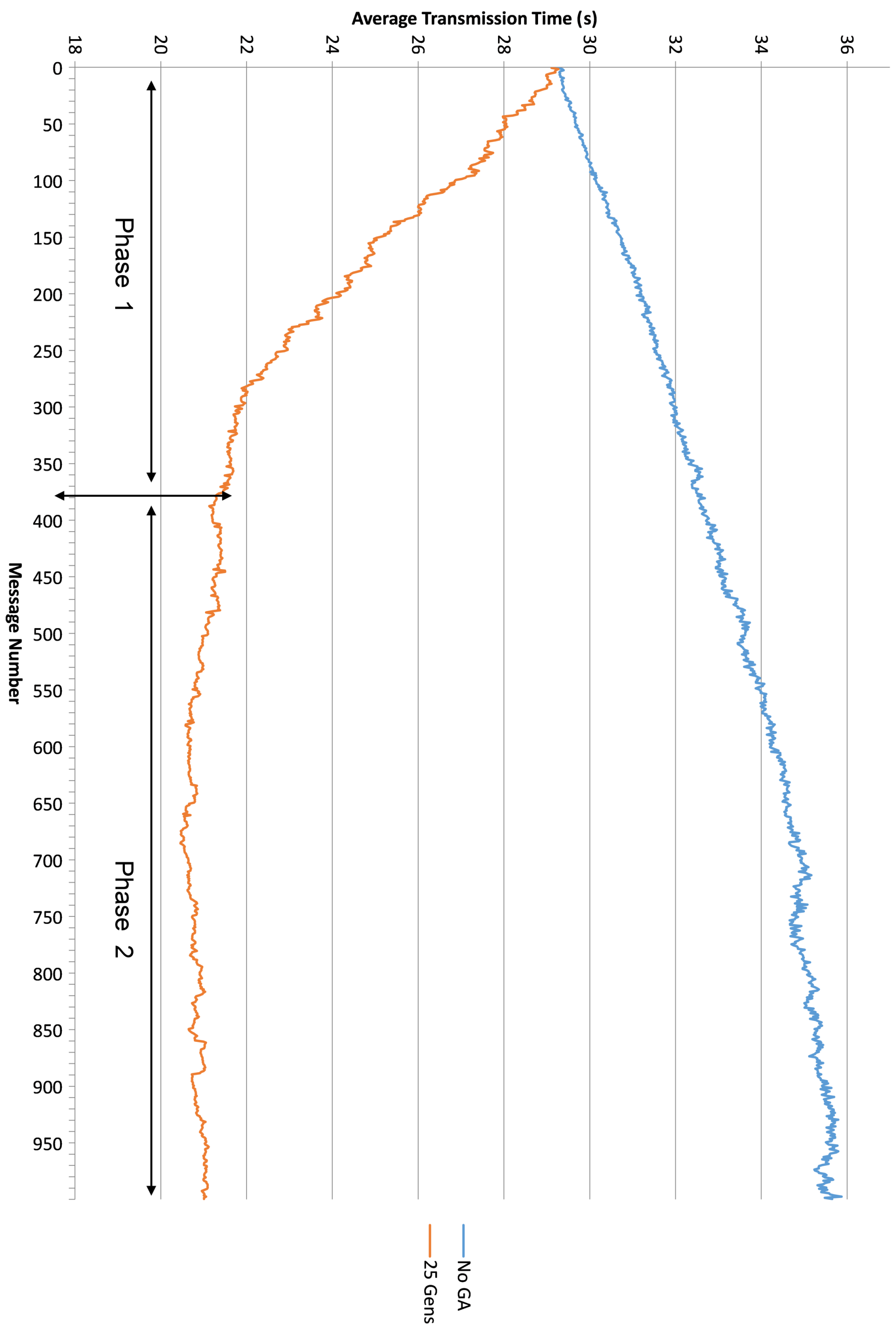

Figure 5.25: Average durations, $N = 64$ network, GA enabled with $G = 25$

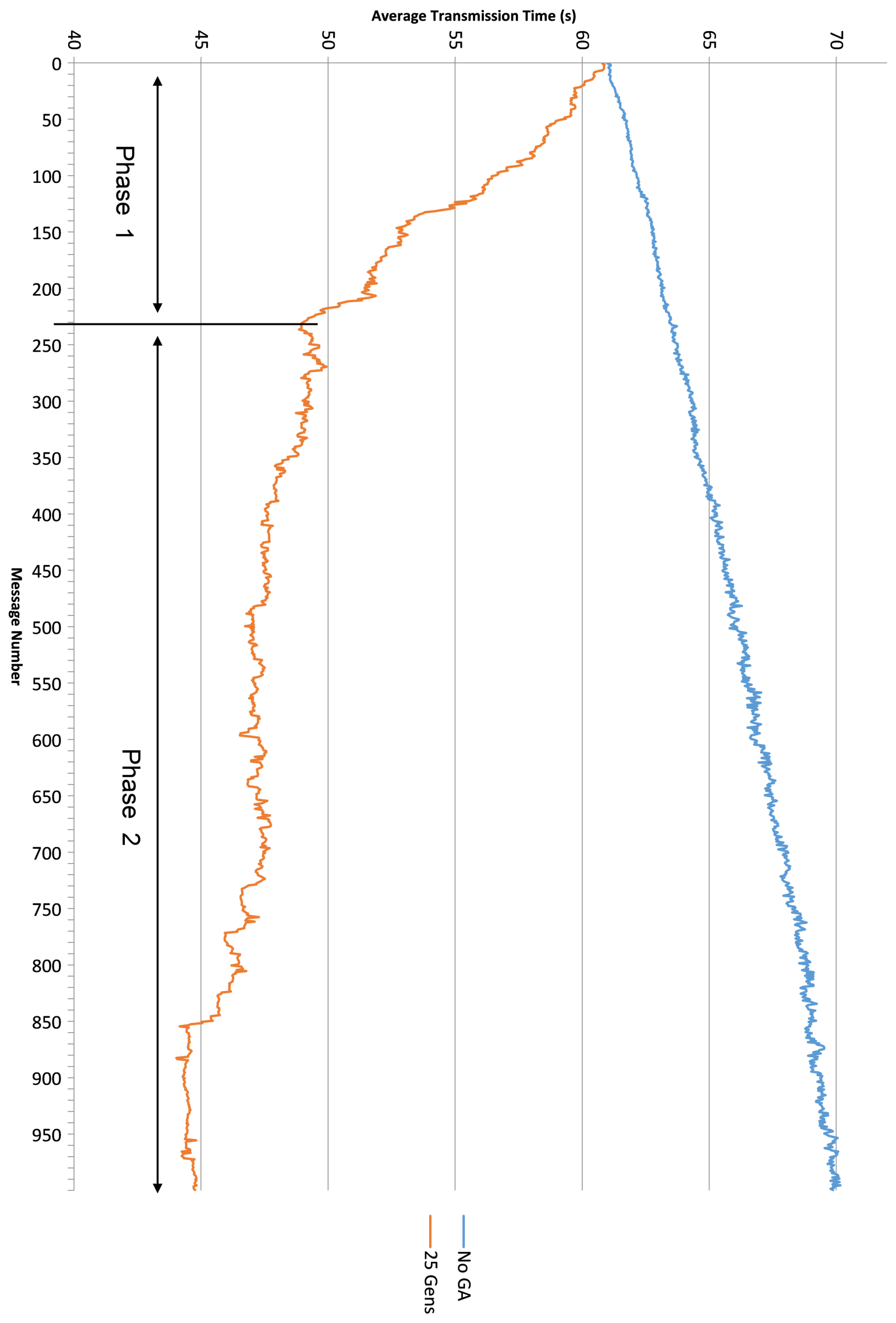

Figure 5.26: Average durations, $N = 64$ network, GA enabled with $G = 25$

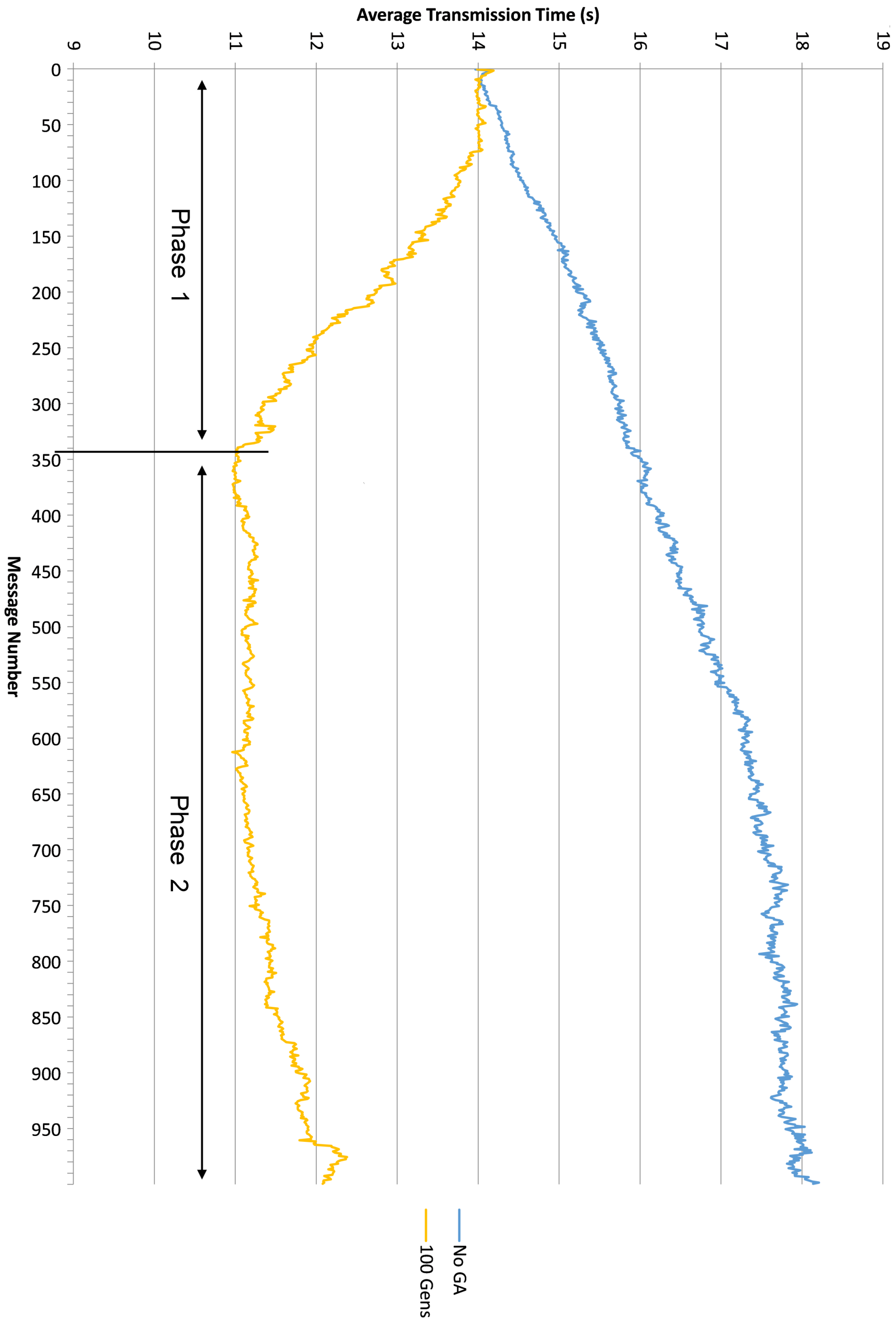

Figure 5.27: Average durations, $N = 32$ network, GA enabled with $G = 100$

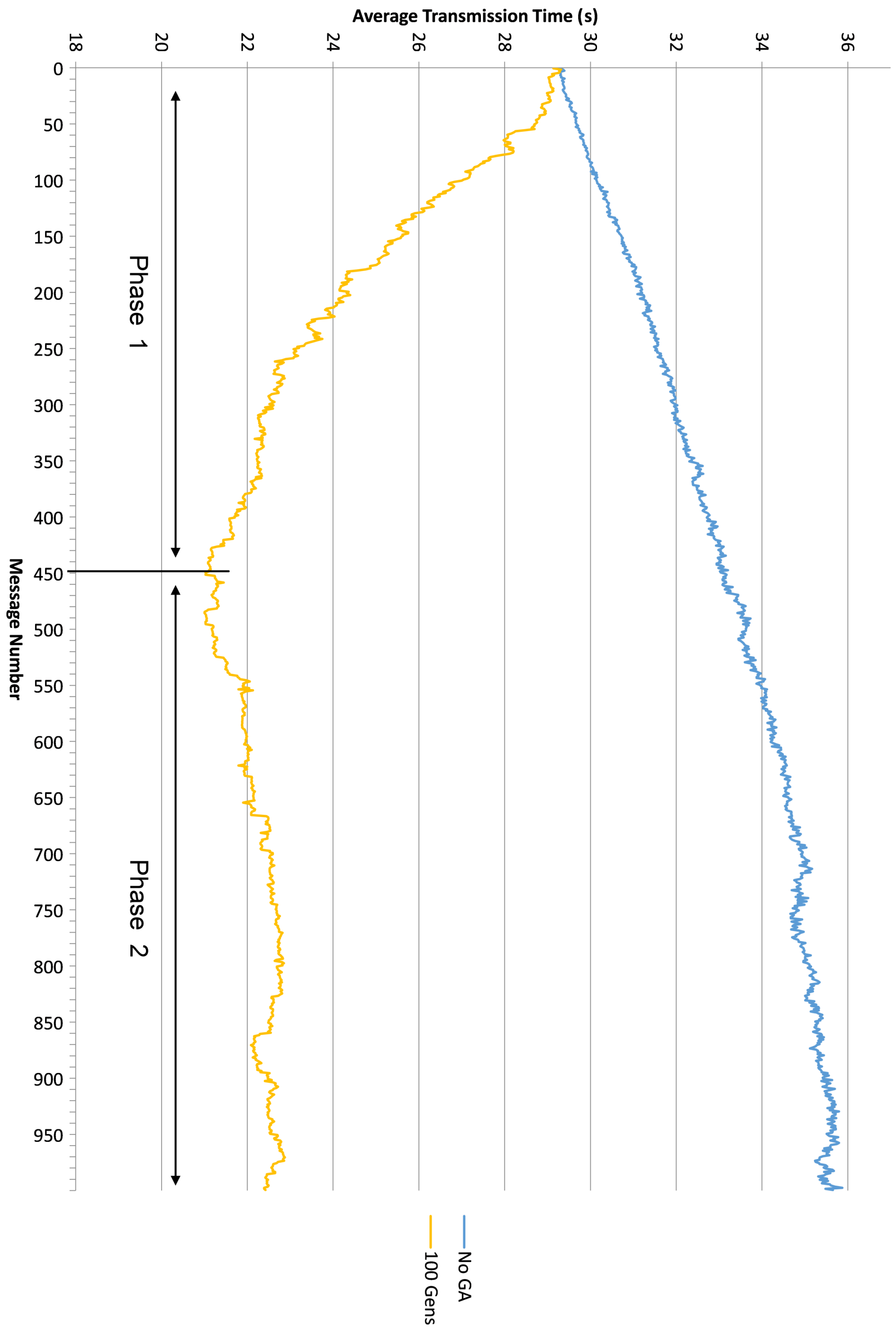

Figure 5.28: Average durations, $N = 64$ network, GA enabled with $G = 100$

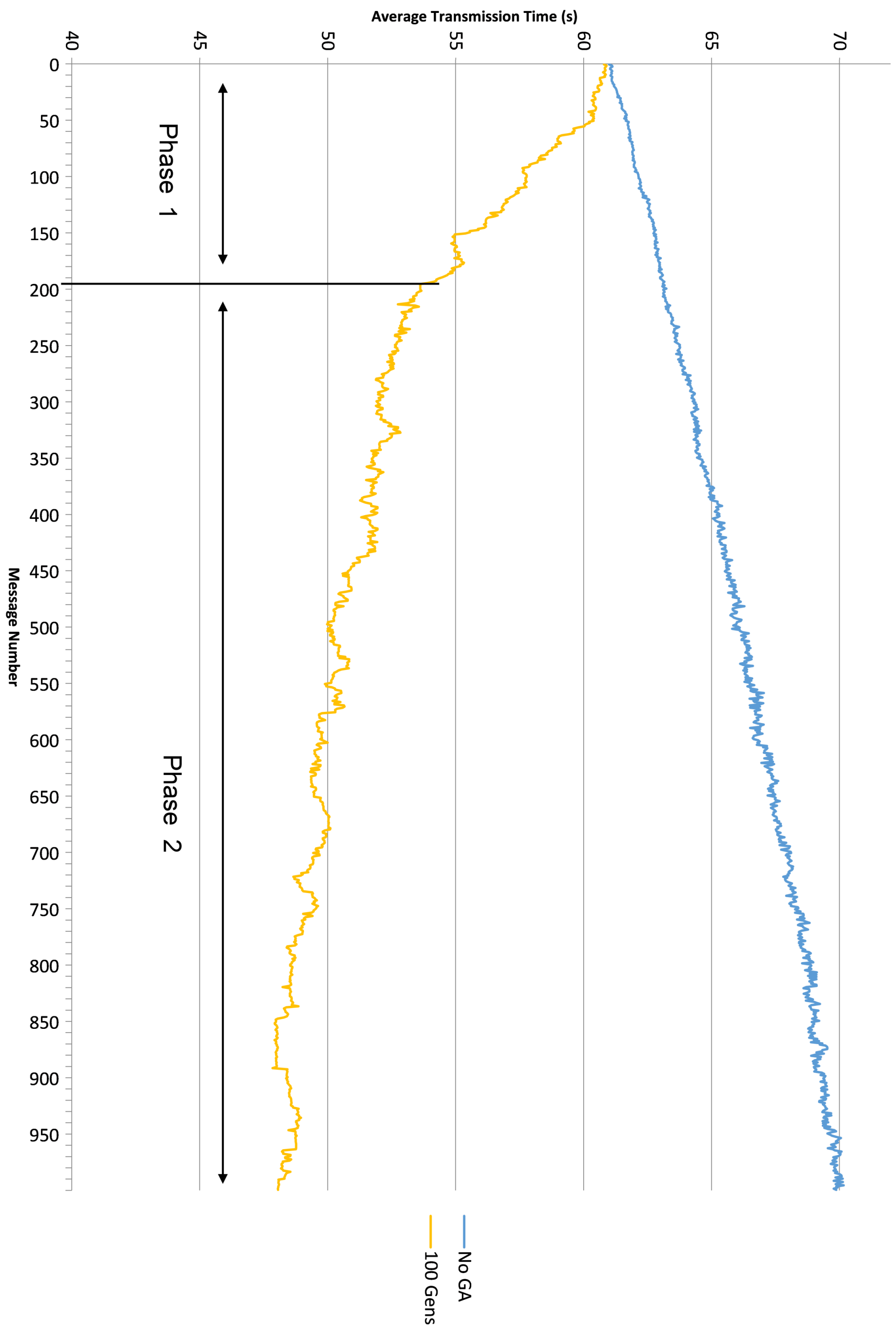

Figure 5.29: Average durations, $N = 64$ network, GA enabled with $G = 100$

same levelling out as in the previous test with $G = 25$, although it never reached the same improvement as the test with $G = 25$. The tests with $G = 100$ showed an initial improvement close to that of the tests with $G = 25$ however their overall improvement decreased later in the tests. The relative effects of the different generations can be seen more clearly on a graph comparing the percentage improvement each test had over the base test with no GA as shown in Figures 5.30, 5.31 and 5.32.

Despite the smaller gains, the test with $G = 5$ can be considered a success. The faster execution cycles of the GA, resulting from the smaller number of generations, allows the GA plan modification component to complete its execution with fewer changes to the environment occurring during that execution time. It also allows the MA to operate with shorter interruptions as the GA executes.

Given the lack of improvement in effectiveness of the test with $G = 100$, increasing the number of generations is considered to be inefficient in this scenario and preference is given to lower generation counts in further experiments.

5.8 Tests with fewer selected plans

In the light of the effectiveness of the test with a small number of generations a further test is done with a reduced number of plans added to the plan library. In previous tests, the $^1/_8$ of the plans with the highest fitness were added to the plan library each time the GA completed its execution. In this test, the number of plans added is the $^1/_{64}$ with the highest fitness. This percentage results in two plans on the $N = 32$ network, four plans on the $N = 64$ network and eight plans on the $N = 128$ network.

As in the initial set of tests, $G = 25$. The results of the tests in this configuration can be seen in Figures 5.33, 5.34 and 5.35.

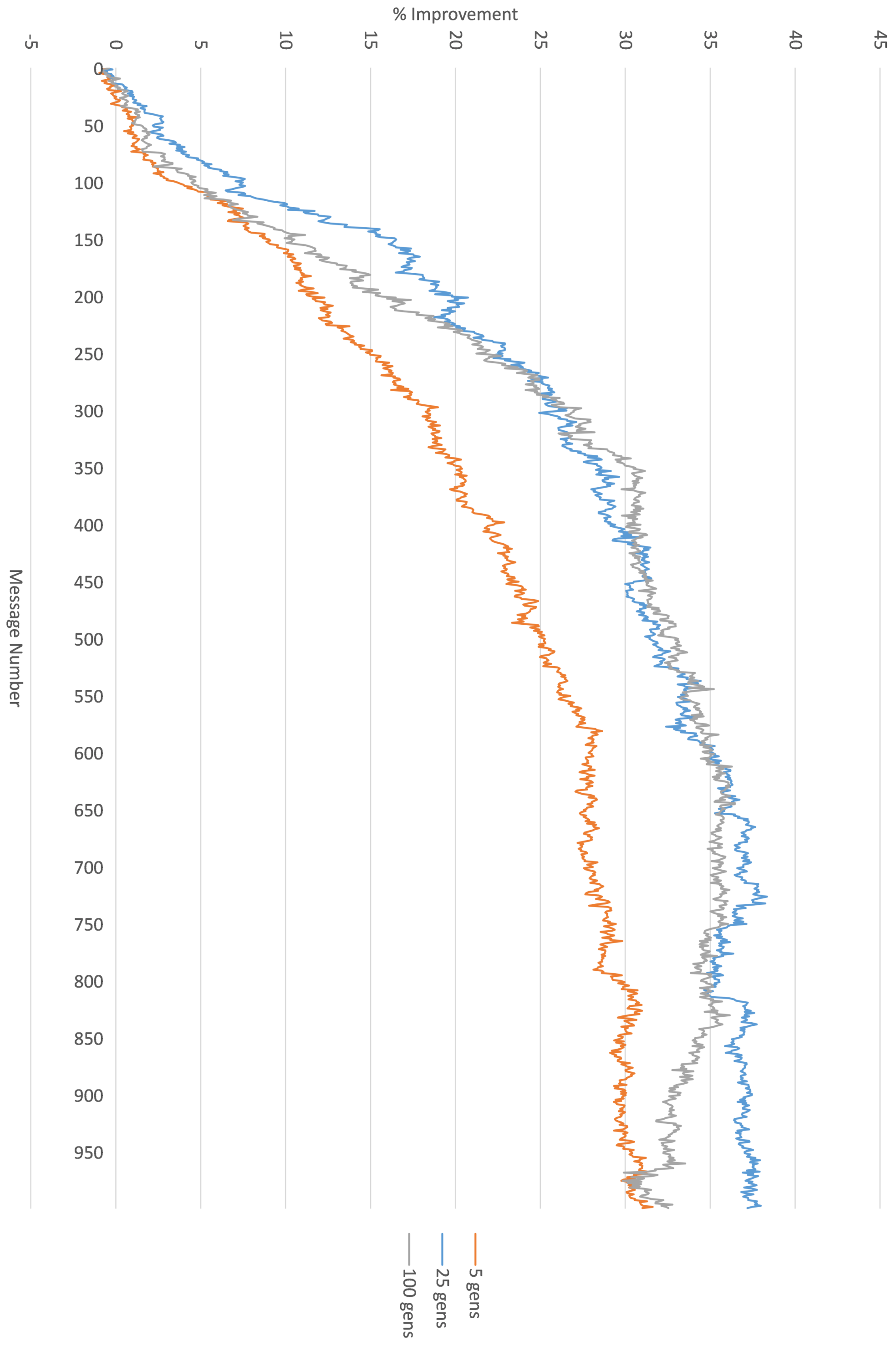

Figure 5.30: Percentage improvement, $N = 32$ network, GA enabled with different number of generations

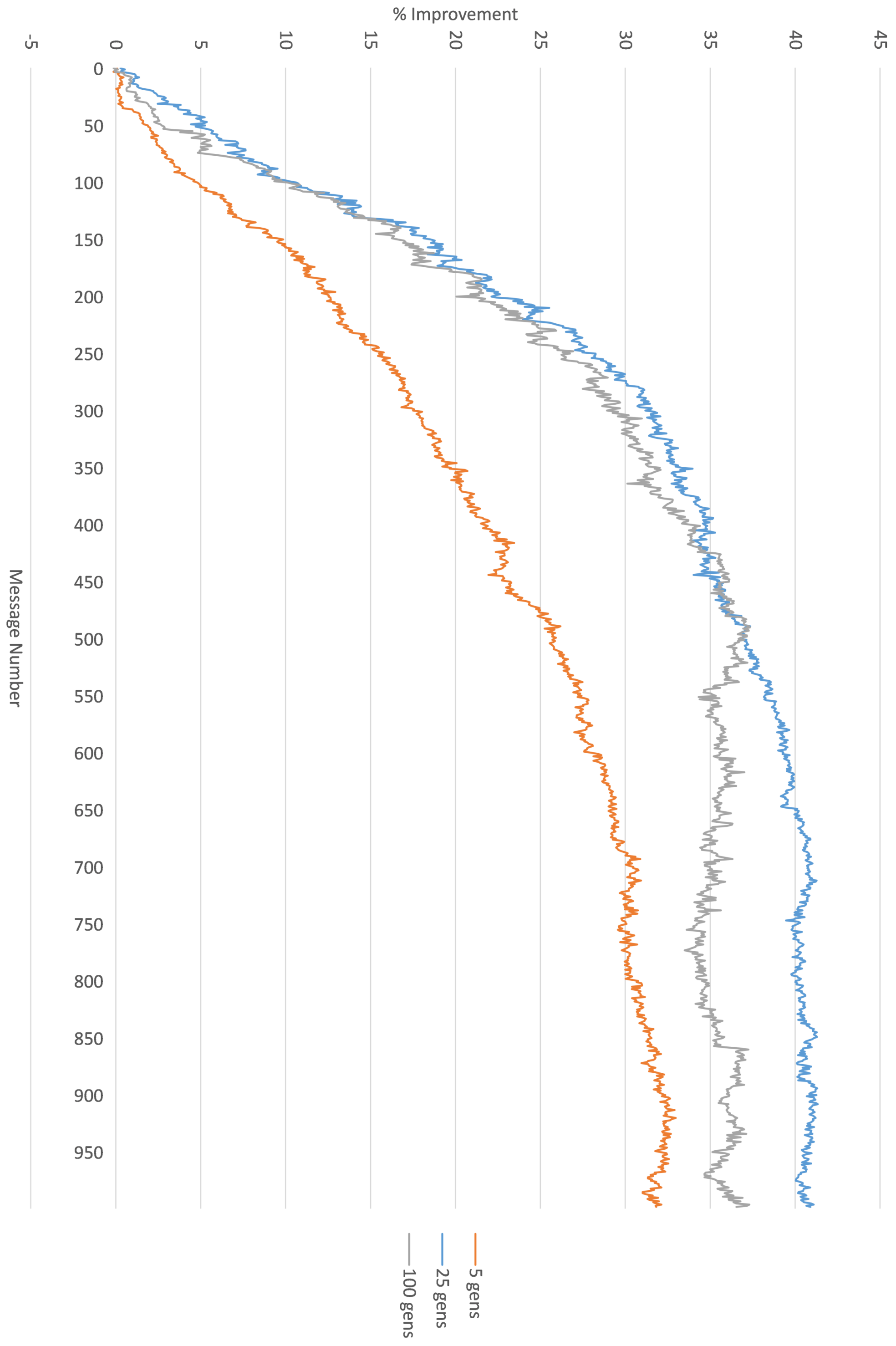

Figure 5.31: Percentage improvement, $N = 64$ network, GA enabled with different number of generations

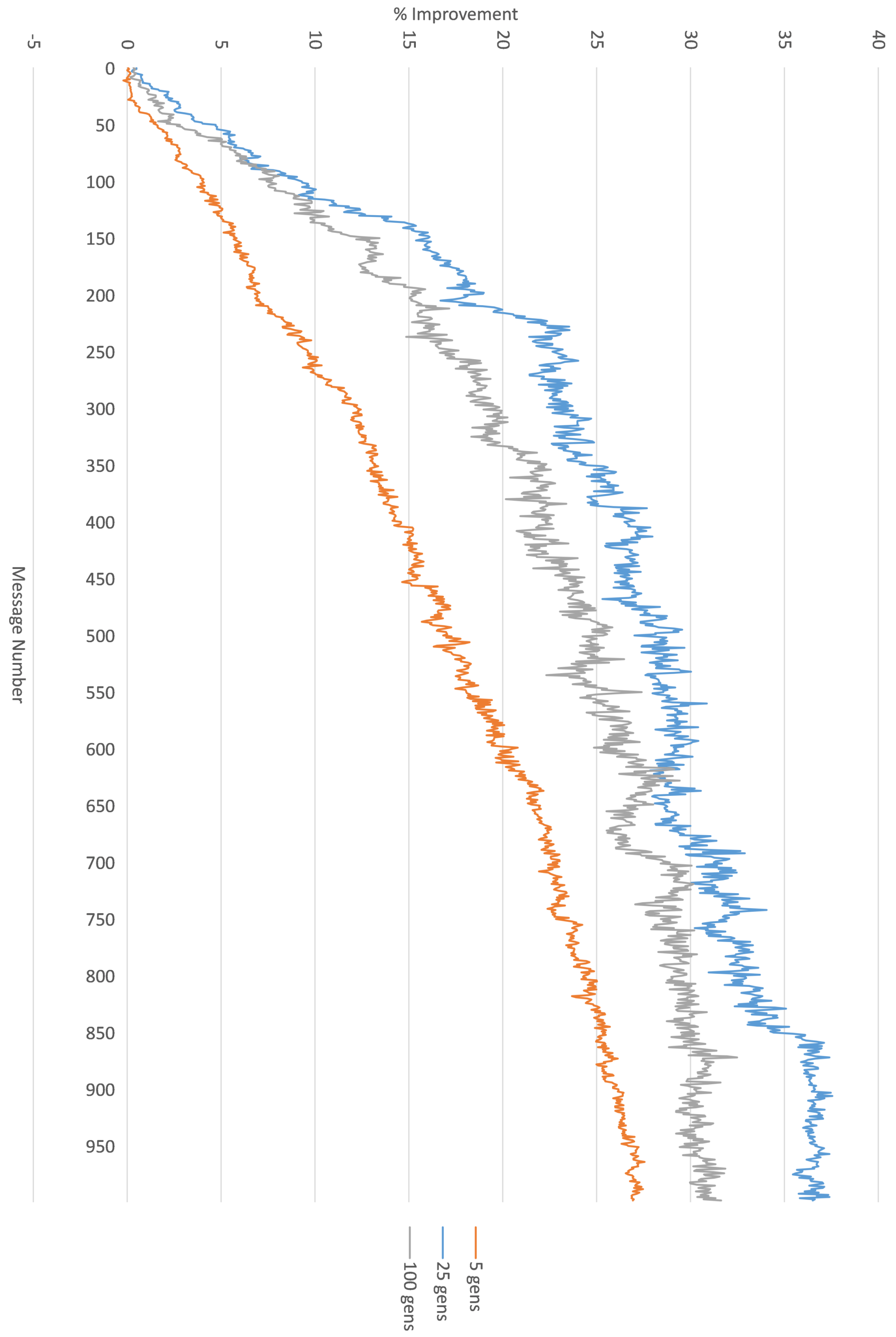

Figure 5.32: Percentage improvement, $N = 128$ network, GA enabled with different number of generations

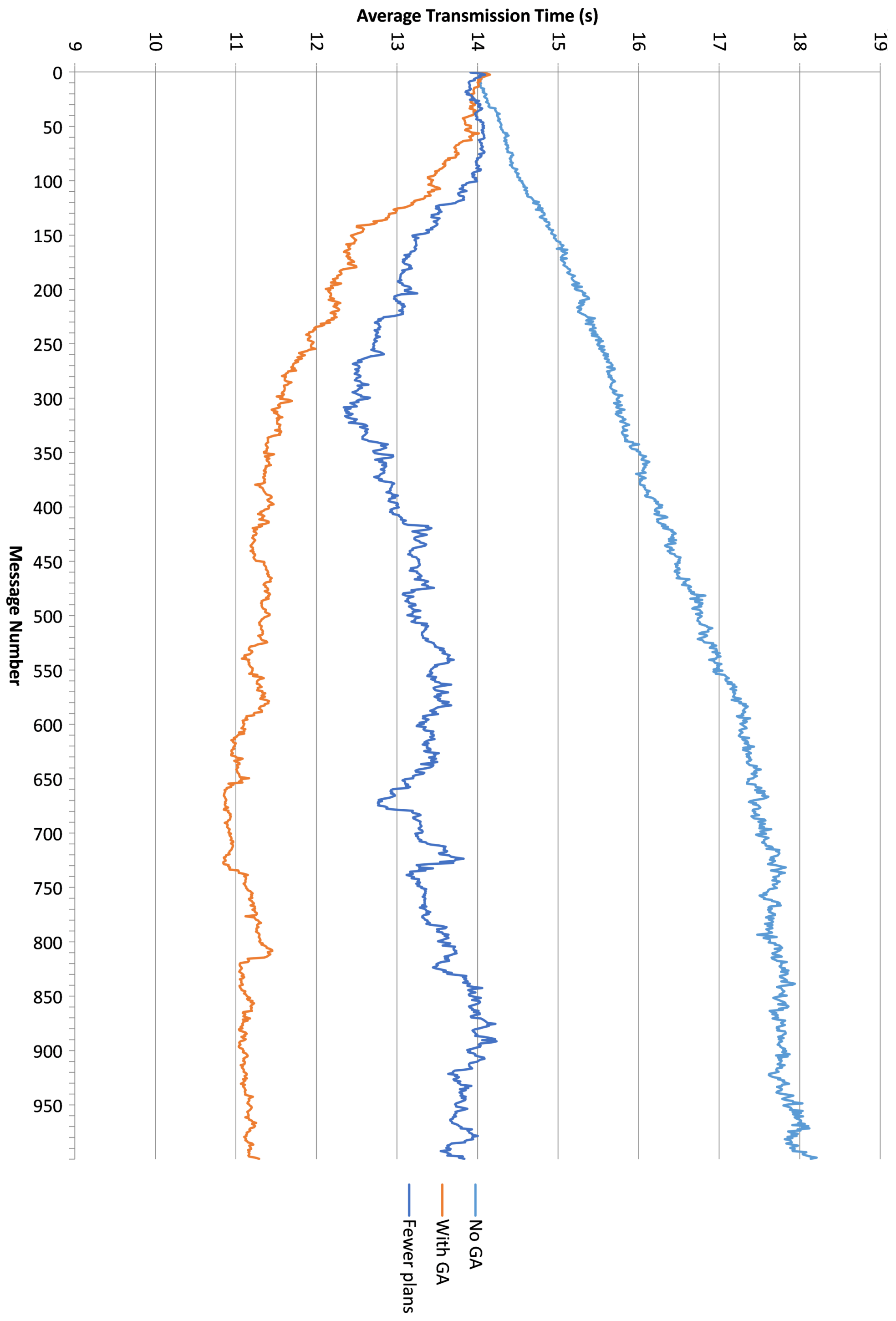

Figure 5.33: Average durations, $N = 32$ network, GA enabled with fewer plans added to the library

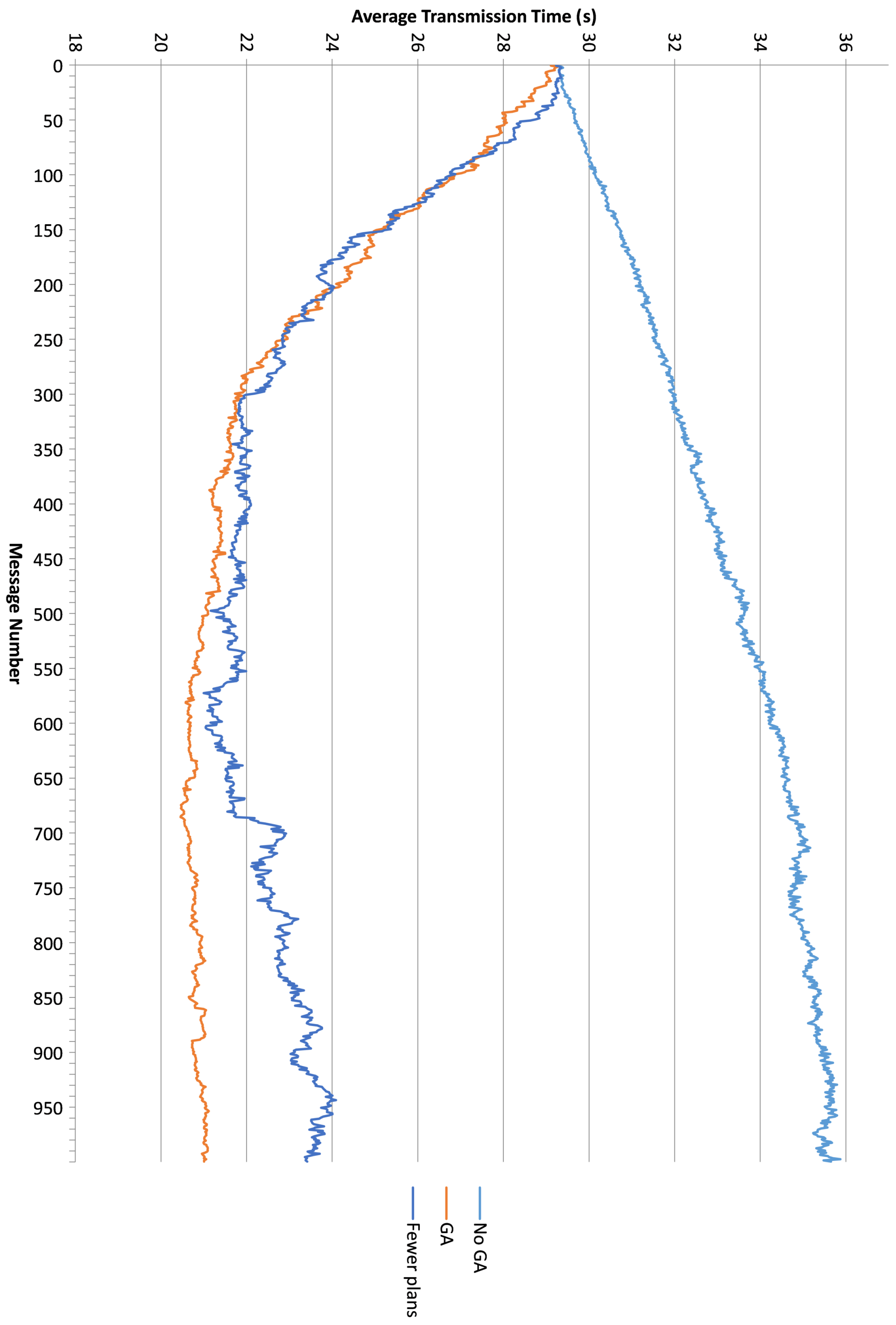

Figure 5.34: Average durations, $N = 64$ network, GA enabled with fewer plans added to the library

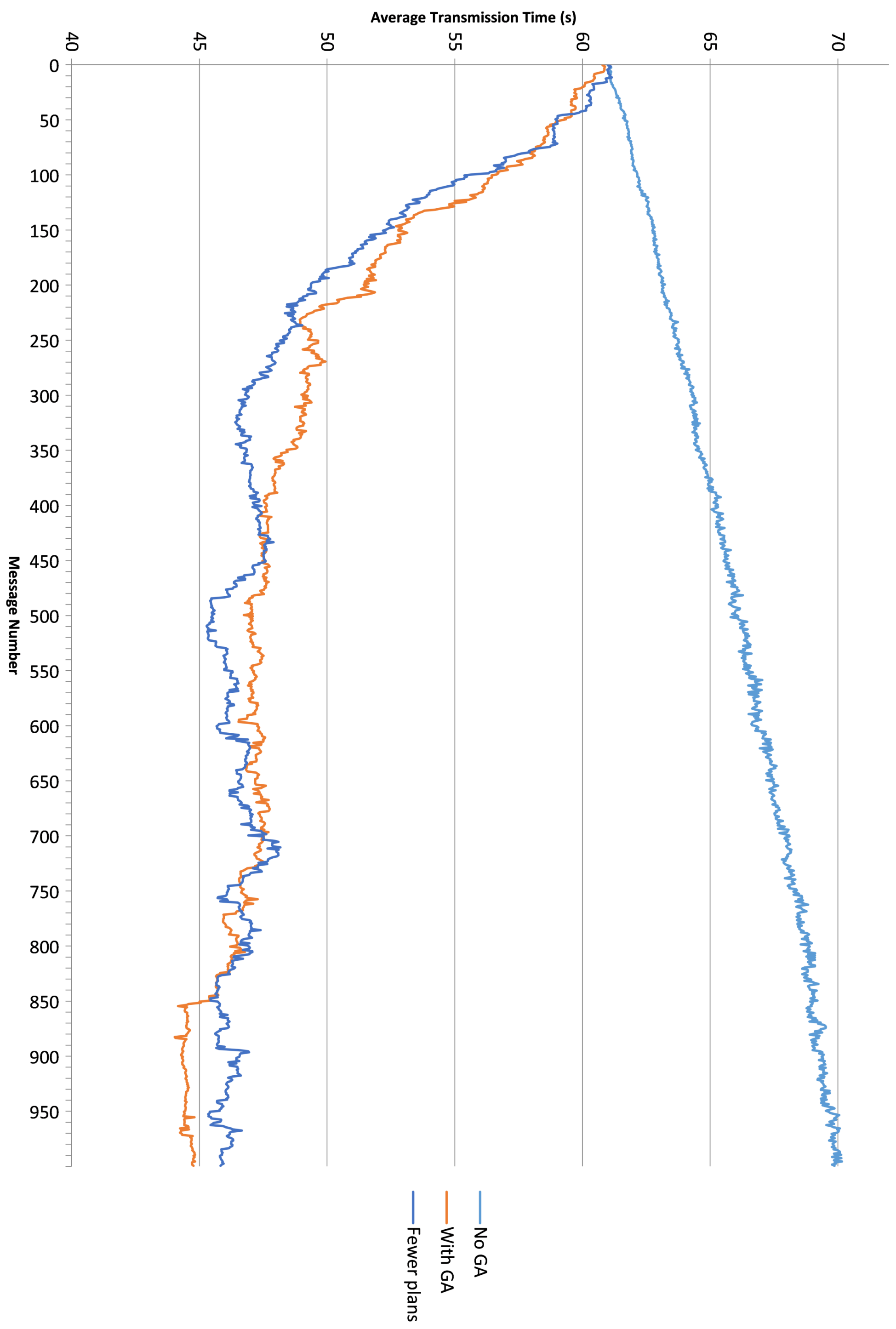

Figure 5.35: Average durations, $N = 128$ network, GA enabled with fewer plans added to the library

The results of this test suggest that it is not necessary to make large changes to the plan library on each execution. Smaller changes may be adequate. It also suggests that adding a single plan on each execution is not as effective as adding multiple plans. The balance between adding too few plans and reducing the effectiveness of the GA plan modification component and adding too many plans and bloating the plan library is likely to differ from one scenario to another. The ideal number of plans added would have to be determined on a case-by-case basis if using the GA plan modification mechanism in a real situation.

On the $N = 32$ network adding a smaller number of plans has a measurable negative effect on the efficiency of the MA compared with previous configurations of the GA. The relative improvement achieved is shown in Figures 5.36, 5.37 and 5.38. With the $N = 32$ network the test with reduced number of plans reached a maximum improvement of 26% compared with the 38% which the test with the larger number of selected plans reached, marked as “1” on Figure 5.36.

In the $N = 32$ network the GA operates on a population size of 128 individuals. With this number and the lower percentage of selected plans, only two plans are added after each invocation of the GA. As such, the ongoing changes to the environment are likely to make these two plans inefficient or invalid faster than if a larger number of plans are added. This could be avoided by adding a larger percentage of plans on smaller networks and smaller percentage on larger. An alternative option is to add a fixed number of plans rather than a fixed portion of the starting population size, for example, adding eight plans to the plan library after each invocation of the GA, rather than a number of plans dependent on the size of the network.

On the $N = 64$ and $N = 128$ networks, the test with a reduced number of plans performed better. On the $N = 64$ network the test with reduced

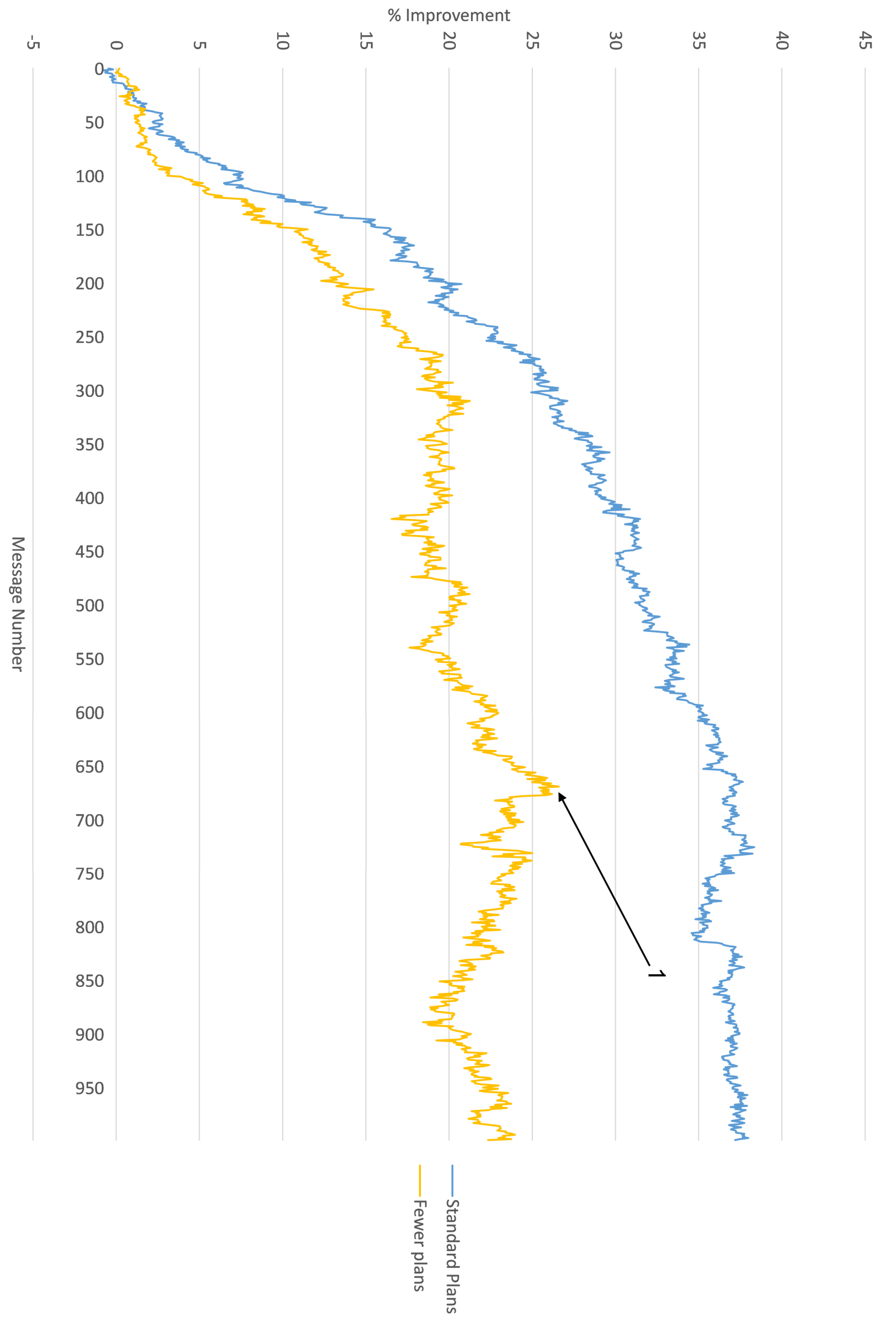

Figure 5.36: Percentage improvement over test without GA, $N = 32$ network, GA enabled with fewer plans

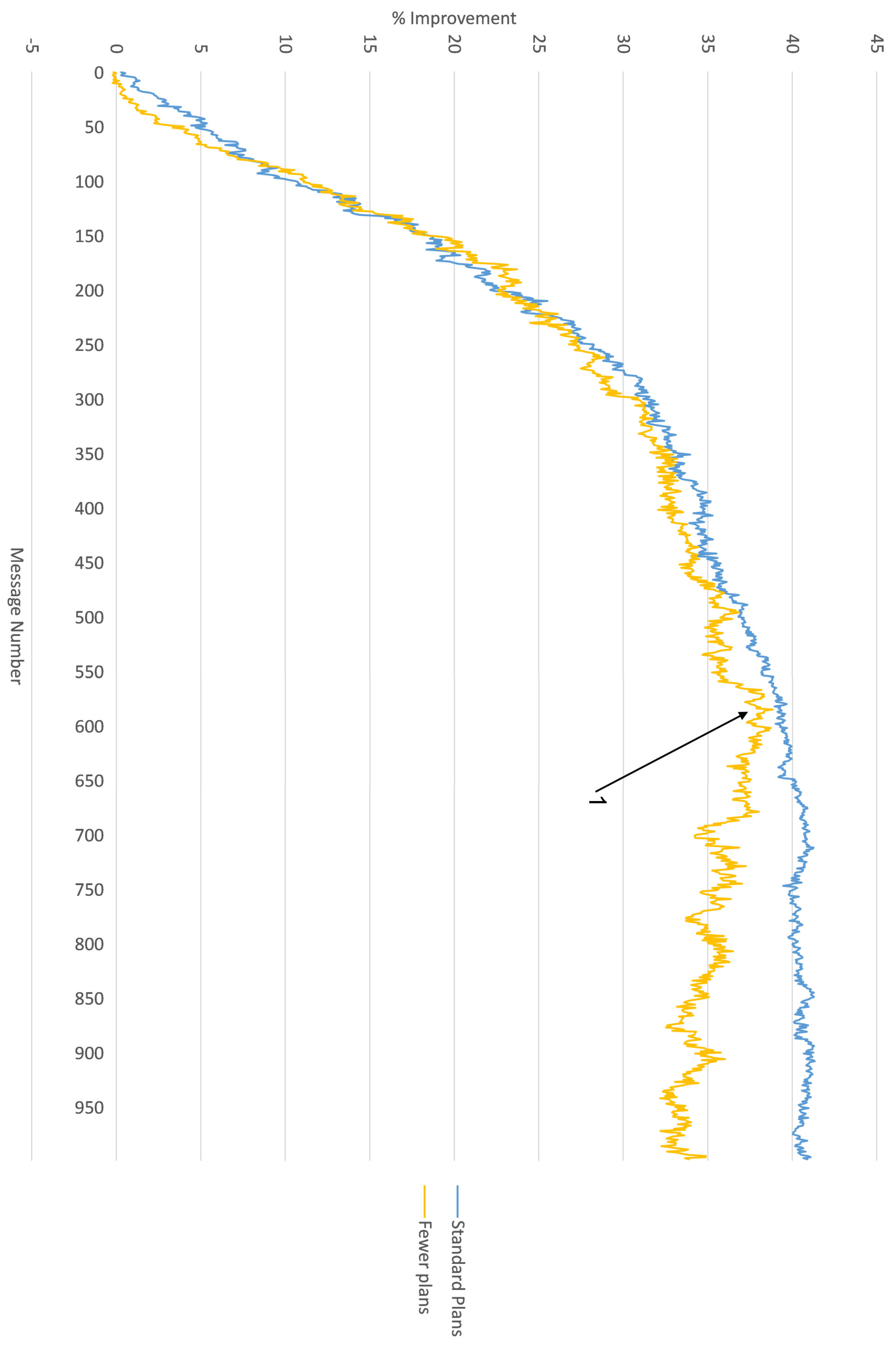

Figure 5.37: Percentage improvement over test without GA, $N = 64$ network, GA enabled with fewer plans

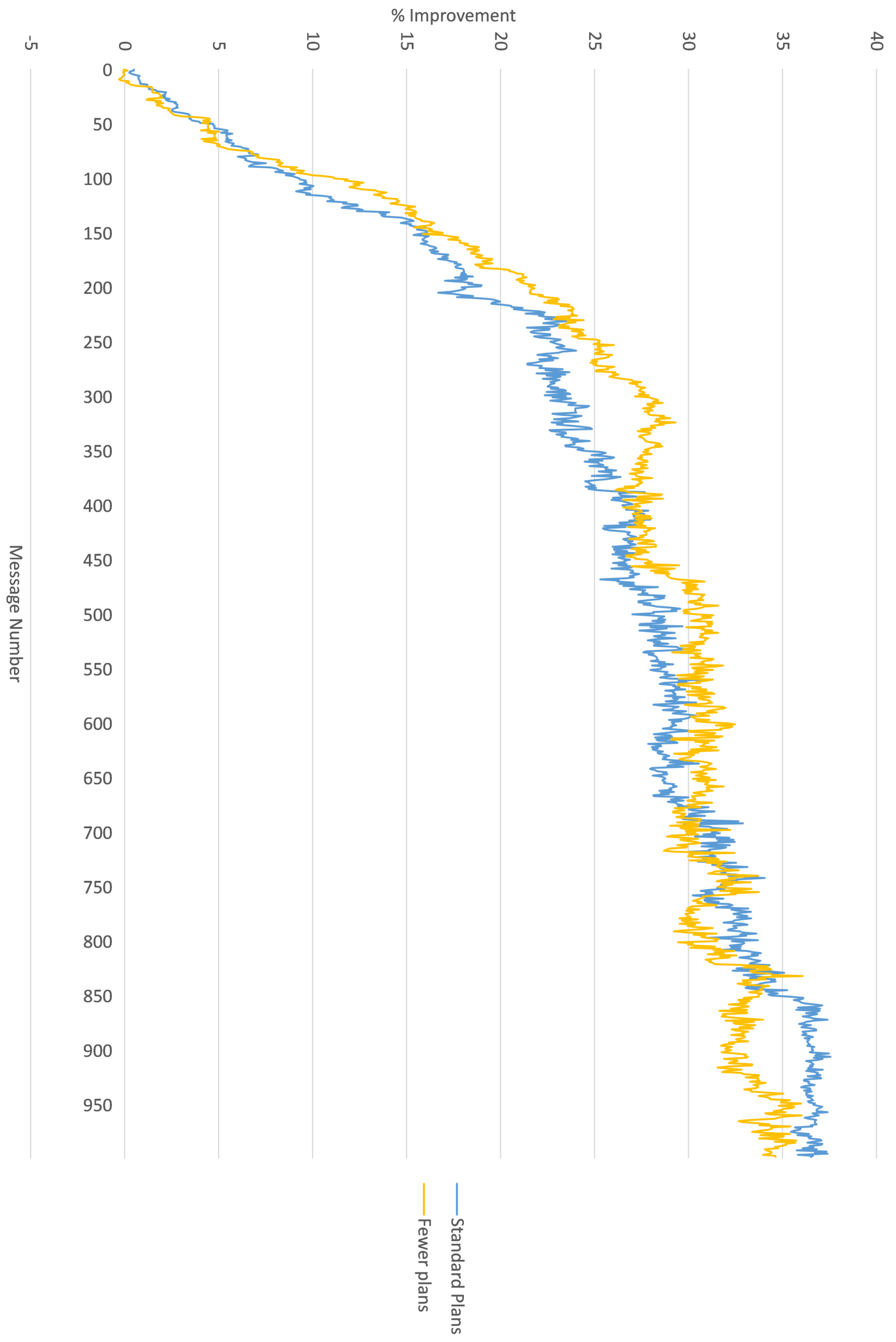

Figure 5.38: Percentage improvement over test without GA, $N = 128$ network, GA with fewer plans

number of plans performed comparably with the previous test until around message number 580, indicated by “1” on Figure 5.37. After this point the performance degraded, with the percentage improvement over the base test without GA decreasing for the remainder of the test.

On the $N = 128$ network, the test with fewer plans performed comparably with the test with the larger number of plans across the entire 1000 messages.

5.9 Tests with scheduled executions of the GA

In all of the previous configurations, the trigger for the invocation of the GA is a fixed number of messages failing to be delivered, specifically three. The reasons why messages fail are discussed in Section 4.2.3.

Since the plans are valid at the time they are created, a failure of a plan indicates that the environment has changed sufficiently so that plans which were valid at the time they were created are now failing.

The potential problem is that the metric used to decide when to invoke the GA is not the same one that the GA is optimising to minimise. The GA is optimising for reducing t_{msg}, not for reducing failures.

In this series of tests the GA is invoked at scheduled times. The first set of tests had the GA invoked after 50 messages have been sent, regardless of how many message failures had occurred. The second set of tests had the GA invoked after 20 messages. The results for this series of tests can be seen in Figures 5.39, 5.40 and 5.41.

The sharp drop seen in Figures 5.39, 5.40 and 5.41 occurs at the first scheduled execution of the GA. This is at 50 messages for the slower scheduled GA and message number 20 for the faster scheduled GA. The average

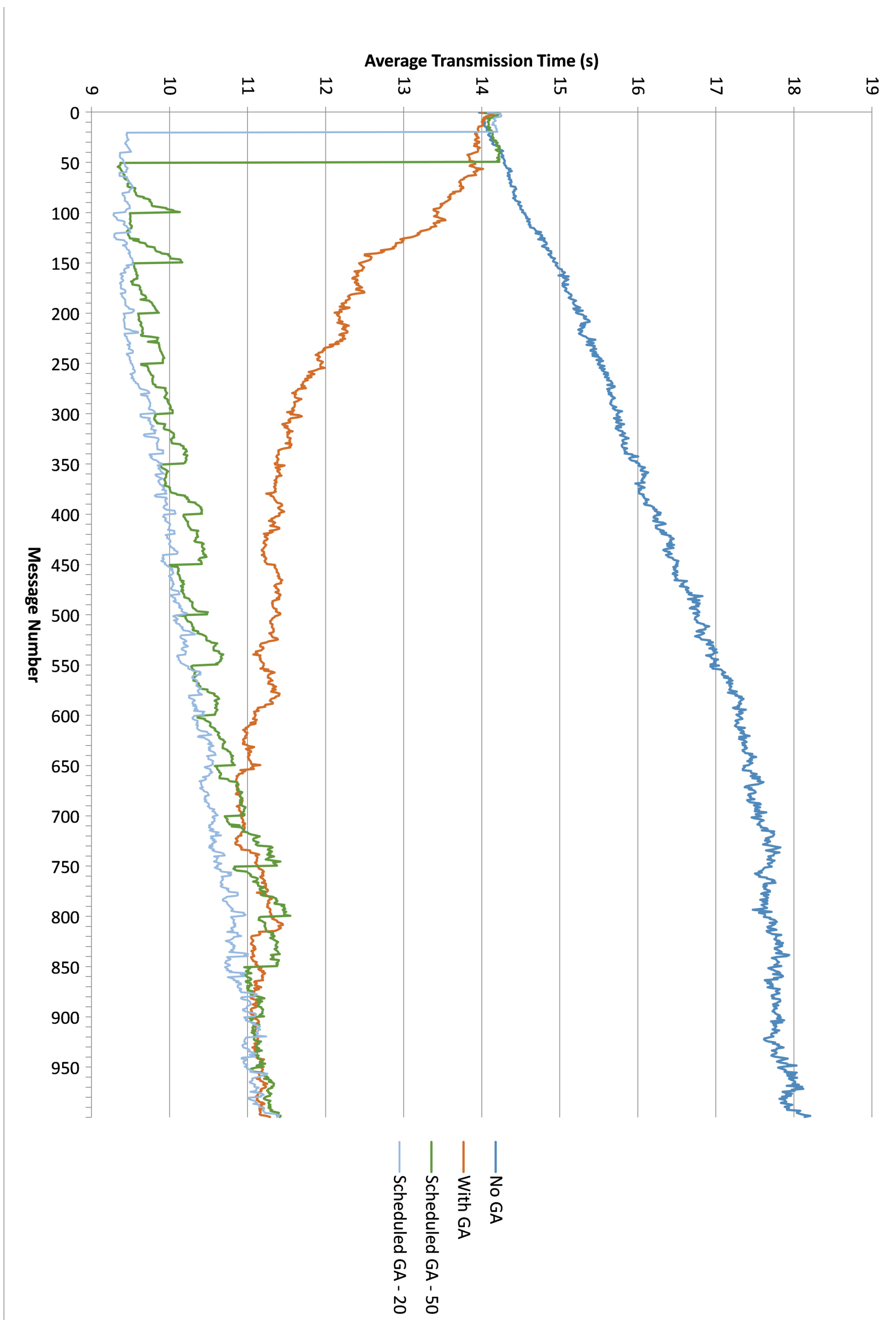

Figure 5.39: Average durations, $N = 32$ network. Scheduled invocations of the GA

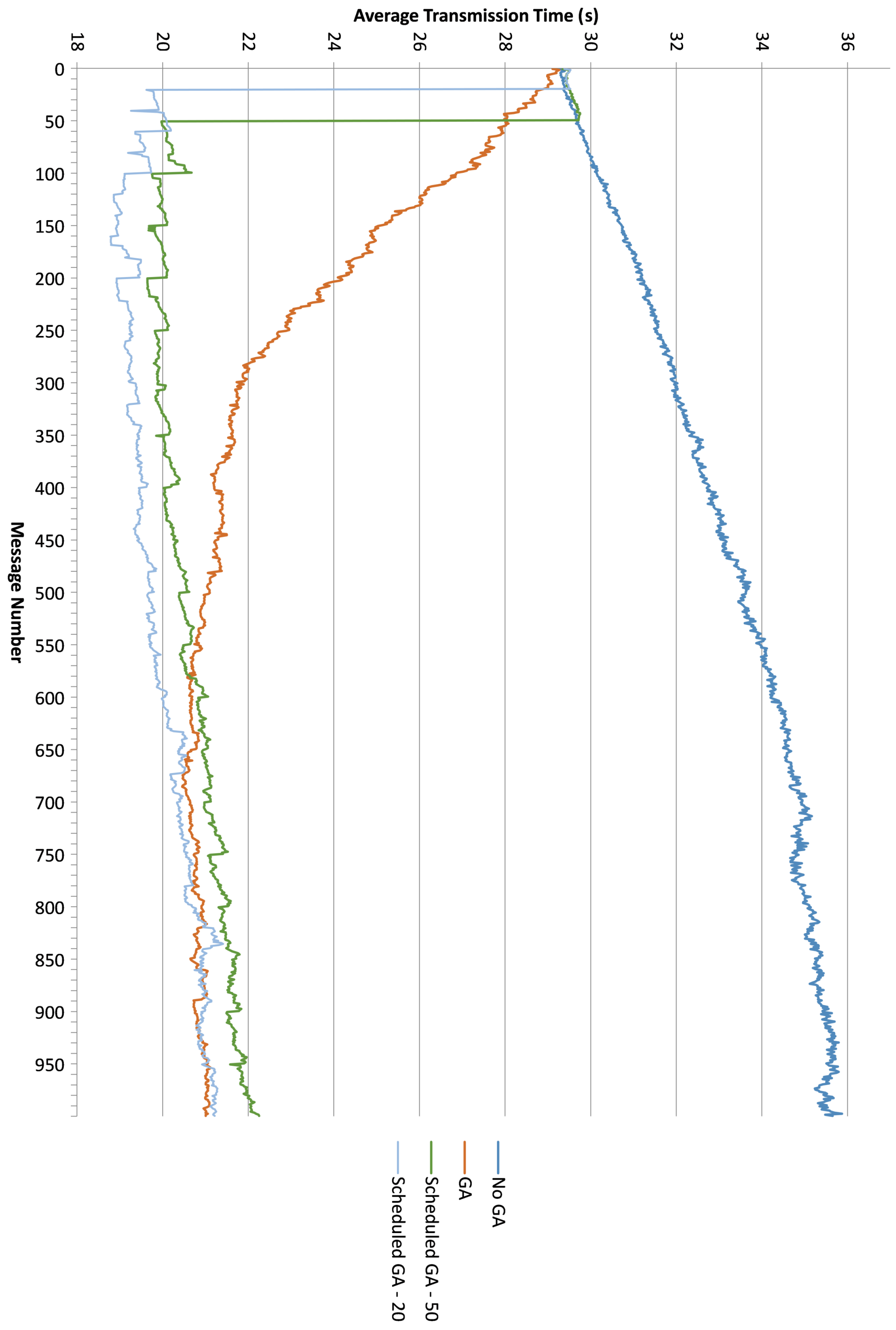

Figure 5.40: Average durations, $N = 64$ network. Scheduled invocations of the GA

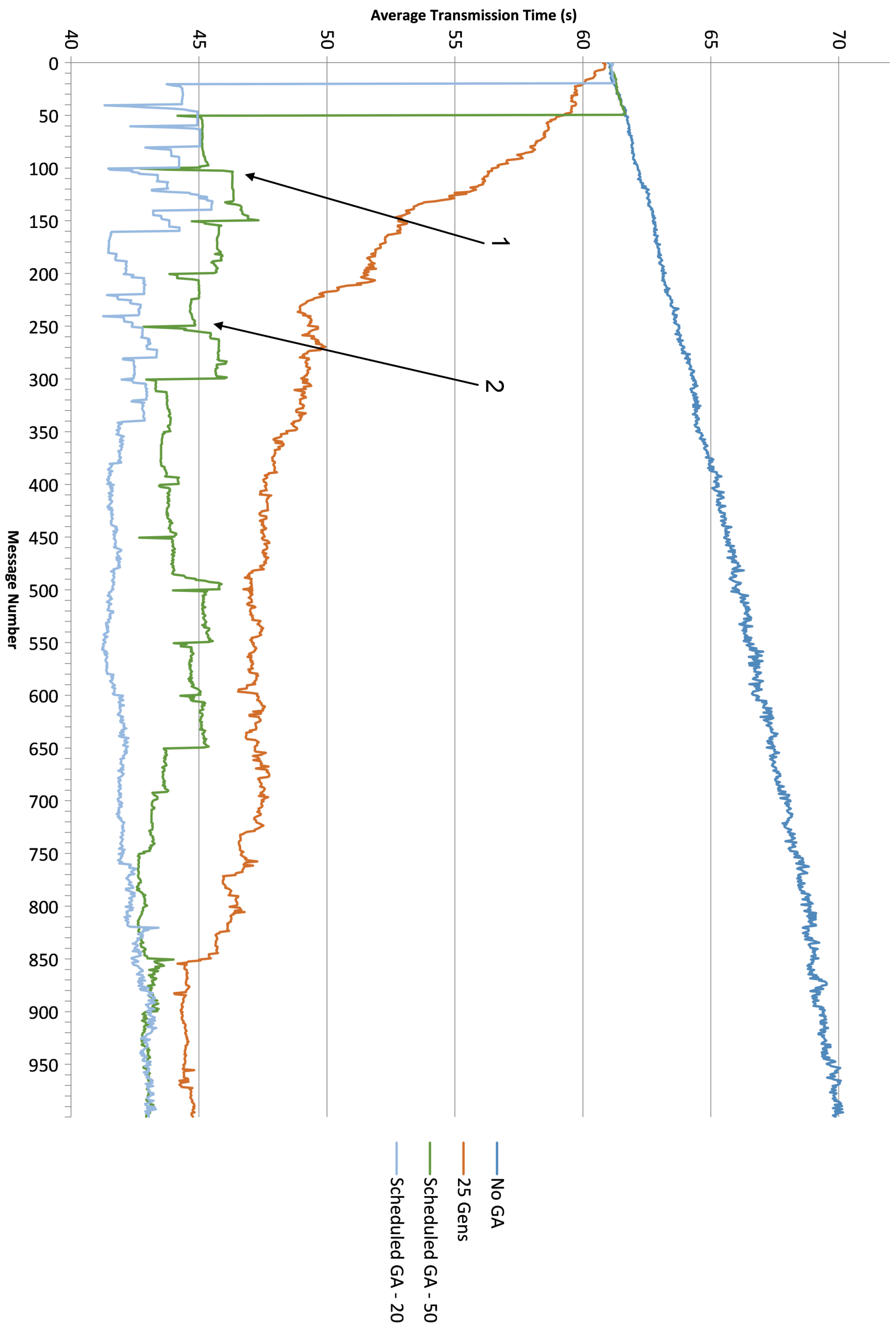

Figure 5.41: Average durations, $N = 128$ network. Scheduled invocations of the GA

rate of change of t_{avg} after that first sharp decrease is detailed in Table 5.9

Network size	Average rate of change, GA after every 50 messages	Average rate of change, GA after every 20 messages
32	0.002	0.002
64	0.002	0.002
128	0.001	-0.001

Table 5.9: Rate of change in seconds/message

The alternate analysis, graphing the percentage improvement, is shown in Figures 5.42, 5.43 and 5.44.

The sharp changes at intervals of 50 messages in the first set of tests and at 20 messages in the second set of tests occur because in this configuration the GA is invoked at the same point for each of the 100 tests. In previous configurations the GA was invoked after a fixed number of messages failed delivery. As such, the point at which the GA was invoked differed across the 100 tests and the decrease in message delivery time appears as a steady decrease once the average was taken.

It can be seen, especially on the largest network, that the GA is not always equally effective. After some optimisations the resultant plans are better as after others, in that they result in lower t_{avg}. This is especially noticeable at message number 100 and again at message 250, indicated by points “1” and “2” on Figure 5.41.

The varying effectiveness would be a result of the environment changing while the GA is active. The larger network shows this effect best because the GA has the largest population with this network size and hence the GA takes the longest to execute out of all the network sizes. As such the network has the highest chance of changing during this execution. This

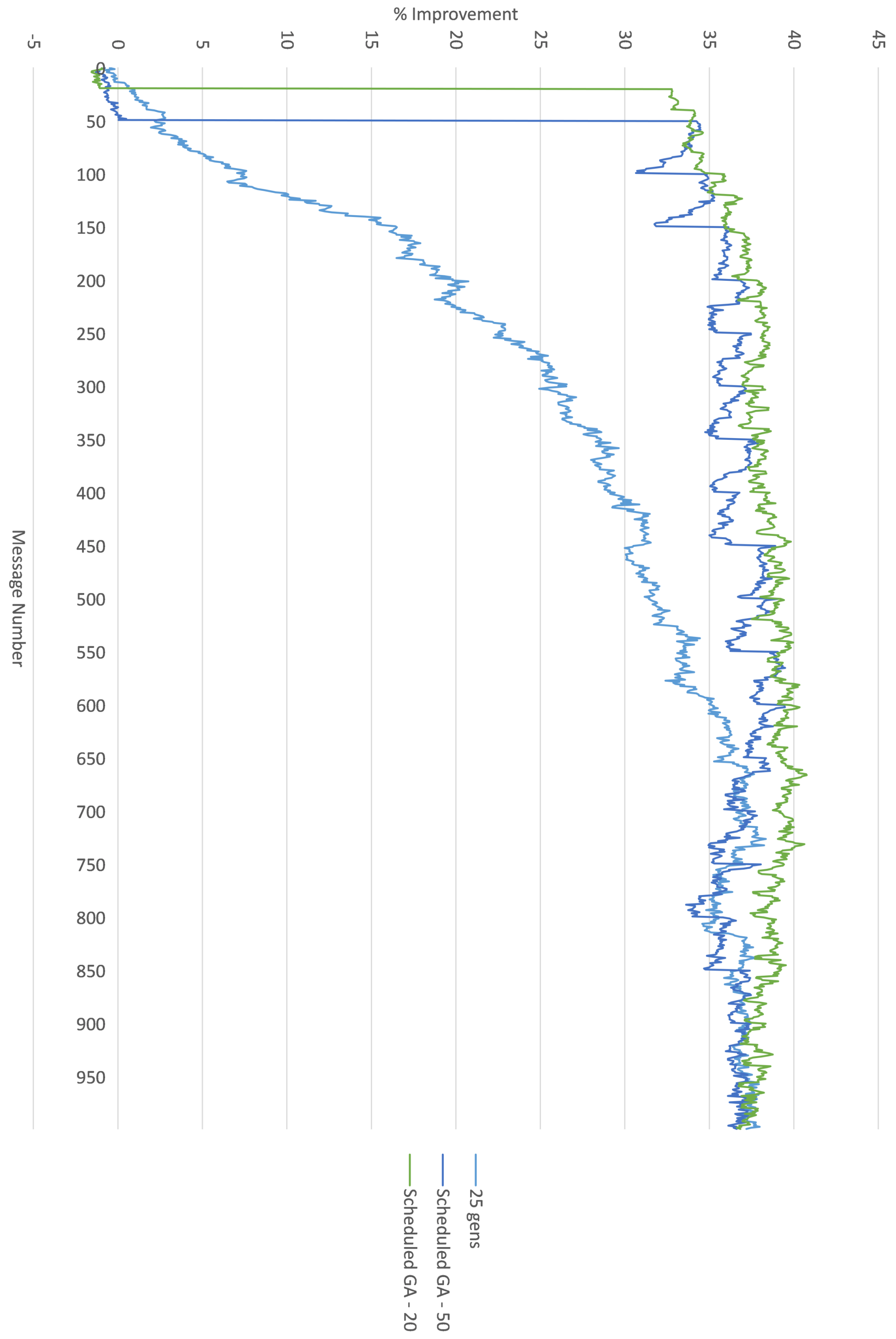

Figure 5.42: Percentage improvement over test without GA, $N = 32$ network, Scheduled GA

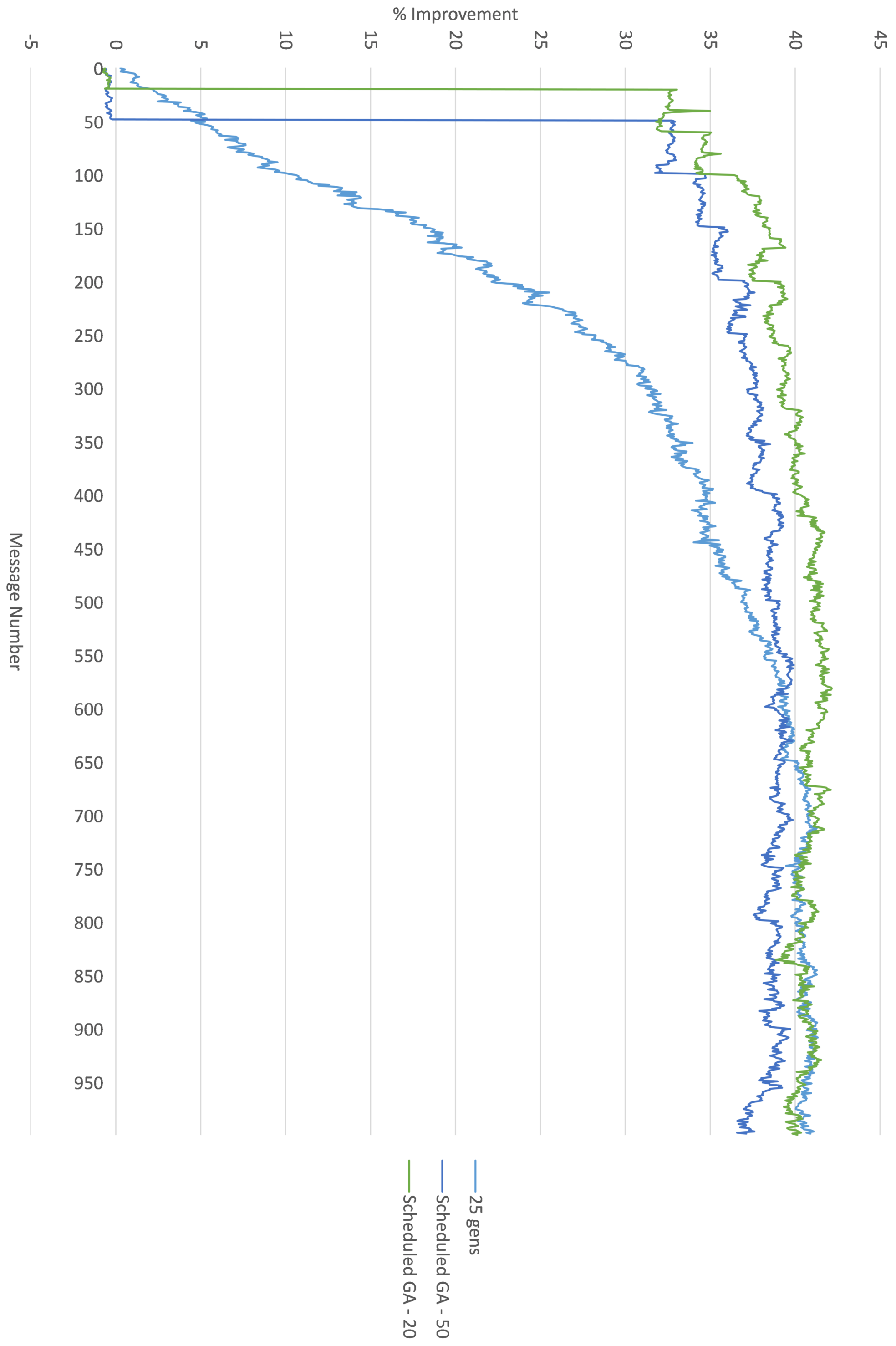

Figure 5.43: Percentage improvement over test without GA, $N = 64$ network, Scheduled GA

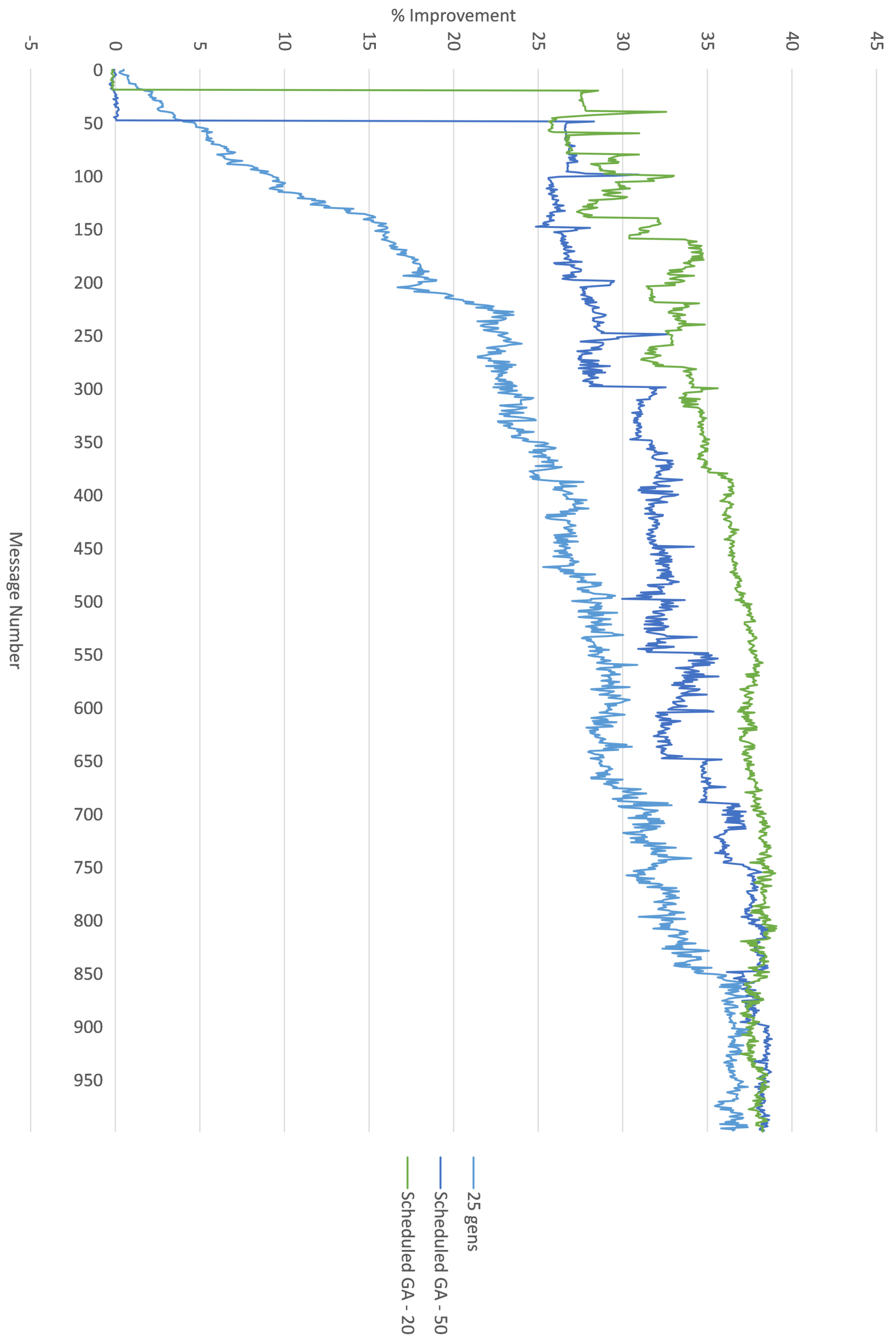

Figure 5.44: Percentage improvement over test without GA, $N = 128$ network, Scheduled GA

is because the network changes occur at a constant rate, the longer the GA takes to execution, the more changes would have occurred during that execution.

5.10 Effects of different rates of environment change

One of the conditions of the hypothesis investigated in this dissertation is that the environment is slowly changing. “Slowly” is defined as relative to the execution time of the GA plan modification component. The idea is that the GA plan modification component has time to run the required number of evolutions, add new plans to the plan library and those plans still be valid and efficient when the MA goes to use them.

As the rate of change of environment, C, increases, the efficiency of the GA plan modification component is expected to decrease. If it is triggered by a set number message failures, as the first four sets of tests were, the more rapidly changing environment is expected to result in more messages failing delivery. These failures, occurring more frequently, trigger the GA plan modification component more frequently, increasing the overhead of the GA.

To determine whether those expectations are true, the test with $N = 64$ and $G = 25$ was re-run with higher values of C. C is defined for this test as the number of times which the modification code in Listing 4.3 is executed for each node in the network.

For example with $C = 2$, when the environment change routines, described in Section 4.2.2 run, for each node in the network the code in Listing 4.3 is executed twice. Similarly if $C = 4$ then the code in Listing 4.3 will be executed four times.

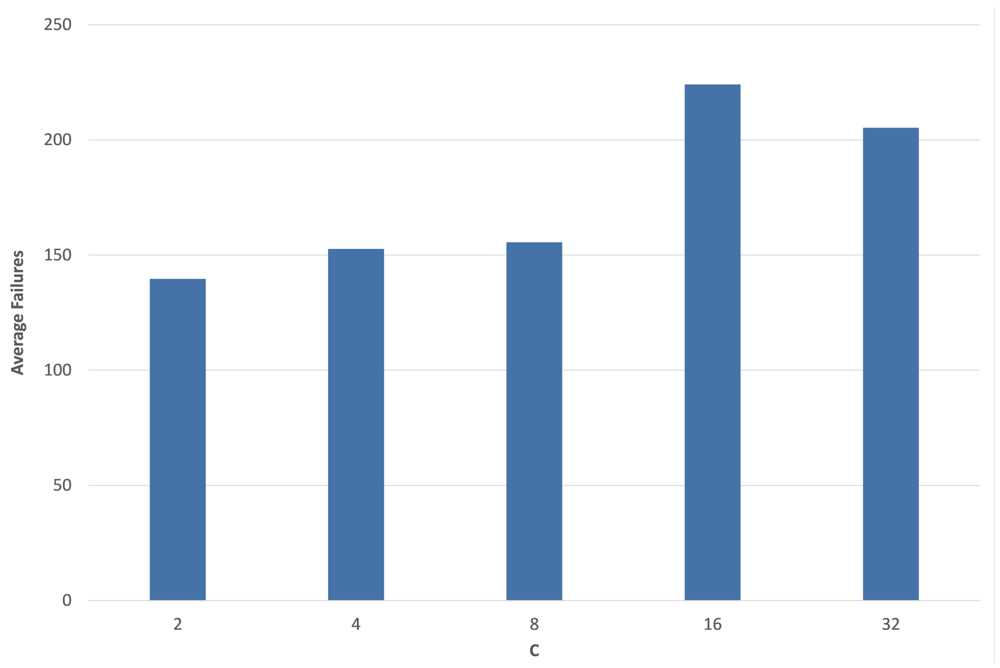

Figure 5.45: Increase in number of failures, and hence invocations of the GA, as the environment changes faster

C	Average Number of Failures
2	140
4	153
8	156
16	224
32	205

Table 5.10: Average failures at higher rates of change

The average number of failures over the 1000 messages increased as the environment changes proceeded at higher rates, shown in Figure 5.45 and Table 5.10.

The effectiveness of the GA decreased as well, as shown in Figures 5.46, 5.47, 5.48, 5.49 and 5.50.

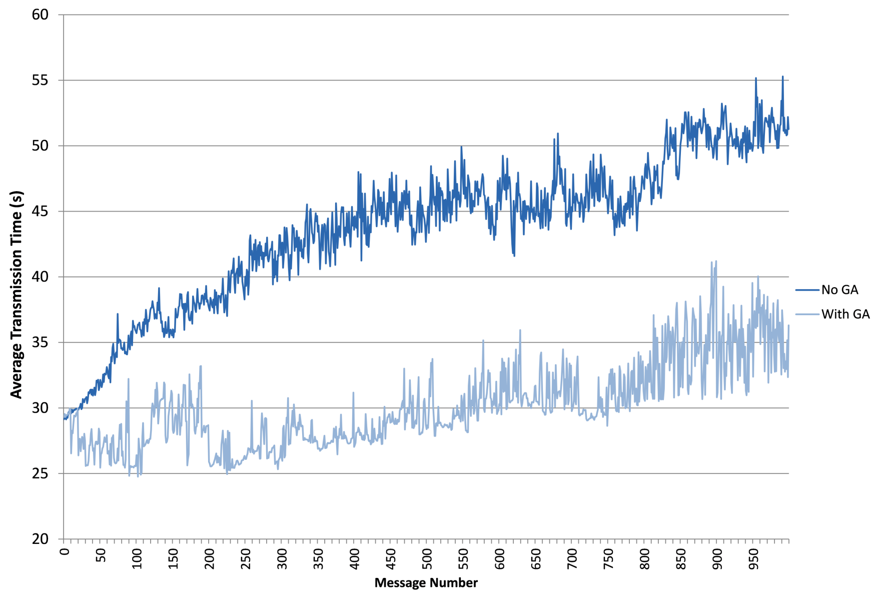

Figure 5.46: The performance of the agent with and without the GA plan modification mechanism, $C = 2$

As the rate of change of the environment increased, from $C = 2$ up to $C = 32$, the GA became less effective, despite more frequent invocations of the GA coming from the higher rate of failures. This decrease in effectiveness is especially noticeable at $C = 16$ and $C = 32$.

From this it is clear that the use of a GA plan modification mechanism becomes less suitable as the environment changes faster, confirming this as a plan modification mechanism for slowly changing environments only.

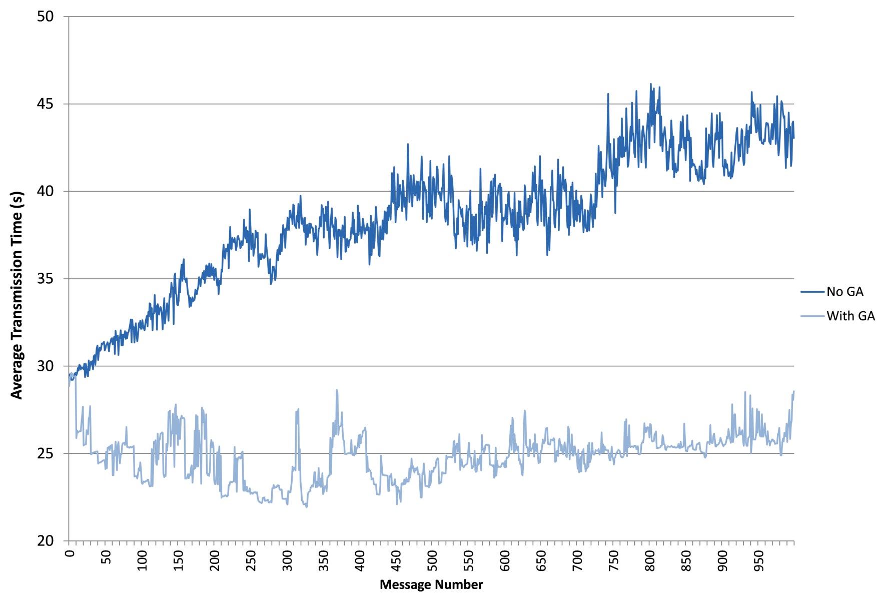

Figure 5.47: The performance of the agent with and without the GA plan modification mechanism, $C = 4$

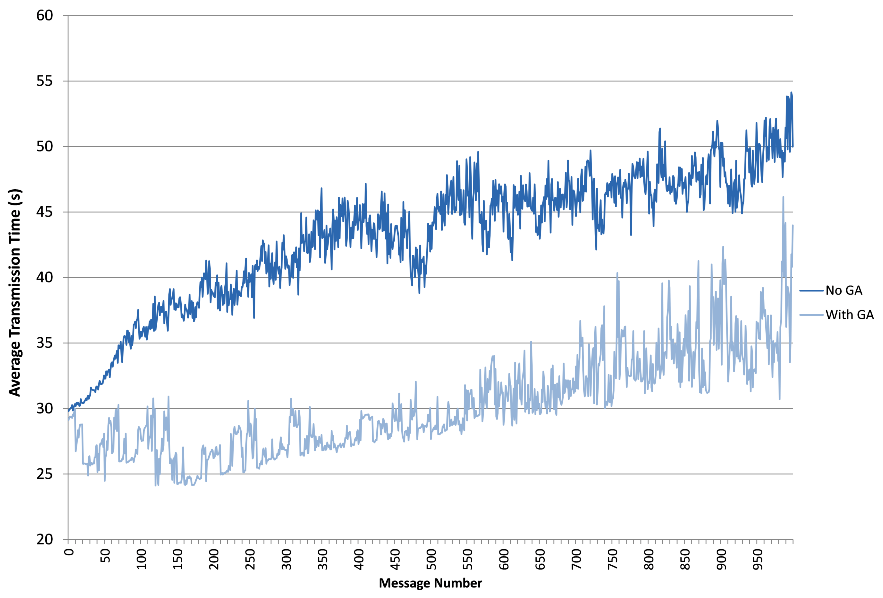

Figure 5.48: The performance of the agent with and without the GA plan modification mechanism, $C = 8$

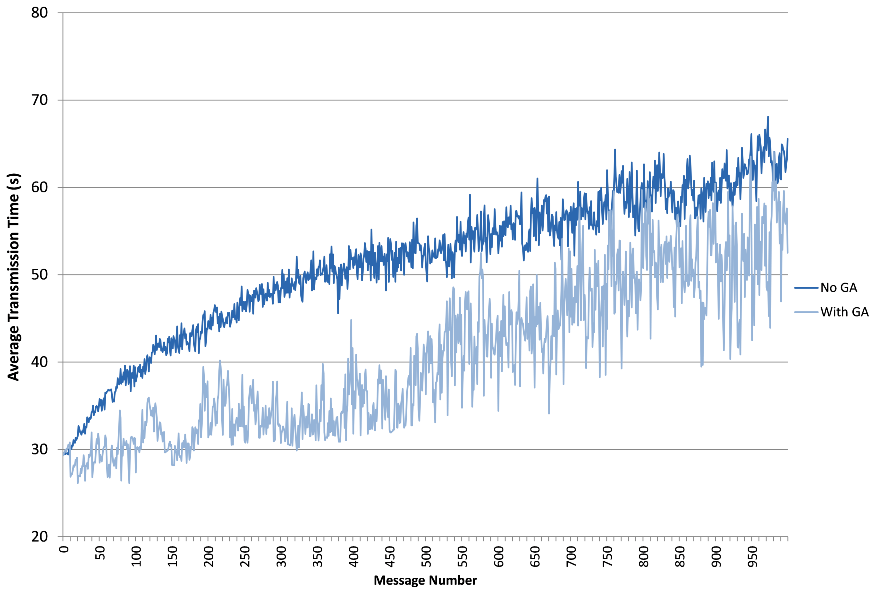

Figure 5.49: The performance of the agent with and without the GA plan modification mechanism, $C = 16$

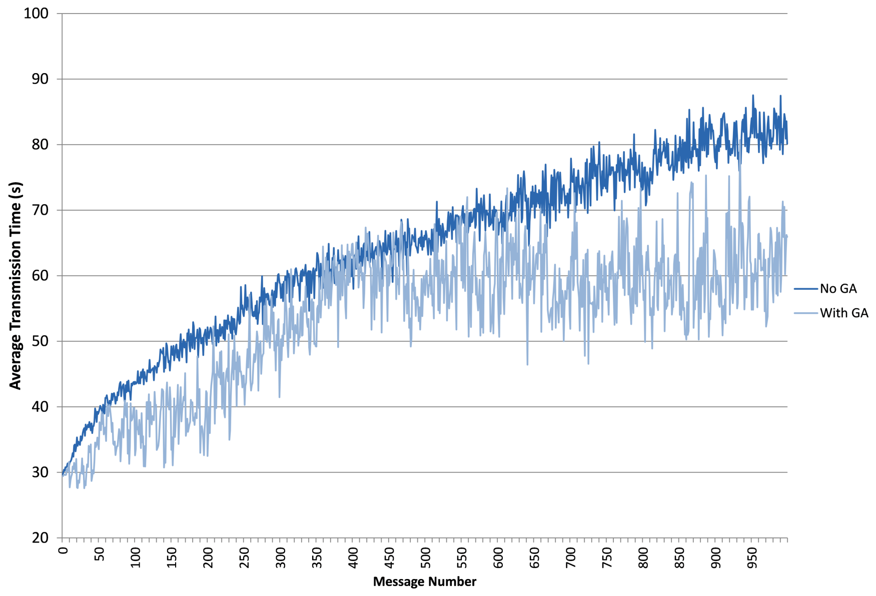

Figure 5.50: The performance of the agent with and without the GA plan modification mechanism, $C = 32$

5.11 Analysis

From the graphs included in this chapter it is clear that it is possible for a GA to act as an effective plan modification mechanism for BDI agents in a simulated network routing environment. In all cases examined the average performance of the agent with the GA plan modification component outperformed the agent without the GA plan modification component.

The main factor which became apparent over the course of these experiments as to whether a GA plan modification mechanism is likely to be efficient is that of the rate of change of the environment.

For the GA to be effective and useful, the rate of change of the environment needs to be slow in comparison to the evolution time of the GA. That is, during the process of the evolution, the environment should not change. If it does, then the evolution process is attempting to optimise a fitness value which changes as the environment changes. This is contrary to the requirement for a fitness function, discussed in Section 3.4, that the function return the same value for an individual across the course of the evolution. The environment also needs to remain relatively stable for long enough after the GA completes for the new plans that it generates to be useful.

If the time between changes to the environment is defined as t_e, then the ideal relationship between that and t_{GA} is $t_e > t_{GA}$.

Hence one can conclude that a suitable application for this manner of plan modification would be one where the environment changes, large or small, occur seldom and the GA can complete the full evolution process between those changes.

If the environment is one that experiences frequent, extreme changes, then this form of genetic plan modification is not likely to be appropriate.

The other factor that must be considered is the time taken for the GA to evolve. In this, it is much like any standard planning algorithm. Planning is a complex problem and takes time.

There were two possible approaches to running the GA in this experiment:

1. It could be run synchronously by the message delivery agent. This means that when the GA is invoked all message delivery halted until the GA completed.

2. It could run asynchronously. In this case when the GA is invoked, the evolution proceeds on another thread and the message delivery continues unhindered.

There are advantages and disadvantages to both. In the first option, which is the design that was used in this experiment, there were delays before messages could be sent if the messages were queued for delivery while the GA was executing.

In the second option, there would have been no delays, however the messages sent while the GA was in the process of evolving would have been using the older plans, plans that had resulted in message failure often enough to result in the GA being invoked in the first place. The second option was not examined in this dissertation and may be suitable for future research.

The second option would also allow for the GA to be used to predict potential future changes to the environment. Such a prediction is beyond the scope of this study, but it could involve a simulation of the environment upon which possible changes that might occur in the real environment to be implemented and the GA allowed to evolve plans. This would mean that should the predicted state occur in the real environment, plans would already exist for that state of the environment. This is also a possible avenue for future research.

Potential applications of this form of genetic plan modification must allow for processing to halt while the plans are evolved, or for processing to continue with inefficient plans while the GA evolves. If neither is possible, then an alternate form of plan modification or plan generation should be considered instead.

As has been shown, the use of GA can be effectively used to modify plans for a BDI agent in a slowly changing environment, provided the overhead of the GA can be accepted and providing that the environment changes relatively slowly relative to the time taken for a set of plans to be modified.

5.12 Conclusion

The core question in this dissertation is whether the use of a GA to evolve the plan library of a BDI agent provides any benefit for message routing in a changing environment and, if it does, under what circumstances.

In this chapter the results of the experiments run to answer that question were discussed and analysed. These results were in the form of the performance characteristics of a BDI agent operating in a slowly-changing simulated network environment.

The performance characteristics of the MA operating without a GA plan modification component were compared with the performance characteristics of the MA operating with various configurations of the GA plan modification component. The different configurations included changing the number of generations which the GA evolved for, changing the number of plans added to the plan library on each invocation of the GA and changing the criteria for invoking the GA.

The results showed an improvement in the performance of the MA in the tests which used the GA plan modification mechanism, thus answering the core question of this dissertation.

There is also scope for improvement on the design used in this study, which will be elaborated on in the next chapter.

Chapter 6

Conclusion

The aim of this study was to investigate whether Genetic Algorithms (GAs) are a feasible method for plan modification in a changing environment.

To achieve this, a simulated network environment was created, using a directed graph, and two Belief-Desire-Intention (BDI) agents were added to that environment. The task of one agent was to make changes to the environment, the job of the other agent was to transmit messages across the network as quickly as possible. The design of these agents was discussed in detail in Chapter 4.

The performance of the BDI agent, with and without the GA plan modification mechanism was discussed in Chapter 5.

In this chapter the results will be summarised and discussed in the light of the problem statement laid out in Section 1.1. Future possibilities for further research will also be discussed.

6.1 Motivation

The BDI Agent architecture uses a plan library, defined at the time the agent is created, which it uses to achieve its goals. This plan library is fixed for the life of the agent.

The fixed plan library exists so that the agent does not have to perform resource-intensive planning during its normal execution. In a static environment, a static plan library is sufficient. In a dynamic environment, the static plan library may be problematic as the plans become less suitable as the environment changes (Meneguzzi et al. 2004).

Various forms of planning have been investigated for incorporation into a BDI agent to allow for the plan library to be modified during the execution of the agent. These include state-based planners, hierarchical task networks and propositional planning (Walczak et al. 2006; Silva and Padgham 2004; De Silva and Padgham 2005; Meneguzzi et al. 2004).

GAs have not been used within BDI agents for this purpose. The motivation for this study was to determine whether GAs could be used in this manner.

6.2 Findings

The research question, posed in Section1.2, was based on the defined problem statement. The answer to that question, as supported by the results of this study will be discussed in this section.

Is a GA a feasible mechanism for plan modification in a BDI agent?

In order to answer this question, it was necessary to set up a test environment to situate the agents. The environment chosen was that of a network routing environment and the performance of the BDI agent was defined to be a function of the time taken to send the messages across the network.

In retrospect, the choice of environment was possibly not the best. While GAs have been used in network routing problems (Ahn and Ramakrishna 2002; Nagib and Ali 2010), they are not common there. In addition, the use of an environment for which there are well-known algorithms, discussed in Section 3.7, allows for confusion as the purpose of the study. A choice of robot motion, warehouse-type storage or resource optimisation based environments may have been more suitable as an experiment choice.

In the simulated network routing environment, adding the GA plan modification mechanism to a BDI agent resulted in improved performance over the default BDI agent in all experiment types.

Additional experiments were done to understand the behaviour of the GA plan modification mechanism. One unexpected result, was that increasing the number of generations which the GA evolved for did not improve the performance. Instead, when the GA was allowed to evolve for more generations, the performance remained the same or decreased compared to performance with the GA evolving for fewer generations.

This behaviour was likely due to the changing environment. With the GA running for more generations, there was a higher chance that the environment would change while it was evolving, which would hinder the convergence of the population.

It was also interesting to note that decreasing the number of generations down to only five still resulted in a performance improvement over the base agent without the GA plan modification mechanism. The performance improvement was not as great as for the experiments where the GA ran for more generations.

The experiments where the GA plan modification mechanism ran on a schedule rather than in response to failures of the existing plans showed strong performance improvements, better for much of the duration of the tests than when the GA was invoked due to failure of the plans.

Overall these results show that GAs can indeed be a feasible method of plan modification for BDI agents, at least within the environment selected for these experiments

6.3 Contribution to the Field

This study contributes to the body of knowledge in that it adds another mechanism that may be used with BDI agents when the static plan library that they include hinders the performance of the agent in a changing environment.

The study shows that, in a sufficiently slowly changing environment, a GA can run for a small number of generations and provide an improvement over the plans that it started operating on. This is not the usual way of running GAs, but has shown to be effective in this scenario and may be applicable for other scenarios using GAs.

6.4 Future Research

The most obvious avenue for future research is in investigating whether these results hold in other types of environments, such as the robot motion mentioned earlier. Such research could lead to an analysis of what kind of environments a GA plan modification mechanism can be effective in.

A second path for future research would be to adapt the GA plan modification mechanism to work in a MAS application, either with the multiple agents sharing a plan library which is the target for the GA plan modification mechanism, or for each agent to have its own, independently changing plan library.

Another potential line of research is in making the GA plan modification mechanism run continually in real time, similar to what was done in the NERO video game (Stanley et al. 2005). The later experiments in this study, specifically the small number of generations and the scheduled invocation of the GA could be taken further to a scenario where the GA plan modification mechanism runs continually, evolving for a small number of generations $G = 5$ or even $G < 5$ and adding the one or two highest fitness plans into the plan library on each execution.

A fourth, and more complex, potential path of research is that of predicting possible future changes to the environment and evolving plans in anticipation of their being needed. In short, instead of evolving plans to match how the environment has changed, the GA plan modification mechanism could use a history of environmental changes in order to predict possible future states. It could then evolve plans based on those possible future states so that if those states occurred, optimal plans would already exist.

6.5 Conclusion

The basic form of a BDI agent contains a static plan library to reduce the need for computationally expensive deliberative planning. The existence of that plan library however could hinder the performance of the agent in changing environments. Previous research has focused on integrating planning mechanisms into BDI agents.

The aim of this study was to investigate whether GAs are a feasible method for plan modification in a changing environment. The results have shown that, in a specific environment, the integration of a GA plan modification mechanism provided a performance benefit to the BDI agent over the agent without the GA component.

Appendix A

Terms and Abbreviations

Abbreviation	Definition
AFSM	Augmented Finite State Machine
AI	Artificial Intelligence
BDI	Belief-Desire-Intention
D	Destination Node
DAI	Distributed Artificial Intelligence
EA	Environment Agent
GA	Genetic Algorithm
HTN	Hierarchical Task Network
MA	Messaging Agent
MAS	Multi-Agent System
S	Source Node
SD	Standard Deviation

Table A.1: Abbreviations

Appendix B

Variables

Variable	Definition
C	Rate of change of the environment
c_0	Chance to run environment modification
c_N	Chance to a node to be modified
F	Fitness
G	Number of generations
i	Invocations of the GA
K	Number of plans in the plan library
L	Latency
m	max links per node
N	Network size
P	Population size
R	Reliability

Table B.1: Variables

Variable	Definition
t_e	Time between changes of the environment
t_f	Execution time of the fitness function
t_{GA}	Total execution time of a GA
t_{msg}	Delivery time for a message
t_{avg}	Average delivery time across all tests

Table B.2: Variables, cont

Bibliography

Ahn, C. W. and R. S. Ramakrishna (2002). "A Genetic Algorithm for Shortest Path Routing Problem and the Sizing of Populations". In: *Evolutionary Computation, IEEE Transactions on* 6.6, pp. 566–579.

Ahuactzin, J. M., E. Talbi, P. Bessiere, and E. Mazer (1991). "Using Genetic Algorithms for Robot Motion Planning". In: pp. 84–93. URL: `citeseer.ist.psu.edu/article/ahuactzin92using.html`.

Back, T., U Hammel, and H. Schwefel (1997). "Evolutionary Computation: Comments on the History and Current State". In: *Evolutionary computation, IEEE Transactions on* 1.1, pp. 3–17.

Bagley, J. D. (1967). "The Behavior of Adaptive Systems which Employ Genetic and Correlation Algorithms". PhD thesis. University of Michigan.

Beasley, D., D. R. Bull, and R. R. Martin (1993a). "An Overview of Genetic Algorithms: Part 2, Research Topics". In: *University Computing* 15.4, pp. 170–181. URL: `citeseer.ist.psu.edu/article/beasley93overview.html`.

— (1993b). "An Overview of Genetic Algorithms: Part I, Fundamentals". In: *University Computing* 15.2, pp. 58–69. URL: `citeseer.ist.psu.edu/16527.html`.

Beetz, M. and D. V. McDermott (1994). "Improving Robot Plans During Their Execution". In: *Proceedings of the 2nd International Conference* *on AI Planning Systems*. Chicago, pp. 7–12. URL: `citeseer.ist.psu.edu/beetz94improving.html`.

Bellifemine, F., A. Poggi, and G. Rimassa (2001). “JADE: a FIPA2000 Compliant Agent Development Environment”. In: *Proceedings of the Fifth International Conference on Autonomous Agents*. ACM, pp. 216–217.

Blum, A. L. and M. L. Furst (1997). “Fast Planning Through Planning Graph Analysis”. In: *Artificial intelligence* 90.1, pp. 281–300.

Blythe, J. (1999). “An Overview of Planning Under Uncertainty”. In: *Artificial Intelligence Today*, pp. 85–110.

Boella, G. and R. Damiano (2002). “A Replanning Algorithm for a Reactive Agent Architecture”. In: *Lecture notes in computer science*, pp. 183–192.

Bratman, M. (1987). *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information - Lecture Notes Series. C S L I Publications/Center for the Study of Language & Information. ISBN: 9781575861920.

Bratman, M. E. (1990). “What is Intention”. In: *Intentions in communication*, pp. 15–32.

Bratman, M. E., D. Israel, and M. Pollack (1991). “Plans and Resource-Bounded Practical Reasoning”. In: *Philosophy and AI: Essays at the Interface*. Ed. by Robert Cummins and John L. Pollock. Cambridge, Massachusetts: The MIT Press, pp. 1–22. URL: `citeseer.ist.psu.edu/bratman88plans.html`.

Brooks, R. (1986). “A Robust Layered Control System For a Mobile Robot”. In: *Robotics and Automation, IEEE Journal of* 2.1, pp. 14–23.

Brooks, R. A. (1990). “Elephants Don’t Play Chess”. In: *Robotics and Autonomous Systems* 6.1&2, pp. 3–15. URL: `citeseer.ist.psu.edu/188611.html`.

Busetta, P., R. Rönnquist, A. Hodgson, and A. Lucas (1999). "Jack Intelligent Agents: Components for Intelligent Agents in Java". In: *AgentLink News Letter* 2.January, pp. 2–5.

Calderoni, S. and P. Marcenac (1998). "Genetic Programming for Automatic Design of Self-adaptive Robots". In: *Genetic Programming.* Springer, pp. 163–177.

Calderoni, S., P. Marcenac, and R. Courdier (1998). "Genetic Encoding of Agent Behavioral Strategy". In: *Proceedings of the 1998 IEEE International Conference on Multi Agent Systems.* IEEE, pp. 403–404.

Chapman, D. (1989). "Penguins can make cake". In: *AI Magazine* 10.4, pp. 45–50. ISSN: 0738-4602.

De Jong, K. A. (1975). "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." In:

De Jong, K. A. and W. M. Spears (1991). "An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms". In: *Parallel problem solving from nature.* Springer, pp. 38–47.

De Silva, L. and L. Padgham (2005). "Planning on Demand in BDI Systems". In: *Proceedings of the International Conference on Automated Planning and Scheduling.* University of Southern California.

Dellaert, F. and R. D. Beer. "A Developmental Model for the Evolution of Complete Autonomous Agents". In: *From Animals to Animats 4. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior.*

Fikes, R. E. and N. J. Nilsson (1972). "STRIPS: a New Approach to the Application of Theorem Proving to Problem Solving". In: *Artificial Intelligence* 2.3, pp. 189–208.

Floreano, D. and C. Mattiussi (2008). *Bio-inspired Artificial Intelligence: Theories, Methods, and Technologies.* The MIT Press.

Franklin, S. and A. Graesser (1996). "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". In: *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96).* Vol. 1193. Berlin, Germany: Springer-Verlag. URL: `citeseer.ist.psu.edu/article/franklin96is.html`.

Gaspar, A. and P. Collard (1999). "From GAs to Artificial Immune Systems: Improving Adaptation in Time Dependent Optimization". In: *Proceedings of the Congress on Evolutionary Computation.* IEEE Press, pp. 1859–1866.

Ghallab, M., D. Nau, and P. Traverso (2004). *Automated Planning: Theory & Practise.* Elsevier.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

Hesser, J. and R. Männer (1991). "Towards an Optimal Mutation Probability for Genetic Algorithms". In: *Parallel Problem Solving from Nature.* Springer, pp. 23–32.

Holland, J. H. (1962). "Information Processing in Adaptive Systems". In: *Information Processing in the Nervous System: Proceedings of the International Union of Physiological Sciences.* Vol. 3, pp. 330–339.

Holland, John H (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* U Michigan Press.

Huber, M. J. (1999). "JAM: A BDI-theoretic Mobile Agent Architecture". In: *Proceedings of the Third Annual Conference on Autonomous Agents.* ACM, pp. 236–243.

Jamali, N. and S. Ren (2005). "A layered architecture for real-time distributed multi-agent systems". In: *SELMAS '05: Proceedings of the fourth international workshop on Software engineering for large-scale* *multi-agent systems*. St. Louis, Missouri: ACM Press, pp. 1–8. ISBN: 1-59593-116-3. DOI: `http://doi.acm.org/10.1145/1082960.1082978`.

Jennings, N. R., K. Sycara, and M. Wooldridge (1998). "A Roadmap of Agent Research and Development". In: *Journal of Autonomous Agents and Multi-Agent Systems* 1.1, pp. 7–38. URL: `citeseer.ist.psu.edu/jennings98roadmap.html`.

Kambhampati, S., C. A. Knoblock, and Q. Yang (1995). "Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-order Planning". In: *Artificial Intelligence* 76.1, pp. 167–238.

Keane, A (2001). "An Introduction to Evolutionary Computing in Design Search and Optimisation". In: *Theoretical aspects of evolutionary computing*. Springer, pp. 1–11.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

Luck, M., P. McBurney, and C. Preist (2003). *Agent Technology: Enabling Next Generation Computing (A Roadmap for Agent Based Computing)*. AgentLink.

Luke, S. et al. *ECJ: A Java-based Evolutionary Computation Research System*. URL: `http://www.cs.gmu.edu/%5C~%7B%7Declab/projects/ecj`.

Maes, P (1995). "Artificial Life meets Entertainment: Lifelike Autonomous Agents". In: *Communications of the ACM* 38.11, pp. 108–114. ISSN: 0001-0782. DOI: `http://doi.acm.org/10.1145/219717.219808`.

Mascardi, V., D. Demergasso, and D. Ancona (2005). "Languages for Programming BDI-style Agents: an Overview." In: *Proceedings of WOA 2005: Dagli Oggetti agli Agenti*. Pitagora Editrice, pp. 9–15.

Medhi, D. and K. Ramasamy (2010). *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann.

Meffert, K., N. Rotstan, C. Knowles, and U. Sangiorgi. "JGAP–Java Genetic Algorithms and Genetic Programming Package". In: *URL: http://jgap.sourceforge.net/*.

Meneguzzi, F. R., A. F. Zorzo, and da Costa M. M. (2004). “Propositional Planning in BDI Agents”. In: *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, pp. 58–63.

Mitchell, M and S. Forrest (1994). “Genetic Algorithms and Artificial Life”. In: *Artificial Life* 1.3, pp. 267–289. URL: `citeseer.ist.psu.edu/mitchell93genetic.html`.

Moscato, P. (1989). “On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms”. In: *Caltech concurrent computation program, C3P Report* 826, p. 1989.

Mouton, J. (2001). *How to Succeed in Your Master’s and Doctoral Studies: A South African Guide and Resource Book*. Van Schaik.

Nagib, G. and W. G. Ali (2010). “Network Routing Protocol using Genetic Algorithms”. In: *International Journal of Electrical & Computer Sciences IJECS-IJENS* 10.02.

Nebel, B. and J. Koehler (1992). *Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective*. Tech. rep. RR-92-48, p. 15. URL: `citeseer.ist.psu.edu/nebel92plan.html`.

Newell, A. and H. A Simon (1976). “Computer science as empirical inquiry: Symbols and search”. In: *Communications of the ACM* 19.3, pp. 113–126.

Newell, Allen and Herbert A Simon (1961). *GPS, a Program that Simulates Human Thought*. Defense Technical Information Center.

Nonas, E. and A. Poulovassilis (1998). “Optimisation of Active Rule Agents using a Genetic Algorithm Approach”. In: *Lecture notes in computer science*, pp. 332–341.

Olivier, M. S. (2009). *Information Technology Research: a Practical Guide for Computer Science and Informatics*. Van Schaik.

Paton, N. W. and O. Díaz (1999). “Active Database Systems”. In: *ACM Computing Surveys (CSUR)* 31.1, pp. 63–103.

Pokahr, A., L. Braubach, and W. Lamersdorf (2003). “Jadex: Implementing a BDI-Infrastructure for JADE Agents”. In: *EXP – in search of innovation* 3.3, pp. 76–85. URL: `citeseer.ist.psu.edu/braubach03jadex.html`.

Rao, A. S. (1997). “A Unified View of Plans as Recipes”. In:

Rao, A. S. and M. P. Georgeff (1995). “BDI-agents: from theory to practise”. In: *Proceedings of the First International Conference on Multiagent Systems*. San Francisco. URL: `citeseer.ist.psu.edu/rao95bdi.html`.

Riemsdijk, M. B. van, J. C. Meyer, and F. S. de Boer (2004). “Semantics of Plan Revision in Intelligent Agents”. In: *Algebraic Methodology and Software Technology*. Springer, pp. 426–442.

Russell, S. J. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall. ISBN: 9780132071482.

Russell, S. J. and E. H. Wefald (1991). *Do the Right Thing: Studies in Limited Rationality*. MIT press.

Silva, L. P. de and L. Padgham (2004). “A Comparison of BDI Based Real-time Reasoning and HTN Based Planning”. In: *Proceedings of the of Australian Joint Conference on Artificial Intelligence*. Springer, pp. 1167–1173.

Stanhope, S. and J. Daida (1998). “Optimal Mutation and Crossover Rates for a Genetic Algorithm Operating in a Dynamic Environment”. In: pp. 693–702.

Stanley, Kenneth O, Bobby D Bryant, and Risto Miikkulainen (2005). “Real-time Neuroevolution in the NERO Video Game”. In: *Evolutionary Computation, IEEE Transactions on* 9.6, pp. 653–668.

Subagdja, B., L. Sonenberg, and I. Rahwan (2009). “Intentional Learning Agent Architecture”. In: *Autonomous Agents and Multi-Agent Systems*

18.3, pp. 417–470. ISSN: 1387-2532. DOI: `http://dx.doi.org/10.1007/s10458-008-9066-5`.

Sugihara, K. and J. Smith (1997). “Genetic Algorithms for Adaptive Motion Planning of an Autonomous Mobile Robot”. In: *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation.* IEEE, pp. 138–143.

Ten Hoeve, EC, M Dastani, F Dignum, and JJ Meyer (2003). “3apl platform”. Master’s thesis. Utrech University.

Tichy, W. F. (1998). “Should Computer Scientists Experiment More?” In: *Computer* 31.5, pp. 32–40.

Ventura, S., C. Romero, A. Zafra, J. A. Delgado, and C. Hervás (2008). “JCLEC: a Java Framework for Evolutionary Computation”. In: *Soft Computing* 12.4, pp. 381–392.

Walczak, A., L. Braubach, A. Pokahr, and W. Lamersdorf (2006). “Augmenting BDI Agents with Deliberative Planning Techniques”. In: *Proceedings of the 5th International Workshop on Programming Multiagent Systems.*

Weld, D. S. (1999). “Recent Advances in AI Planning”. In: *AI magazine* 20.2, p. 93.

Wooldridge, M. J. (1995). “A Logic of BDI Agents with Procedural Knowledge”. In: *Proceedings of the 2nd ModelAge Workshop, Lisbon*, pp. 301–315.

— (2000). *Reasoning about Rational Agents.* Cambridge, Massachusetts: The MIT Press.

Wooldridge, M. J. and N. R. Jennings (1995a). “Agent Theories, Architectures, and Languages: A Survey”. In: *Workshop on Agent Theories, Architectures & Languages (ECAI’94).* Ed. by Michael J. Wooldridge and Nicholas R. Jennings. Vol. 890. Lecture Notes in Artificial Intelligence. Amsterdam, The Netherlands: Springer-Verlag, pp. 1–22. ISBN: 3-540-

58855-8. URL: `citeseer.ist.psu.edu/article/wooldridge94agent.html`.

— (1995b). "Intelligent Agents: Theory and Practise". In: *Knowledge Engineering Review* 10.2, pp. 115–152. URL: `citeseer.ist.psu.edu/article/wooldridge95intelligent.html`.