Fredrik Foss
Truls Stenrud

Foraging, Genetic Network Programming and its Hybridization with NEAT: Evolving Evolutionary Swarm Robotics

Master's thesis in Computer Science
Supervisor: Pauline Catriona Haddow
June 2021

Fredrik Foss
Truls Stenrud

Foraging, Genetic Network Programming and its Hybridization with NEAT: Evolving Evolutionary Swarm Robotics

Master's thesis in Computer Science
Supervisor: Pauline Catriona Haddow
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Swarm robotics is rising in relevance to modern society, as it allows for the solving of many problems cheaply and efficiently. Foraging is a particularly interesting domain within swarm robotics, due to it requiring an aptitude for multiple rudimentary behaviours like dispersion, area monitoring, collision avoidance and communication in order to be solved efficiently, thus making it an excellent proving ground for automatic controller design solutions, like evolutionary algorithms. Genetic network programming (GNP) is a relatively unexplored method within evolutionary computation, solving problems by evolving interconnected graph structures. Genetic network programming exhibits many traits that make it suitable for evolutionary robotics, yet is rarely used outside of simple dummy problems.

In this thesis, genetic network programming is evaluated in the foraging domain and compared to state of the art approaches. Results show that GNP is a very suitable methodology for swarm robotics, beating out other evolution-based approaches by solid margins. The only algorithms that are able to beat GNP, are human designed foraging algorithms, signifying that evolution based approaches still need improvement. In an attempt at improvement, a novel hybridization between GNP and neural networks is introduced. The novel hybrid resulted in worse performance than that of GNP, but contextualized and provided insight into the workings of GNP indirectly. On the topic of insight, this thesis introduces rudimentary analysis techniques for GNP through the use of heatmaps, leading to interesting discoveries.

Sammendrag

Svermrobotikk blir mer og mer relevant for moderne samfunn ettersom teknologien legger til rette for å løse mange problemer både billigere og mer effektivt enn andre alternativer. Sanking er et spesielt interessant domene innen svermrobotikk grunnet kravene det stiller til gode løsninger. Mer spesifikt, så må roboter kunne spre seg, overvåke områder, unngå kollisjoner og kommunisere med hverandre, oppførsler som er grunnleggende for å løse mer avanserte problemer. Hvis en generell metode klarer å utvikle god oppførsel for sankedomenet, så klarer den mest sannsynlig å gjøre det for andre domener også. Genetisk nettverksprogrammering (GNP) er en relativt lite utforsket metode innen evolusjonær komputasjon, hvor man evolverer en sammenhengende multigraf for å løse et problem. GNP har mange trekk som gjør det gunstig for evolusjonær robotikk, men blir allikevel sjeldent brukt til noe mer avansert enn rutenettbaserte problemer.

Denne masteroppgaven tar GNP til sankedomenet og sammenlikner metoden med både det beste innen sanking (algoritme designet av mennesker), og det mest populære innen evolusjonær robotikk generelt. Resultatene viser at GNP ikke bare egner seg for svermrobotikk, men gruser de andre evolusjonsbaserte algoritmene med god margin. På tross av grundig grusing, blir GNP fortsatt slått av det beste innen sanking generelt, som tyder på at evolusjonsbaserte metoder trenger flere forbedringer for å kunne konkurrere med menneskedesignede oppførsler. Som et forsøk på en slik forbedring, kombineres GNP med nevrale nett for å lage en hybridalgoritme. Morsomt nok gjorde tillegg av nevrale nett at GNP mistet en god del hjerneceller, men hybriden slo fortsatt rent nevrale metoder med god margin. På tross av hybridens ferdighetsnedgradering, var den brukbar for mer detaljert analyse angående hvilke hensyn som bør tas når man jobber med GNP. Apropos analyse, så introduseres noen grunnleggende metoder for å analysere GNP. Disse er alle basert på varmekart, og leder til noen kule oppdagelser.

Preface

This thesis is the resulting work of a master thesis in artificial intelligence at the Norwegian University of Science and Technology in Trondheim. The thesis has been worked on in the period of 23.01.2020 - 14.06.2020. We would like to thank our supervisor Pauline Catriona Haddow for guidance and detailed feedback on the project, and NTNU for providing access to the IDUN computing cluster. [Själander et al., 2019]. We would also like to thank Kristian for coming up with the GIN acronym, and Marius for bringing super smash bros to school, allowing us to ignore Koskom's tyrannical corona rules.

Fredrik Foss   Truls Stenrud
Trondheim, June 14, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter briefly explains the motivation behind this thesis, what the goals of the research are, how research was performed, and an overview of the contributions this thesis provides.

1.1 Background and Motivation

Genetic Network Programming (GNP) was proposed in [Katagiri et al., 2000] as an alternative to genetic programming, better suited towards dynamic domains and especially artificial agents. Since then, the methodology has remained relatively unexplored, with most papers (for example [Li et al., 2018; Wang et al., 2015; Sendari et al., 2011]) neglecting comparisons against popular modern techniques like NEAT. In addition, most GNP research is done in domains that are inherently advantageous to GNP, but generalize poorly to real world problems ([Li et al., 2018; Wang et al., 2015; Sendari et al., 2011]). Research being done in such domains isn't inherently a bad thing, but evaluating GNP in less suitable domains will provide insight into the potential and generality of the technique, as well as allow direct comparisons between GNP and other evolutionary techniques on more even grounds. This paper aims to evaluate and compare GNP to state of the art methods in the foraging domain, which is continuous and highly relevant to real world applications. In addition, a hybrid algorithm combining NEAT and GNP will be introduced and investigated, in an attempt to improve GNP.

1.2 Goals and Research Questions

There are two main, separate goals for this thesis, as well as multiple research questions which will be investigated in order to reach those goals.

Goal 1 To investigate the potential of GNP in evolutionary robotics, by comparing it to state of the art methods within the foraging domain.

Goal 2 Investigate a potential improvement to GNP, by hybridizing it with NEAT.

Research question 1 Is GNP competitive in the foraging domain when compared to the current state of the art?

Research question 2 How do resource clustering methods affect GNP compared to the state of the art?

Research question 3 How does GNP scale with swarm size compared to the state of the art?

Research question 4 Which nodes are important for GNP in the foraging domain?

Research question 5 How does hybridizing GNP with NEAT affect performance?

Research question 6 How does node usage in NEAT-enhanced GNP differ from that of normal GNP?

1.3 Research Method

Over the course of the project, there were three disctict phases of research with respective research focuses. First, for preliminary research, internet search engines were used to scout conferances and journals in robotics and evolutionary robotics for interesting topics and papers. If a paper was deemed interesting, key citations would be further investigated for more inspiration. Notable conferences used include *Robot Intelligence Technology and Applications (RITA)*, *International Conference on Evolutionary Robotics (ICER)*, *International Journal of Swarm Intelligence (IJSI)*, *International Conference on Swarm Intelligence (ICSI)*, and *IEEE International Conference on Robotics and Automation (ICRA)*. In the first stage of research, paper quality was not a concern as the main goal was to find interesting research avenues. The main takeaways from the first stage of research were that NEAT was omnipresent in evolutionary robotics, swarm robotics was cool, foraging was a popular domain in swarm robotics, and that Genetic Network Programming (GNP) looked interesting.

The second phase of research focused on exploring foraging and GNP further. This was accomplished by first finding key early papers to understand the basics of each area, then switching focus to more recent contributions. The recent contributions provided potential research directions, hints to the current state of the art as well as citation trails leading to key papers. Quality assessment in phase two was based on several factors including the complexity of the chosen domain, number of comparison bases, the competitiveness of comparison bases, potential lacking core features, whether or not effort had been put into parameter optimization, number of -, length of -, and breadth of experiments as well as consistency with previously reported results. Information from the second phase of research was used to establish the state of the art, as well as build a large space of ideas regarding architecture possibilities, inclusions and research directions.

The second phase of research revealed that most researchers use different platforms and configurations for foraging simulations, which was a problem when comparing algorithms and attempting to determine the state of the art. In addition, it was revealed that GNP had mostly been used in very simple domains, and never in foraging, rendering the potential of the method unclear. Initially, a large project with many quirks was planned. However, since GNP had never been used in the foraging domain, it was deemed more scientifically appropriate to explore its performance in a less feature-rich version of foraging, as complicated experiments are hard to interpret without a base. In line with the above, the project was restructured to accomplish two goals: First, provide comparisons between GNP and both popular and state of the art methods in a basic foraging environment. Second, explore novel contributions as a separate goal, allowing novel contributions to GNP without disturbing this papers role in determining the potential of GNP. With a clear project plan, the final phase of research consisted of finding software and papers with publicly available code, in order to allow implementation of the project described above within the given time frame. In addition, impromptu searches were carried out on IEEE

Explore or Google scholar whenever a question arose (for example when selecting operators or checking theoretical hunches).

1.4 Contributions

This thesis contributes to the growing list of evolution-based foraging literature, and empirically proves that GNP can do well in more complex domains than those in which GNP is most often used, outperforming the state of the art in evolutionary swarm robotics in many scenarios. A novel hybridization of GNP and NEAT, GIN, is also introduced in this thesis. GINs performance is somewhere in between that of GNP and NEAT, resulting in lower performance than that of ordinary GNP in the foraging domain. GINs middle ground performance indicate that GIN embodies a mixture of strengths and weaknesses from both GNP and NEAT, but which traits and whether or not the mixture is suitable for any particular domain is unknown. This thesis also acts as one of the first attempts at analyzing GNP behaviour, setting a basis for further analysis and resulting in some interesting information.

In addition to the direct results above, the source code used in this thesis is also publicly available at https://github.com/TrulsStenrud/GNP-with-roborob3 in order to aid research in the foraging domain, as simulators with several foraging algorithms readily available in them are rare to find, and are often more than a decade old, or built for old simulators, resulting in slow runtimes. The software package contributed here will certainly have its problems as well, but they will be different from that of other publicly available software suites.

1.5 Thesis Structure

This thesis is split into 6 major parts: The introduction, which this section concludes. Background theory, where necessary rudimentary knowledge and terms are introduced. State of the art, where best practices and the forefront of research within certain subfields are presented. Architecture/model, where novel contributions are presented and explained in detail. Chapter 5 describes the experiment setups, containing experimental data as well as discussions surrounding the results of said data. In addition, chapter 5 brings analysis of the more novel components of this thesis. Finally, chapter 6 concludes the paper, reflecting back at the original goals of the paper, highlighting strengths and weaknesses of the work.
Disclaimer: the background theory and state of the art chapters are heavily modified versions of text previously written for a research assignment in connection to the TDT4501 course. This is a core part of how masters theses are written at NTNU, but this disclaimer is still necessary.

Chapter 2

Background Theory

This chapter contains explanations of rudimentary concepts and methods, as well as terminology needed to understand the rest of the thesis. Specifically, evolutionary algorithms, NeuroEvolution of Augmented Topologies (NEAT), Genetic Network Programming (GNP), evolutionary robotics, swarm robotics and foraging will be covered. This chapter describes the basics of these subjects, whereas relevant discoveries and recent contributions within the subjects are covered in the state of the art chapter.

2.1 Evolutionary Algorithms

Evolutionary algorithms have been around since the introduction of evolutionary search in [Turing, 1950]. Over the years the term has grown to cover a large set of techniques, mostly used for optimization problems, and taking inspiration from the biological concept of evolution. Evolutionary algorithms adapt *reproduction*, *mutation* and *selection* on a population in order to evolve highly fit solutions to a given problem. Evolutionary algorithms are good general optimization algorithms because they have no comprehension of the inner workings of proposed solutions, simply striving to optimize a number of values whose purpose is unknown. Evolutionary algorithms typically consist of a single *population* filled with *individuals.* Each individual has a single *genome* (also known as the *genotype* of the individual), which is a collection of all values which the evolutionary algorithm needs to optimize. While all such values are part of the genome, genomes are sometimes partitioned into subgroups known as *chromosomes.* As for the values themselves, they are usually referred to as *genes.*

Parent selection is based on a fitness function, which evaluates the quality of each genome. This is done by parsing the genome into a *phenotype*, which is the form of the actual solution. For example, a genome could consist of three numbers, and the phenotype could be a polynomial function based on those three numbers. After a phenotype is created from the genome, it is then evaluated by the fitness function, and a score is assigned to the genome. After two parents have been selected, they mix their genomes in a process known as *crossover* to create children, which additionally have a chance to mutate, causing small or large changes in their genome depending on the mutation operator. Since parent selection is based on the fitness function, highly fit individuals will have more children than less fit individuals, causing their genes to spread throughout the population. By applying the process of evolution over multiple iterations (generations), highly fit solutions probabilistically emerge. Figure 2.1 shows a simple evolutionary algorithm loop.

The shape of fitness functions is very important for evolutionary algorithms to be able to solve problems efficiently. To understand why, think of the previously mentioned example, where the genome is three numbers, and the phenotype is a polynomial function. For this example, the goal of the evolutionary algorithm is to approximate a function. If the fitness function is the square distance between the polynomial and the ground truth, the evolutionary algorithm is able to make gradual improvements as individuals climb the fitness curve. If the fitness function is a boolean function describing whether the approximation is good enough or not, there's no gradient to climb, and the evolutionary algorithm will essentially be doing random search. Many domains feature fitness functions that have vast areas of equally useless solutions, in which case the evolutionary algorithm must first a suboptimal, but useful solution, then start climbing in the direction of increasing fitness. Escaping the useless part of such fitness functions is known as the *bootstrap* problem, and it will be important later on in this thesis.

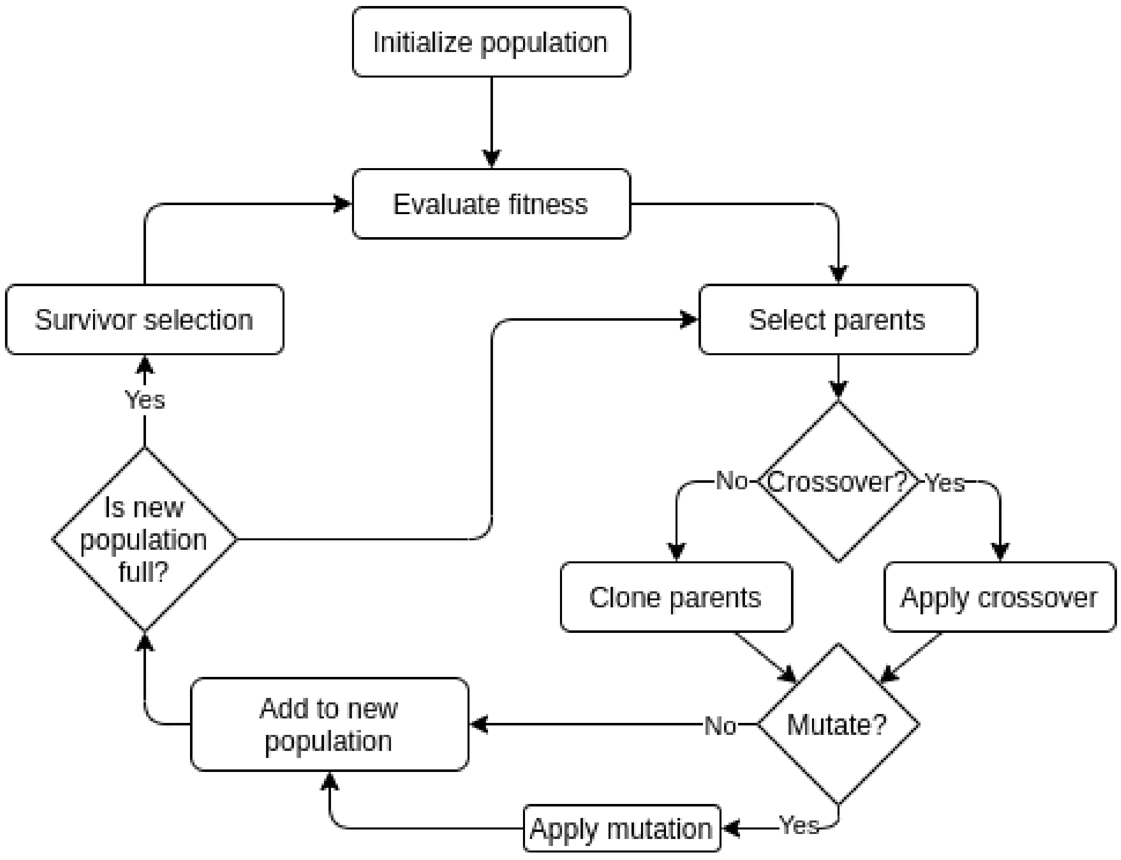


Figure 2.1: Flowchart of a basic evolutionary algorithm

There are multiple operators for all parts of the evolutionary algorithm loop, but these are particularly relevant for this paper:

- **Tournament selection:** Tournament selection is a parent selection method that can be described as follows: T_s individuals are randomly chosen from the population, then the most fit individual among those is chosen as a parent. T_s is typically kept low, as low values of T_s provide lower selection pressure, and therefore more exploration power. Higher values of T_s however, are more geared towards exploitation. Tournament selection is repeated twice in order to select two parents.

- **Single point crossover:** A random point $p \in [0, n]$ is chosen, where n is the genome length. Child c_1 will inherit genes in index $[0, p]$ from parent 1, and genes in index $[p+1, n]$ from parent 2. Child c_2 will inherit genes in index $[0, p]$ from parent 2, and the rest from parent 1.

- **Age based replacement:** Age based replacement is a form of survivor selection where the previous population dies and is replaced by the new one. It is used by all algorithms featured in experiments in this paper.

- **Elitism:** Elitism alters survivor selection by copying the best individuals in the old population into the new population. This prevents the best solutions from dying due to unlucky parent selection or crossover. All algorithms featured in this thesis also use elitism.

2.1.1 Parameter Tuning/Control

Evolutionary algorithms depend on many parameters like population size, tournament selection size, crossover chance, mutation chance, number of elitist individuals and more depending on the algorithm. The performance of EAs is very dependent on these parameters, so configuring them is an important optimization task in itself, one which can either be done manually, by secondary optimization algorithms, or dynamically by internal mechanisms while the algorithm is running. The subfield for optimizing parameters is split into two main categories: parameter control and parameter tuning. Parameter tuning revolves around setting parameters as optimally as possible before running an algorithm, whereas parameter control modifies parameters while an algorithm is running.

One of the most common approaches to parameter tuning is manually testing configurations, but that takes a long time, and can get tedious when optimizing many algorithms. An alternative to manual parameter tuning is to use an automatic parameter tuning algorithm. One of the earliest approaches to automatic parameter tuning can be found in [MERCER and SAMPSON, 1978], where an external evolutionary algorithm was used to optimize another evolutionary algorithm. The primary problem with such meta-EAs is that EAs require a lot of fitness evaluations, and since every fitness evaluation for the meta-EA is a full run of another EA, such a method for parameter tuning can take weeks to finish. In an effort to reduce the required number of fitness evaluations, [Maron and Moore, 1997] introduced the *racing* algorithm. This work was later developed into the F-race algorithm [Birattari et al., 2002], then the iterated F-race algorithm [Balaprakash et al., 2007], and the most recent iteration, the *irace* algorithm [López-Ibáñez et al., 2016]. All of the racing algorithms are inspired by horse races, and work by first creating random parameter configurations, then optimizing each of them one step at a time, eliminating statistically significantly worse configurations as they go.

As for parameter control, there are three main categories: *deterministic*, *adaptive* and *self-adaptive* parameter control. Deterministic approaches are based on human intuition and follow predictable patterns. An example of deterministic parameter control can be, for example, increasing mutation rate as generations go by, in order to prevent the population from prematurely converging in one part of the fitness landscape. Adaptive approaches gather statistics from the population, and use that to control parameters. As an example, information entropy can be used to manage mutation rate to manage the same scenario as in the deterministic example. Self-adaptive approaches include parameters in the genomes of each EA individual. Sticking with mutation rate, each individual could simply use the chance that is stored in its own genome to figure out if it should mutate or not.

2.1.2 Genetic Programming

Genetic programming (GP) is one of many evolutionary paradigms invented after the introduction of the basic bitstring GA in [Holland John, 1975]. Similar to most EA's, GP aims at being a method for solving problems through a high level description with minimal amounts of domain-specific knowledge. Like most EAs, it solves problems through the use of selection, crossover and mutation over several iterations. The defining difference between genetic programming and other evolutionary paradigms lies in the phenotype. Instead of solutions being encoded as bitstrings as in the basic GA, they are encoded in tree structures that, when parsed, become the candidate solution.

An example of a GP tree is illustrated in figure 2.2. The internal nodes of the tree, marked in blue, are called function nodes. The nodes are connected through links, illustrated as arrows pointing from parent to child. Each function takes in their child nodes as arguments. The leaf nodes, marked in green, are called terminal nodes. The terminal nodes can be fixed constants, variables or zero argument functions. For example, the bottom left blue node in figure 2.2 is a multiplication symbol, and it has two children, 5 and x. The two children are the arguments that goes into the multiplication function, and the result of the multiplication serves as an argument to the parent of the multiplication node.

For any given GP algorithm, the set of possible node types must be determined based on the current domain. Which variables and which functions should be included are all problem dependent and is required to be set before using GP. A common problem in genetic programming is that of *bloat*. When GP solutions struggle to improve their fitness further, children will likely have the same fitness as their parents, or a worse fitness. Crossover operators combine elements from both parents, which most of the time will lower the childrens fitness. Because of this, GP solutions are implicitly rewarded for adding useless / noncoding nodes, as it decreases the likelihood of disturbing the useful / active part of their children's genomes [Terrio and Heywood, 2002]. Bloat not only adds computational complexity, but actively combats the crossover operator, and therefore lessens the likelihood of finding better solutions. To combat bloat, it is normal to introduce a variable which restricts the maximum depth of the genome tree. While that approach works, it also introduces another parameter that must be tuned specifically for any given domain.

2.2 NEAT

Neuroevolution of Augmented Topologies (NEAT) is a technique used to evolve neural networks using direct encoding, speciation and complexification. It was proposed in [Stanley and Miikkulainen, 2002] as a way to evolve both the topology and the weights of a network; as opposed to manually designing the network, then evolving the weights, as was common before. At the time, there were three key features that separated this approach from other techniques in neuroevolution: Historical markings, or *innovation numbers*, for tracking genes; speciation for protecting innovation; and complexification through incremental growth from a minimal sized network.

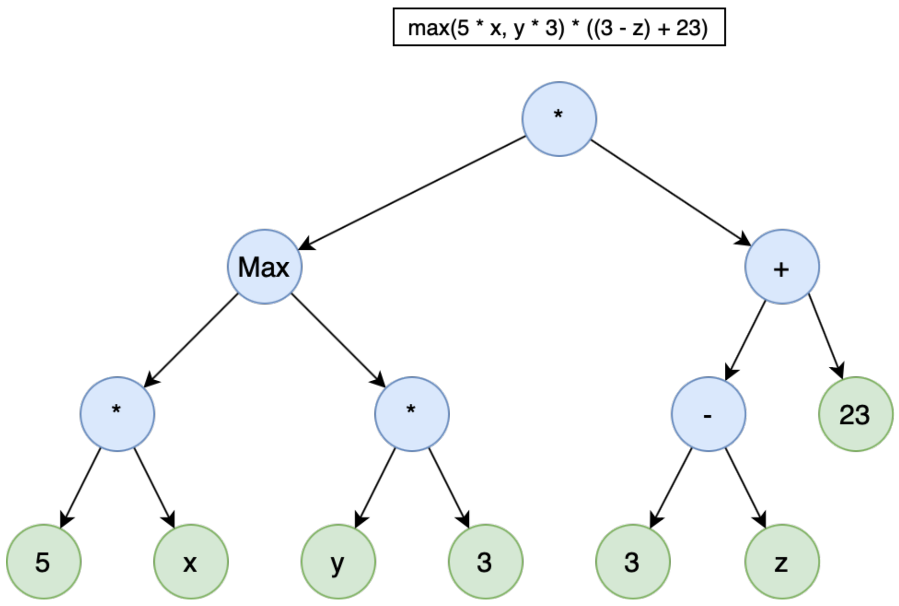


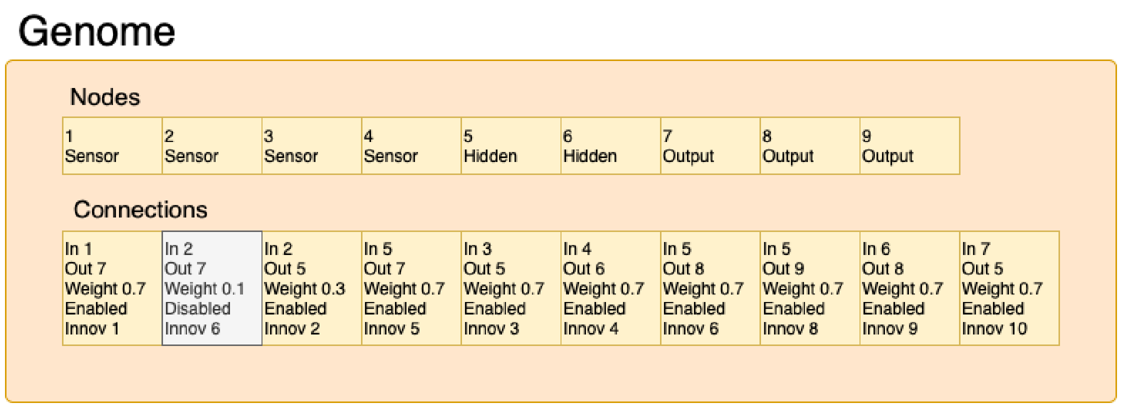
Figure 2.2: An example of a GP tree, terminal nodes in green and function nodes in blue. A more common way to express the same program shown above the tree.

Through the use of these key features NEAT is capable of creating networks of minimal complexity while maintaining a good balance between exploration and exploitation.

To achieve minimal complexity networks, NEAT starts with a population of the smallest networks possible, gradually introducing more complexity through the use of crossover and mutation. By gradually increasing complexity, NEAT is able to find neural networks whose sizes are adapted to the problem, minimizing the risk of generating bloated, overly complex solutions.

NEAT uses direct encoding, which means that every node, connection and weight is represented directly in the genome. As shown in figure 2.3 NEAT uses two chromosomes: one node-chromosome and one connection chromosome. The node chromosome contains a list of nodes, with their id and node type (sensor, hidden or output). In the connection chromosome, each element represents a connection between two nodes. Each connection specifies an input node, output node, a weight and an innovation number (innov #). The innovation number is assigned when the connection is created based on the state of the global mutation counter, and is used as a historical marker for both crossover and speciation which will be explained in more detail later in this section.The genome illustrated in figure 2.3 will result in the network illustrated in figure 2.4. As shown in the genome there are 4 sensor-nodes, also called input nodes, 2 hidden nodes and 3 output nodes. All the connections are present, except the one between node 2 and 7 (innov 6), which is disabled.

When new topological structures are introduced, they are often poorly optimized, and will therefore suffer from lowered fitness from that of other individuals in the population. Speciation is a mechanism that helps prevent these novel individuals from dropping out of the gene pool prematurely, giving them time to optimize their novel structures. This makes sure that individuals only compete with other individuals within the same niche, i.e. similar individuals. To recognize



Genome

Nodes

1 Sensor	2 Sensor	3 Sensor	4 Sensor	5 Hidden	6 Hidden	7 Output	8 Output	9 Output

Connections

In 1 Out 7 Weight 0.7 Enabled Innov 1	In 2 Out 7 Weight 0.1 Disabled Innov 6	In 2 Out 5 Weight 0.3 Enabled Innov 2	In 5 Out 7 Weight 0.7 Enabled Innov 5	In 3 Out 5 Weight 0.7 Enabled Innov 3	In 4 Out 6 Weight 0.7 Enabled Innov 4	In 5 Out 8 Weight 0.7 Enabled Innov 6	In 5 Out 9 Weight 0.7 Enabled Innov 8	In 6 Out 8 Weight 0.7 Enabled Innov 9	In 7 Out 5 Weight 0.7 Enabled Innov 10

Figure 2.3: An example of a NEAT genome with the two chromosomes. Nodes with their id's and connections with their respective values.

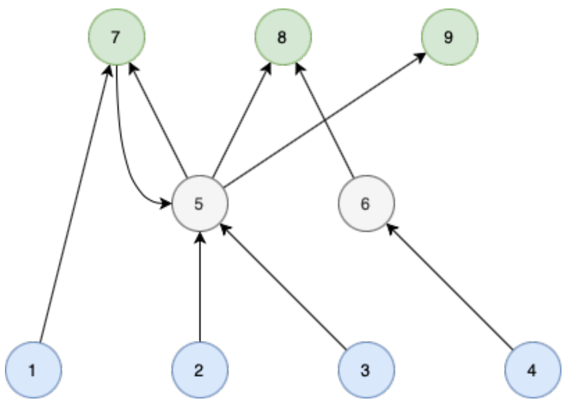


Figure 2.4: A neural net involving 4 input nodes, 2 hidden nodes and 3 output nodes. This is the phenotype of the genome shown in figure 2.3.

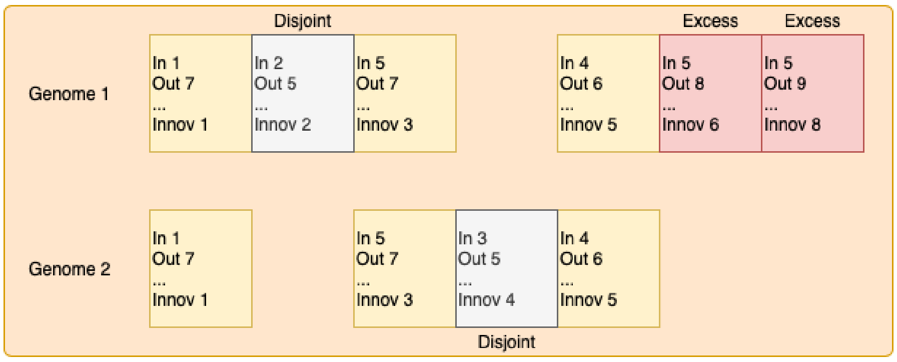


Figure 2.5: A comparison between two genomes, highlighting the disjoint and excess genes in both.

which individuals are similar, a comparison metric between genomes must be in place. For this comparison, the innovation numbers have a key role. This is illustrated in figure 2.5, where genes with similar innovation number (innov #) are lined up. Those genes originate from the same mutation, and are therefore called *matching* genes. Non-matching genes are either *disjoint* (marked in gray) or *excess* (marked in red), depending on whether the gene's innovation number is within the range of the other parent's innovation numbers. For instance, in figure 2.5, in genome 1, the gene with innovation number 2 (innov 2) is marked as disjoint because the other genome, genome 2, has genes with innovation numbers below and above 2. On the other hand, the gene with innovation number 6 and 8 is marked as excess, because the other genome only has innovation numbers up to 5. The total distance between two candidate solutions is found by summing the number of disjoint and excess genes found within them, as well as the average connection weight difference of matching genes. In the case of figure 2.5 it would be 2 disjoint genes and 2 excess genes, summing up to 4.

As mentioned, the innovation number is also used when performing crossover. During this operation matching genes are passed on to the child at random while disjoint and excess genes are inherited from the more fit parent. In the case of a tie, disjoint and excess genes are also inherited randomly.

2.3 Genetic Network Programming

Genetic Network Programming (GNP) was proposed in [Katagiri et al., 2000] as a variant of genetic programming (GP) designed for intelligent agents in dynamic domains. The core idea was to overcome many of the problems GP faces in such problems by evolving an interconnected graph of nodes, instead of a tree. A graph structure removes the need for intelligent agents to reset to a root node every time it makes a decision, allowing agents to make contextual decisions based on previous decisions in the program flow. Since GNP phenotypes never return to their root node, program flow tends to revolve around certain parts of their networks, reusing key nodes and subgraphs. This behaviour allows GNP solvers to evolve state machines if the problem calls for it, whereas GP would need preprogrammed state sensors to evolve similar behaviour.

Genetic programming will bloat itself up to its maximum tree depth if given enough time, because adding useless nodes is implicitly rewarded (see 2.1.2 for details). The maximum tree depth parameter is therefore a very sensitive parameter, as too low a value will prevent the global optimum from being found, and too high a value will introduce bloat problems [Terrio and Heywood, 2002]. GNP solvers work with a static number of judgement and processing nodes, and will therefore never introduce bloat. Using more nodes doesn't provide an inherent advantage for GNP, so solutions tend to only use the nodes they need, while leaving the rest unused / disconnected [Hirasawa et al., 2001]. This difference makes GNP more robust to its own parameters, as higher node quantities increase the search space, but not necessarily the solution complexity. GNP consists of three distinct node types, clearly echoing the node types of GP. A detailed description of the three node types is provided below:

- **The start node:** There is exactly one start node in every GNP-network. It consists of a single output edge, and cannot be the destination of outgoing edges from other nodes. At program start, the agent starts by leaving this node, never to return to it. This node is analogous to the root node in GP.
- **Judgement nodes:** Judgement nodes evaluate sensor values and/or internal information in order to determine which node should be processed next. These nodes have multiple outgoing edges, and choose one based on a judgement function, which can be arbitrarily complex. Judgement nodes can lead to processing nodes, or other judgement nodes for complex decision making. This node type is analogous to internal / tree nodes in GP.
- **Processing nodes:** Processing nodes are actions the agent can perform, such as setting/modifying the values of actuators, or altering the internal state of the agent. These nodes only have a single output edge. This node type is analogous to terminal / leaf nodes in GP.

Figure 2.6 shows an example GNP network with 10 judgement nodes and 7 processing nodes. Most of the graph edges go to neighboring nodes in the illustration, but that is purely for visual clarity as the current state of the art in GNP does not involve any neighborhood relations or positional coordinates. Note how the processing node (circle) in the top right will never be reached, and how the judgement node (diamond) on the bottom left has a recurrent connection. Also note how multiple judgement nodes can be chained together to enable complex decision making and how multiple processing nodes can be chained together to form routines.

Since GNP never returns to the start node, it needs a mechanism to halt processing and let time pass in simulated environments. This is achieved using time costs. Simply put, the GNP controller allows for a certain number of time points to be spent before time passes, with 5 and 10 being common limits in the literature [Wang et al., 2015; Roger and Jordan, 2015; Li et al., 2014; Katagiri et al., 2000]. Commonly, the start node costs 0, judgement nodes cost 1 and processing nodes cost 5 time points. After all time points have been spent, the controller halts processing and allows time to pass. A side effect of the time cost system is predictable computational complexity, as large GNP controllers will use the same amount of time points as small GNP controllers.

One of GNP's advantages is on demand data usage. As judgement nodes only need data when they're visited, a robot can elect not to process data when it isn't needed, reducing computational complexity. Another boon with on demand data usage is that it allows dynamically sized inputs. For example, a neural approach would have to preemptively decide how many inputs should be dedicated to information regarding nearby robots, whereas GNP allows sending a full list of nearby robots to a judgement node, then letting the node do anything it wants with that information.

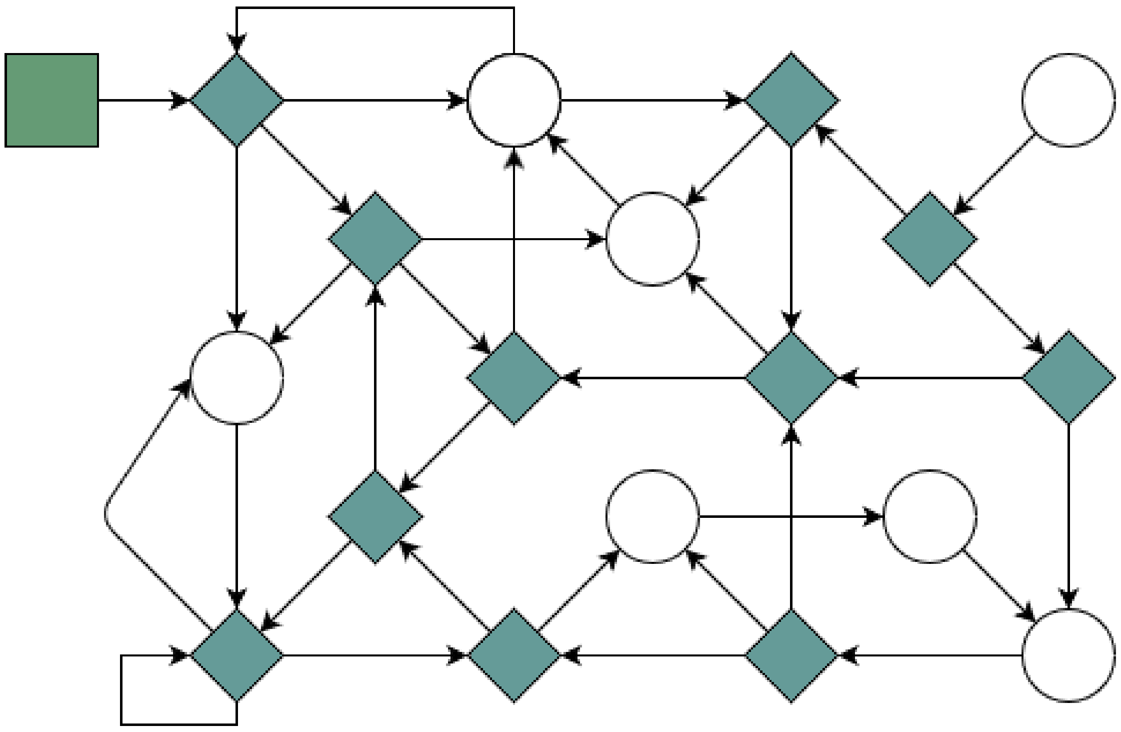


Figure 2.6: An example of a GNP network. The green square represents the start node, cyan diamonds represent judgement nodes and white circles represent processing nodes.

Compared to neural approaches

Compared to neural approaches like NEAT, GNP exhibits both advantageous and disadvantageous traits. One such trait is that GNP is inherently discrete and nonlinear in its decision making, whereas neural approaches make decisions based on continuous outputs. Many domains feature continuous inputs and outputs, and one of the problems with discrete decision making in such domains is that it relies on arbitrary separation points, and if those are poorly set, the algorithm might struggle with finding good solutions.

Neural networks come with infinite phenotype search spaces, and separate neural networks with different weights and topologies can encode the exact same behaviour despite phenotypic dissimilarities. GNP networks do not have infinite phenotype search spaces, and therefore have an innate advantage over neural networks when dealing with the bootstrap problem, as separate GNP networks necessarily produce different behaviours.

GNP's contextual decision making is both a strength and a weakness. It allows GNP to evolve state machines, and provides a sense of time to the controller, but it prevents making decisions based on the full state of all sensors available to the agent. If GNP needs to make decisions based on all of its available sensors, it is necessary to evolve a large and complex decision tree, which is hard. Another option is of course to have a specific node that evaluates the full state of the agent, but that introduces the problem of optimizing said node.

2.4 Evolutionary Robotics

Since the introduction of evolutionary search in [Turing, 1950] and the invention of the GA in [Holland, 1962], evolutionary search has spread into many domains and research directions. Since the main inspiration behind evolutionary search lies in the evolution of creatures, it is only natural that evolutionary search would find its way into the robotics community. Evolutionary robotics as a science is focused on evolving robotic creatures, both in mind and body, transfering the adaptability of biological creatures into artificial agents and autonomous robots. Given a problem description, the perfect ER algorithm should be able to provide a suitable controller for solving the problem, as well as a body if it is wanted by the user. Unfortunately, the perfect ER algorithm currently doesn't exist, but the quest continues.

A robot consists of two major parts: a controller and a body, both of which can be the subject of evolution. The most common approach is to have a fixed body for the robot and focus on evolving a controller for it, but some researchers have looked into co-evolving the morphology and the controller [Sims, 1995], as seen in nature. This approach is less common though, so most researchers either design the robot themselves, or use previously designed robots, and evolve just the controller. In the field of evolving controllers for the robots the field has been split in two, one direction focusing on cognitive science and biology, and the other using classical EA techniques for evolving controllers. In regards to the latter, the goal is to be able to evolve controllers, for robots, being able to solve a given task, just by the task-description in the form of a fitness function. There should be no need to manually program the robots actions.

As with all variations of evolutionary algorithms, ER algorithms need to evaluate the performance of candidate solutions. There are two main approaches for evaluation, both providing their own advantages and disadvantages. The most directly applicable technique is to evaluate candidate solutions on real robots. This is very slow, requires space and equipment and is potentially costly, but guarantees that the solution works in the real world. A more common method among researchers is to use simulators. Simulators are a lot faster, only requiring computational resources to evaluate performance. They are also a lot less costly in the case of a masochistic candidate solution, as no physical parts can be damaged. The main problem with simulated fitness evaluation is known as the *reality gap*, a problem caused by the various simplifications simulators do, which makes solutions that are great in simulators less than great in the real world. Common problems in real-world robotics like imperfect sensors, imprecise actuators and varying terrain properties are often not present in simulators. In addition, most simulators feature very simple physics, potentially causing candidate solutions to ignore advanced cause and effect relationships which might be present in the real world. Several countermeasures to the mentioned problems have been proposed [Robots, 1996] to varying degrees of success, but the problem still persists [Silva et al., 2016].

2.5 Swarm Robotics

One of the earliest mentions of a robotic swarm was in the works of Gerardo Beni who studied cellular robotic systems in [Beni, 1988]. The author proposed a new type of robotic system, consisting of a homogeneous group of robots capable of cooperating to solve their given task. The system is inspired by social insects in nature, where large groups of relatively simple individuals are capable of performing complex and difficult tasks when cooperating. The concept of swarm intelligence can be seen in ants, that alone cant do much, but together can form bridges to navigate difficult terrain (see figure 2.7). Another example can be found in the termite, which is mostly useless alone, but can build great structures when part of a swarm (see figure 2.8). Swarm robotics as a field is focused on replicating the success of biological swarms by having complex, intelligent behaviour emerge from interactions between the environment and a swarm of relatively simple agents [Şahin, 2005].

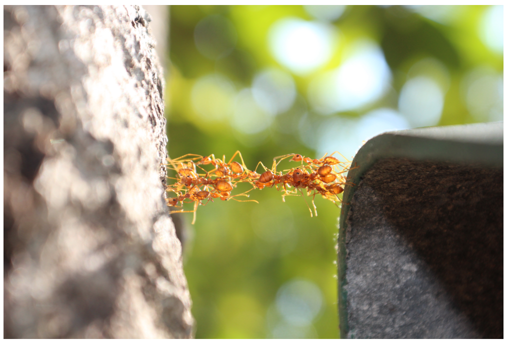


Figure 2.7: A group of ants forming a bridge across a gap too long for any single ant to cross. Image by [Igor Chuxlancev, 2015]

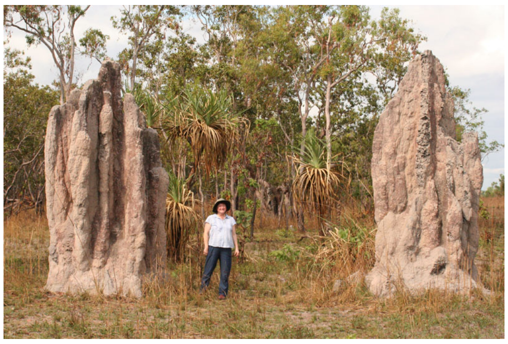


Figure 2.8: Termites can build incredible structures even though every individual termite follows fairly simple rules. Image by [Ray Norris, 2005]

In addition to being able to achieve complex group behaviour from simple individual behaviour, biological swarms have certain features that are beneficial for robotic systems: robustness and scalability. Swarm systems have no central control systems coordinating efforts, as the intelligent behaviour arises from interactions between individuals and the environment. This lack of central control means that one, or multiple, individuals can malfunction or otherwise be removed from the swarm without hindering the swarms ability to do its task. As long as the number of individuals in the swarm is above a task-dependent threshold, the swarm will still function for its intended purpose. The redundant nature of individuals in swarms provides robustness to failure in the way described above, but also comes with another advantage: scalability. By adding more workers, the performance of a swarm generally increases. As such, the size of a swarm can easily be scaled to match the requirements of a task. While swarm solutions are scalable in general, the proportional benefit of adding another member to the swarm is both algorithm- and task specific. While great successes have been reported, as in [Hiraga et al., 2019], where the proportional benefit of additional members surpassed the increase in swarm size; most swarm algorithms see diminishing returns as swarm size increases [Ericksen et al., 2018].

Manually programming good swarm behaviour is a difficult task, as it's easy to imagine what would constitute good behaviour for a single robot working alone; but harder to imagine how individual behaviours affects an entire swarm of robots. One of the main advantages with evolutionary approaches in swarm robotics is that you evade that problem, as humans ideally only need to declare what they want the swarm to accomplish by changing the fitness function, then letting evolution take care of designing individual behaviours that lead to the wanted overarching behaviour.

2.6 Foraging

Ants are very interesting insects, in part due to their relatively advanced teamwork skills. Ants are experts at dispersing and searching their local area for food, and will guide each other to food sources when found by leaving pheromones on the ground. When near or inside their nest, ants also display impressive queuing skills, showcasing great logistics. Ants also exhibit rudimentary memory with behaviour known as *site fidelity*, remembering food locations and returning there until there's no more food there [Beverly et al., 2009]. Another advanced behaviour found in certain ant species is task partitioning [Ratnieks and Anderson, 1999], splitting themselves into roles without having to rely on biological markers like some swarm societies need to (eg: bees). To summarize: ants show proficiency in many rudimentary skills, including collision avoidance, dispersion, area monitoring, communication and cooperative sharing of resources (nest pathways). These skills are interesting to swarm robotics researchers because of their elementary nature. If a robot can become proficient at all of those skills, it can likely accomplish more advanced tasks as well, as most tasks can be boiled down to a combination of the aforementioned rudimentary skills.

Foraging is a problem in swarm robotics that at its most basic level, is about replicating ant behaviour for finding food. It is one of the most studied domains in swarm robotics, mostly due to it's high complexity and relevance to real world problems [Alfeo et al., 2019]. The most basic version of the problem is called *Central Place Foraging (CPF)* and contains the following components:

- A bounded area to forage in.
- Resources to collect.
- A nest to deliver the collected resources to.
- A swarm of agents to perform the foraging task.

The goal is to design controllers that enable agents to bring as many objects to the nest as possible in the allotted time. This requires proficiency in multiple basic skills like collision avoidance, dispersion, area monitoring, communication and cooperative sharing of resources (the nest). The foraging problem can be viewed as a state-based problem, where each state has its own goals and challenges [Winfield, 2009; Alfeo et al., 2019]. The states and their unique problems are more closely described below:

- Search: While searching for resources, agents should disperse in order to scout as much of the area around a nest as possible. If resources randomly appear over time, continuous monitoring of the local area is wanted. If objects are clustered, behaviour that enables returning to previous locations is beneficial [Isaacs et al., 2020].
- Grab: Once sensors pick up a nearby resource, the agent needs to move towards the resource in order to pick it up.
- Return: When carrying a resource, agents should make an effort to return it to the closest nest. If there are many agents attempting to return resources at the same time, queuing problems can arise, making collision avoidance a particularly useful skill.

Finding good behaviour for all of the mentioned states is key to solving the foraging problem.

Chapter 3

State of the Art

In this chapter, the state of the art within certain relevant topics will be elaborated on. Specifically, the state of the art within foraging and GNP will be explained, as well as the state of the art for novelty search within swarm intelligence. The *multiple place foraging algorithm (MPFA)* will also be explained in this section, as it is believed to represent the state of the art within foraging.

3.1 Foraging

Many variations of the foraging problem exist that make it more general, more realistic or more challenging. [Chapman et al., 1989] extended CPF into *Multiple Place Foraging (MPF)* by allowing the use of multiple nests. Multiple nests have been shown to improve scalability because it decreases overall travel time of agents as well as reducing queuing problems that might arise when many robots want to deposit into a nest at the same time [Lu et al., 2016a]. [Lu et al., 2018] Introduced moving nests, which resulted in greater performance at the cost of being less applicable to real world domains. [Alfeo et al., 2019] made the MPF problem more application oriented by imposing a city grid on the harvesting area and forcing the robots to periodically charge their batteries at nests. [Hiraga et al., 2019] Introduced poison to the CPF domain without letting individual robots separate poison from food, forcing swarms to rely on group cognition to separate food from poison. They also made individuals too weak to independently move food/poison, forcing the emergence of teamwork.

Regarding the distribution of collectable objects, many variants have been proposed in the literature. The oldest and most common variant is random distribution [Hecker and Moses, 2015], but both homogeneous and heterogeneous cluster sizes have been used in the literature, as well as some oddities like infinite collection sites [Hoff et al., 2013]. According to [Hecker and Moses, 2015], heterogeneous clusters require more complex communication, memory and environmental sensing than other variants.

According to [Ericksen et al., 2018], the most common approaches for solving foraging problems involve manually designing state-based swarm intelligence / stigmergy-based behaviours. While such approaches do seem to work, they are often highly domain specific. These human-designed controllers often emulate swarm intelligence approaches, with ant colony optimization, particle swarm optimization, and bee colony optimization taking the lead in popularity [Efremov and Kholod, 2020]. Emergent intelligence approaches are less common, but have shown promising results [Ericksen et al., 2018; Hiraga et al., 2019]. Noteworthy human-designed approaches include the central place foraging algorithm (*CPFA*) [Hecker and Moses, 2013] and its extension to multiple places, *MPFA* [Lu et al., 2016b]. What makes these algorithms noteworthy isn't just their performance, but their well established role as common benchmark algorithms. Both algorithms are state-based and employ evolutionary computation to optimize state transition probabilities according to the environments needs, and base themselves on ant behaviour. In [Hecker and Moses, 2015] the CPFA is tested in various real world environments on iAnt robots, which are known for their poor sensors and poor navigation capabilities [Hecker et al., 2013], demonstrating the robustness of the CPFA. The foraging problem is important for swarm robotics because it requires proficiency in many basic behaviours that are important for solving real problems, not simply because the foraging problem is interesting [Winfield, 2009]. With this in mind, the CPFA and MPFA are useful as comparison benchmarks when developing general methods, but for the goal of generalized problem solving, their usefulness stops there.

A major problem in evolutionary robotics is the lack of standardized benchmark problems. There are many simulators to choose from, and researchers tend to stick to their choice once they've learned a particular simulator. Consequently, the widespread adoption of a set of benchmark problems requires that the problems are readily available for many different simulators. Simulators all have their own quirks and assumptions regarding physics, as well as featuring vastly different robot properties (sensor and actuator parameters), making the development of a widespread set of benchmark problems a deceptively hard task. Even when researchers use the same robot setups, the available sensory information in the environment is usually different. Unfortunately, these problems make it hard to determine the state of the art in foraging.

The closest resemblance to a foraging benchmark can be found in [Ericksen et al., 2018], who reproduced DDSA and CPFA experiments in the ARGoS simulator, then compared them against a foraging-optimized version of NEAT that they themselves designed. Their approach didn't change any of NEAT's mechanics, but simply provided capabilities for placing and detecting pheromones. All three algorithms were compared on various permutations of swarm sizes and resource distribution styles to measure their relative performance. Their experiments show that none of the three algorithms perform strictly better than another, but they found large differences in performance trends as they changed the swarm size for the different algorithms, suggesting that swarm size scaling is heavily dependent on algorithm, and therefore worthy of investigating when contributing new information to the field of foraging.

There is no state of the art approach to foraging due to the aforementioned problems, but there is a general consensus on which sensors robots need to be effective. The majority of foraging papers provide robots with a nest sensor of some sort, whether it's by inputting an angle to a neural network [Hiraga et al., 2019], providing the location of a physical lamp [Hecker and Moses, 2015], or by providing the directions to the closest nest at every foraging site [Alfeo et al., 2019]. Neural approaches generally include distance to the closest robot as an input value, with most including the angle as well in order to further aid agents in performing dispersion effectively [Ericksen et al., 2018]. [Hiraga et al., 2019] went even further and included both distance and angle to the 2 closest robots as their foraging variant was fully reliant on teamwork, instead of teamwork simply being a positive contributor as is the case with most foraging variants.

For communication, evaporating virtual pheromones are by far the most common choice [Alfeo et al., 2019; Ericksen et al., 2018; Hecker and Moses, 2013]. Neural approaches like [Ericksen et al., 2018] usually provide an input for local pheromone strength and an output neuron that determines whether to deposit pheromones at the current location. CPFA and MPFA use pheromones to stochastically recruit other ants to promising sites, and in [Hecker and Moses, 2015], the effectiveness of this approach was empirically tested. By comparing simulations evolved with pheromones to solutions evolved without pheromones, they found the pheromone-enabled solutions to be substantially better in all cases, with the percentagewise advantage scaling negatively with swarm size.

Site fidelity is a phenomena observed with real ants [Beverly et al., 2009] that is commonly imported into manually designed foraging algorithms [Fricke et al., 2016; Hecker and Moses, 2013; Lu et al., 2016b]. Like the name suggests, it means that agents return to their previous harvest site after depositing a resource, in case there are more resources to collect at the original location. [Isaacs et al., 2020] compares the relative improvements of several techniques, and finds site fidelity to be a significant positive contributor. They also find recruitment (agents deciding to follow other ants pheromone trails) to give slight performance improvements, and finds delivery lanes (to avoid congestion at the nest) to provide no benefit at all.

There are many differences between approaches in swarm robotics and the workings of real insect swarms in nature, and one of the most obvious one is the homogeneity of robotics approaches. In nature, insect swarms tend to divide labour, allowing individuals to specialize in certain parts of a task [Ratnieks and Anderson, 1999]. [Ferrante et al., 2015] uses grammatical evolution to combine three semi-primitive behaviours in order to evolve task division like in nature. Their evolved solution has agents probabilistically join one of three groups, leading to higher fitness. Their results indicate that task partitioning should be investigated more thoroughly in foraging, as it is bio-inspired, can lead to better performance, and requires more teamwork than standard foraging approaches.

3.2 Multiple Place Foraging Algorithm

The multiple place foraging algorithm (*MPFA*) was introduced in [Lu et al., 2016b] as an ant-inspired, state-based approach to solving the multiple place foraging problem. It was purposely designed to be as similar to the central place foraging algorithm (*CPFA*) as possible, as to enable direct comparisons between single- and multi-nest foraging [Lu et al., 2016b]. Since their introduction, both the CPFA and the MPFA have been featured in many foraging papers, usually performing evenly or better than their competition [Ericksen et al., 2018; Lu et al., 2018; Fricke et al., 2016]. Due to this tendency, the CPFA and the MPFA may be regarded as state of the art in the foraging domain. A state diagram showing the overarching behaviour of the CPFA can be found in figure 3.1. The only differences between CPFA and MPFA are that MPFA-agents return to their closest nest instead of a central one, and their pheromones are nest-specific instead of being globally available.

The MPFA has 5 states, and every time step consists of the MPFA-agent processing its current state, then moving towards its target, which is an internally stored coordinate. Before explaining the states, it is necessary to describe what is meant by informed and uninformed search in the context of the MPFA. When an agent randomly selects a position along the edge of arena to move towards, that is uninformed search. When an agent decides to follow a pheromone trail or use site fidelity to return to a previously found resource, that is informed search. The 5 states as well as their behaviours are described below:

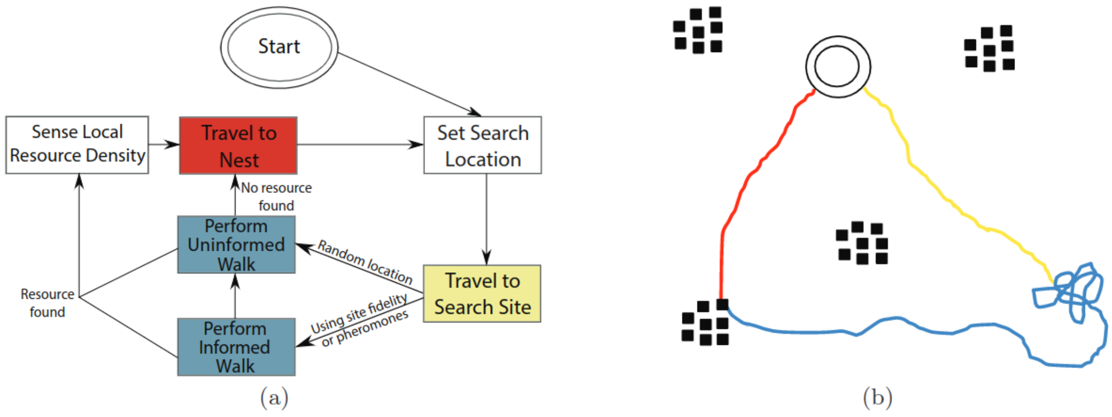


Figure 3.1: State diagram for the CPFA. [Hecker and Moses, 2013]. (a) Shows a simple flowchart of CPFA behaviour. (b) Shows a hypothetical agent displaying the behaviour in the flowchart by traveling from its nest (white ring) to a random area (yellow line), then switching to uninformed search (blue line) until it finds a resource (black square) and returns to the nest with the resource (red line).

Inactive	The starting state. When in this state, the agent will target a random spot along the edge of the arena (uninformed search), then transition to the departing state.
Departing	The agent will keep moving towards its target in a straight line. If informed, it will move to the ”searching” state when it reaches its target. If uninformed, it will simply select a new uninformed search target when it reaches its previous target. In addition, when performing uninformed search the agent has a chance of moving to the ”searching” state every 5 seconds.
Searching	The agent will move sporadically in hopes of finding resources in the local area. If a resource is found, the agent picks it up and attempts to detect additional resources in the area, then switches to the ”returning” state. The number of resource found in this step is referred to as the *resource density* of the area. Every 5 seconds of searching, the agent has a chance of giving up and moving to the ”returning” state without finding any resources.
Returning	The agent returns to the closest nest, at which point it has a chance to deposit a pheromone containing the coordinates where resources were found. Afterwards, the agent will either use site fidelity to return to the previous site, follow a pheromone trail left at the nest by another agent, or perform uninformed search. The pheromone deposition chance and the site fidelity probability are affected by the resource density of the area where the resource was found.
Shutdown	When all resources have been found, return to the nest.

The core behaviour of an MPFA-agent is unchanging, but various probabilities and parameters are optimized for the environment by using an evolutionary algorithm [Hecker and Moses, 2013]. The configurable parameters are as follows:

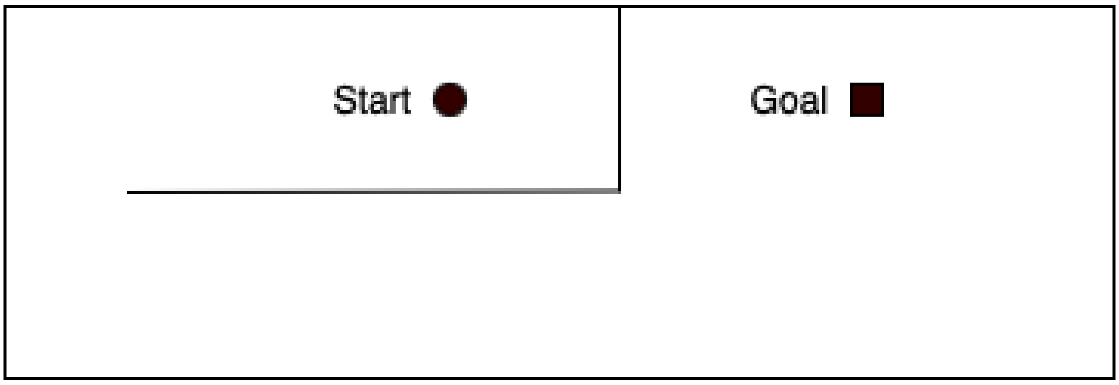


Figure 3.2: A simple, but deceptive maze.

Probability of switching to search	Chance of transitioning from the "departing" state to the "searching" state when performing uninformed search.
Probability of returning to nest	Chance of transitioning from "searching" to "returning" without finding any resource.
Uninformed search variation	In the searching state, the sporadic movement is implemented using turn angles gotten from a gaussian distribution with σ = uninformed search variation.
Rate of informed search decay	When performing informed search in the searching state, the turning angles decay over time, causing the agent to move away from the local area.
Rate of laying pheromones	When combined with resource density in a poisson distribution, provides the chance of placing a pheromone when depositing a resource at the nest.
Rate of site fidelity	When combined with resource density in a poisson distribution, gives the chance of the agent returning to his previous location after depositing a resource.
Rate of pheromone decay	The rate at which pheromones decay.

3.3 Novelty Search

One of EAs strengths lies in its ability to move search towards highly fit locations in the fitness landscape, but this comes with a downside, as so-called *deceptive* search spaces might trap the search process in a local optimum. In order to successfully navigate deceptive fitness landscapes, it is necessary to move away from the direction of increasing fitness and spend time exploring multiple niches. To better illustrate this idea, consider the case where an agent needs to reach the end of a maze, using its distance to the goal as a fitness function. In the case of figure 3.2, a greedy algorithm would have the agent move directly towards the goal and reach a dead end, signifying a local optima in the fitness landscape. In order to reach the goal, the agent needs to first reduce its fitness by moving away from the goal, then increase it by moving towards the goal after having escaped the dead end. In the literature, several methods have been proposed to deal with deceptive landscapes, *novelty search* being one of them.

With novelty search, the core idea is to abandon the fitness function altogether, and instead perform parent selection based on solution novelty. Since its introduction in [Lehman and Stanley, 2008], novelty search has been studied in many settings to various results. In the original paper as well as many that followed, domain specific novelty measures were used, and achieved good results. In the maze example in figure 3.2 this metric could, for example, be the end location of the agent, so that the algorithm would create different solutions ending up in different parts of the maze. For these kinds of problems, novelty search generally finds the goal faster than fitness based search [Lehman and Stanley, 2011].

The aforementioned papers show the power of novelty search when comparison metrics are carefully designed to fit the domain, but [Mouret and Doncieux, 2012] managed to take novelty search one step further by introducing several generalized behavioural similarity measures. [Gomes and Christensen, 2013] built on the work of [Mouret and Doncieux, 2012], and introduced two behavioural similarity measures specifically designed for swarm robotics, one being a special case of the other. Instead of evaluating domain specific things like position on a 2D grid, they aggregate differences between input and output vectors for all agents over multiple time steps, thereby generalizing and making novelty search more accessible to engineers, albeit at the cost of a minimal performance loss when compared to task specific novelty measures. Both using and calculating the input-output difference is very computer intensive, so time windows are used to reduce the amount of comparisons needed. Instead of calculating differences for every time step in the simulation, one can aggregate, for example, 50 time steps into one input-output pair, thus reducing the computational load of novelty comparisons. The generalized similarity measure described above is known as *sampled average state*. The other similarity measure they introduced is a special case of sampled average state, where the time windows are infinitely sized. This method, dubbed *combined state count*, showed lower median and peak performance, as well as having a larger variance in performance.

It's tempting to compare novelty search to random search, but novelty search is adept at avoiding re-exploring previously found areas of the fitness landscape, a claim that can't be made for random search. In addition, many practical domains feature a surprisingly limited space of behaviours, making novelty search a fast heuristic when combined with behaviour-based comparison metrics [Mouret and Doncieux, 2012; Gomes and Christensen, 2013]. Novelty search will exhaust all the simple failing behaviours, then move on to more complex and hopefully successful behaviours. Novelty search also has the advantage of rewarding stepping stones, which is necessary in many complex domains. Imagine an environment with several balls, a basket and an agent. The fitness function could be the number of balls the agent puts in a basket, but getting to the point where the agent even picks up the ball might be difficult for fitness based evolution, as all solutions up until a ball is successfully deposited will have zero fitness. For novelty search however, picking up the ball would be very novel compared to simply wandering around, and would therefore be rewarded.

Novelty search comes with its own set of downsides, requiring an archive of previously explored solutions in order to search productively, which increases space complexity. A phenomena that reduces the severity of this problem is the fact that individuals generally have similar levels of complexity across the population, so archives can forget primitive solutions without risking a return to simplicity. While novelty search is efficient at bypassing deceptive parts of fitness landscapes and enabling the exploration of multiple evolutionary niches, it is less efficient at finding optimas within said niches. A promising method for solving the aforementioned problem is by combining novelty search with fitness based search in a multi-objective fashion [Gomes and

Christensen, 2013].

3.4 Genetic Network Programming

As mentioned in section 2.3, GNP was introduced in [Katagiri et al., 2000] to improve on the shortcomings of genetic programming. Since then, many improvements have been made to various aspects of the methodology. A shared problem between GP and GNP lies in their dependence on correctly set separation points when evaluating real numbers. In an attempt to solve this, [Mabu, 2007] added reinforcement learning to judgement nodes. They achieved drastically superior results with this method, but at the cost of a 30-130% higher time complexity during training, depending on the problem.

In real world control systems, fuzzy logic is commonly used to split continuous values into discrete categories [Feng, 2006]. The long-standing success and popularity of fuzzy control systems partly inspired [Sendari et al., 2011], in which Mabu's approach was further improved by incorporating fuzzy logic into judgement nodes, then using reinforcement learning to change the parameters of the fuzzification function. They evaluated their approach on a wall following challenge, and found that the fuzzy approach had similar performance to the non-fuzzy reinforcement learning approach on the training arenas, but outperformed the non-fuzzy approach by an average of 13% on unseen arenas while at the same time enjoying a lower standard deviation; suggesting that the fuzzy approach is more robust than the non-fuzzy approach. This makes sense considering the popularity and success of fuzzy control in the real world [Feng, 2006], and is a step forward in closing the reality gap problem that permeates evolutionary robotics.

Towards generality

The holy grail of evolutionary robotics is generalized autonomous learning, but according to the no free lunch theorem, operators that contribute positively for one problem may have negative effects for other problems [Wolpert and Macready, 1997]. By viewing general problem solving as one large search space comprising all individual problems search spaces, it's clear that the no free lunch theorem not only applies to entire search spaces, but to parts of them as well. Ensemble algorithms are growing in popularity as a way to deal with the no free lunch problem, generally producing better results than simple evolutionary algorithms [Del Ser et al., 2019]. The basic premise of ensemble algorithms is to have a large portfolio of genetic operators and search methods, so that solutions are improved when a suitable operator for that stage of search is chosen, while elitism prevents solution decay caused by poor operators for that stage of search. The concept of ensembles is independent from GNP because it affects the optimizer without affecting the phenotype, but recent developments have brought the strengths of ensembles to the GNP phenotype as well.

Without explicitly comparing their approach to ensemble algorithms, [Wang et al., 2015] improved the generality of GNP by adding pre-trained domain specific processing nodes to the node repertoire, increasing the amount of node types available to GNP. Their approach resulted in lower training time and better performance, but required domain knowledge. Their training time did not include time spent training the domain specific processing nodes, rendering the overall time complexity of their solution unknown. While the concept of including pre-built nodes is useful for engineering, their approach didn't do much for the goal of generalized autonomous learning.

Inspired by Wang, [Roger and Jordan, 2015] created a fully modular version of GNP called *Cascading GNP*, or *CGNP* for short. In CGNP, the strength of ensembles is fully employed towards generality by allowing any artificial intelligence algorithm to be used as a judgement or processing node, including entire subnetworks of GNP. There's little reason to believe subnetworks of GNP can provide ensemble-like advantages, as graphs with subgraphs is simply another way to present a larger graph. There is, however, potential in using entirely different AI-algorithms as nodes, as this would supply GNP with both the strengths and weaknesses of that algorithm, whenever GNP decided on using the node. Theoretically, the CGNP architecture has no depth limit on subnetworks, but the authors limited it to 2 layers in their implementation. To account for the now highly varying time complexities of nodes, simulated time costs were based on node complexity, rather than static costs. If necessary, particularly complex nodes would have their execution stretched out over multiple time steps. The authors behind CGNP included subnetworks of GNP as well as fuzzy reinforcement learning nodes as possible node types. Interestingly, CGNP resembles Brooks subsumption architecture [Brooks, 1991], as they both involve subsystems handling primitive tasks, with layers of higher level controllers adding intelligence. The primary difference being that CGNP evolves all layers simultaneously whilst the subsumption architecture is a bottom-up approach.

CGNP improved on the generality of GNP, but at the cost of introducing an infinite genotype search space, potentially increasing search time. Despite the infinite search space of genotypes, the space of behaviours is restricted by the problem at hand as well as the sensors and actuators available to the robot. NEAT also has an infinite search space of phenotypes, and promising results have been reported by defining behaviour-based distance measures between NEAT controllers, then using multiobjective novelty search with those distance measures [Mouret and Doncieux, 2012; Gomes and Christensen, 2013]. GNP doesn't struggle with bootstrapping as much as NEAT does, but there's still reason to believe that multiobjective novelty search could enhance the search process.

Evolutionary optimization of GNP

Most GNP literature has used conventional evolutionary algorithms, but positive results have been reported by hybridizing evolutionary algorithms with secondary optimization methods. This is in line with current research on memetic algorithms, theoretically and empirically proving that hybrid approaches improve performance on deceptive multimodal problems [Nguyen and Sudholt, 2020]. [Wang et al., 2015] applied hill-climbing search to the best three individuals every fifth generation, gaining significant performance boosts. They did not provide runtimes, but the hill-climbing version only needed 10 generations to surpass the average performance of the non-hill-climbing version after 60 generations. [Yu et al., 2007, 2008] successfully hybridized GNP with ant colony optimization by storing the transition frequencies of edges during fitness evaluation, converting the frequencies to pheromone values based on fitness, then applying ant colony optimization every 10th generation. Their approach was very successful, providing a 10-23% improved final fitness over that of vanilla GNP. Finally, [Li et al., 2015] swapped out the evolutionary part of GNP entirely, opting for bee colony optimization instead and gaining a slight performance boost compared to ordinary evolutionary GNP.

Hybridization seems to contribute positively, but it's important not to neglect the evolutionary part of memetic algorithms, as it's the primary drive for global exploration [Nguyen and Sudholt, 2020]. The evolutionary side of GNP is relatively unexplored, with most research focusing on the phenotype itself. While few in number, the works focusing on the evolutionary aspects of GNP have contributed some very useful findings. Notably, efforts to directly or indirectly prevent fast ruination of subgraphs have all had positive effects; highlighting an important consideration when designing GNP optimizers.

[Katagiri et al., 2003] compares numerous crossover methods and empirically finds biased uniform crossover to be the best. In biased uniform crossover, every node in a child has an α chance of being inherited from parent p_1 and a $1 - \alpha$ chance of being inherited from parent p_2. When nodes are inherited, they bring their outgoing edges with them. The authors of the paper also note that α should be kept low for optimal performance.

The main advantage GNP has over other evolutionary phenotypes is its ability to reuse important nodes / subgraphs in order to make contextual decisions. Noticing that standard evolutionary operators fail to consider GNP's distinct advantages, [Li et al., 2014] developed novel crossover and mutation operators dubbed *adaptive crossover* and *adaptive mutation* respectively. They theorized that by shifting evolutionary efforts away from heavily reused parts of the network and focusing it on less used parts of the network, evolution would spend less time ruining good subgraphs and spend more time improving behaviour. In their approach, a novel *evolutionary pressure* factor is calculated for each node based on a reusability quantification function. This factor is then used to dictate each nodes chance of moving to the next generation unaltered. Their method provides better initial performance, but they stopped their experiments before reaching a fitness plateau in all of their experiments, rendering the long term value of their approach unclear. The evolutionary pressure calculations also introduce five manually optimized parameters, substantially increasing the time cost of optimizing parameters for a particular scenario.

In nature, all creatures have deactivated, or non-coding DNA, referred to as introns. Introns may encode traits that were useful during earlier stages of evolution, but have since been deactivated. Atavism is a phenomena where deactivated traits are reactivated in the genome, leading to the reemergence of certain traits in an individual, such as teeth in chickens. Atavism is rare in nature, but it would be even rarer, if not impossible, if non-coding DNA was continuously scrambled during reproduction. In [Li et al., 2018], genetic operators are improved upon by importing the biological concept of atavism from natural evolution. They do this by introducing the *simplified crossover* and *simplified mutation*, where modification of unused nodes is completely disallowed, preventing deterioration of deactivated subnetworks in genomes. Their approach encourages mixtures of new and old behaviours to emerge in the population, which proves to be an advantage. Their approach also acts as a stochastic search space reducer, since genetic operators are forced to exclusively focus on active parts of the genome. Their approach is simpler, removes the need for extra parameters, and more biologically plausible than the previously discussed approach, traits most would agree are welcomed.

Network sizing

One of GNP's primary advantages over GP is its natural aversion to bloat, owing to a static number of nodes per candidate solution. The classic approach to network sizing in GNP is to initialize the network with a certain number of copies of each node, but that approach does not consider GNP's tendency to reuse important nodes. By supplying a static amount of copies of each node, solutions are either restricted in the amount of important nodes they have available; or have too many niche nodes available, stifling search. In an effort to solve this problem, [Li et al., 2011] introduced variable size genetic network programming, or *GNPvs* for short. In

GNPvs, both parents employ a binomial distribution to select a number of their nodes. Two children are produced by matching the selected part of one parent with the non-selected part of the other parent. Since the amount of nodes inherited from each parent is stochastic, offspring end up with stochastic genome sizes as well. While successfully increasing the adaptability of GNP, their approach increased bloat and did not explicitly deal with the core problem - GNP needing more copies of certain nodes and less of others.

[Li et al., 2013] improved on GNPvs by replacing binomial selection with a selection scheme based on node usage. Heavily used subnetworks in elite individuals are saved in a temporary database, then offspring from other parents will have their least used subnetwork swapped with a random elite subnetwork. They claimed to be inspired by the biological phenomena of gene duplication, but an elitist gene pool that inserts itself into other parents kids is not exactly bio-plausible.

Applications

As discussed in chapter 2.3, GNP's unique features renders it a particularly useful controller for dynamic, discrete domains where state based behaviour is appropriate. Research on GNP has mostly been confined to domains that are advantageous for GNP; often abstracting away difficulties that arise in the real world, such as unreliable sensors, unreliable actuators or non-deterministic environments. GNP has been applied to mazes and tile worlds [Li et al., 2018], perceptual aliasing problems [Murata et al., 2005], robot soccer [Wang et al., 2015], elevator group control [Yu et al., 2007, 2008], stock market prediction [Chen et al., 2009], associated rule mining [Shimada et al., 2005], traffic control [Zhou et al., 2008], wall avoidance problems [Sendari et al., 2011] and controlling a Sony AIBO ERS-7M2 robot dog [Murata and Okada, 2006], among other domains.

Most of the mentioned problems involve binary sensors that perfectly convey information about the world they operate in. Research in such domains certainly has its merit, but does not reflect the problems robotics currently faces in the modern world. The most common test-bed for GNP seems to be the tile world, a simplified version of foraging that is rare to see in general literature on evolutionary robotics, and for a good reason. In the tile world, all objects exist on a discrete-valued 2-dimensional grid and agents perceive the world through a combination of distances, angles and perfectly reliable binary-valued sensors. These simplifications reduce the real-world applicability of research, making tile world a worse research domain than foraging in the context of evolutionary robotics. Foraging is one of the most popular research domains in swarm robotics [Alfeo et al., 2019], but as far as the authors know, GNP has not been applied to foraging yet.

Chapter 4

Architecture/Model

Every method has their strengths and weaknesses. As mentioned in 2.3, GNP's strengths include being able to naturally evolve state machines, contextual decision making, on demand data usage, and a natural advantage over neural approaches in dealing with the bootstrap problem. GNP however, has inherent problems when working with continuous values, as it relies on properly set separation points when evaluating data. In addition, while GNP's contextual decision making can be beneficial, it can also prevent GNP from using its full range of sensors when making decisions, as only one node is evaluated at any point in time. For GNP to use its full range of sensors in any one decision, it needs to build a decision tree involving every judgement node in its repertoire, which is hard to evolve. Neural networks naturally evaluate all available data simultaneously, making decisions based on the overarching state of the controller instead of an arbitrary context. In addition, neural approaches naturally work with continuous values, and as such might be better suited for evaluating real-valued data. GNP and neural networks have complementary strengths, and as such, a hybrid approach is worthy of investigation.

The purpose of this thesis is twofold: first, to investigate the potential of GNP in evolutionary robotics, by investigating its performance in the foraging domain. Second, to create a hybrid algorithm combining the strengths of both GNP and NEAT. In this chapter, details regarding the GNP implementation, as well as the precise workings of the novel algorithm, will be explained. Each subsection will explain both the choices made and the reasoning behind them, as well as any key implementation details. Source code for the full project including simulator and comparison algorithms is publicly available at https://github.com/TrulsStenrud/GNP-with-roborob3

4.1 GNP

This section includes a detailed description of the GNP model, focusing on node choices, gene structure and the evolutionary loop. This section presumes knowledge regarding the basic principles of GNP, which are explained in 2.3.

4.1.1 Node Library

GNP works by evolving connections between nodes, and as such, the available nodes, as well as their configurations, have a large impact on the performance of the algorithm. The list of available nodes is referred to as the *node library*, and it defines what GNP agents are able to sense in the world as well as which actions are available to them. The node library contains node blueprints, which, on initialization, GNP genomes use to generate an individual node list with multiple copies of each node. An overview of the available nodes as well as their possible values is available in table 4.1.

Symbol	Id	Content	Possible values
J_0	0	Judge object to the left	Nothing, resource, agent, wall
J_1	1	Judge object straight ahead	Nothing, resource, agent, wall
J_2	2	Judge object to the right	Nothing, resource, agent, wall
J_3	3	Judge object to the back?	Nothing, resource, agent, wall
J_4	4	Judge direction to closest nest?	Left, forward, right, back
J_5	5	Is carrying a resource	True, false
J_6	6	Judge pheromones at current location?	True, false
P_0	0	Set speed x	[-1,1] as ratio of max speed
P_1	1	Set rotation x	[-1,1] as ratio of max angular velocity
P_2	2	Drop pheromone	-

Table 4.1: Available nodes and their possible values.

Judgement nodes

GNP networks evaluate one node at a time, which can cause problems in cases where agents have a lot of similar sensors available to them. To understand why, envision an agent with two forward facing object sensors pointing in slightly different angles. To create a well functioning "if resource in front, then move towards it" rule would require a network structure that checks both sensors, which is more complex than a one-sensor solution, and therefore takes more time to evolve. The simulated agents which the networks will be controlling each feature 8 distance sensors, which might lead to such problems. To avoid potential problems, the 8 distance sensors are partitioned into 4 judgement nodes (J_0-J_3), namely left, forward, right and back. Each of these judgement nodes have four possible outputs: nothing, resource, agent or wall. In accordance with the state of the art in foraging (see 3.1), there is also a judgement node for detecting the agents relative orientation to that of the nest (J_4), a node telling whether or not the agent is currently carrying a resource (J_5), as well as a node for detecting pheromones at the agents current position (J_6).

Processing nodes

For controlling the agent, there are 3 available processing nodes. The "set speed x" node (P_0) sets an agents translational velocity to that of a value embedded in the node ([-1, 1]). This value is a ratio, so the actual speed of the agent will be set to the embedded value multiplied by a simulator specific variable for maximum translational speed. Simulator specific parameters will be elaborated on in section 5.1. The embedded value in each node is individual to the node and subject to evolution, as the node library only contains blueprints, with no preset embedded values. A "set rotation x" node (P_1) is also supplied, and works the same way as the "set speed x" node, dictating the agents angular velocity instead. Finally, GNP networks have access to a "drop pheromone" node (P_2), which places a pheromone at the robots current location. Keep in mind that multiple copies of nodes are available to each GNP network, and that each copy of the "set speed x" and "set rotation x" nodes may have different speed/rotation values.

4.1.2 Genetic Operators

With the exception of GNPvs (see 3.4), most GNP implementations use a fixed number of nodes per network, using evolutionary operators to optimize connections between them. This model uses the same approach, but with one addition. Since the "set speed x" and "set rotation x" processing nodes rely on an embedded value, a method for changing that value is necessary. Previously discussed approaches that depended on embedded node values have, among others, relied on reinforcement learning [Mabu et al., 2006] and fuzzy reinforcement learning [Sendari et al., 2011] to modify them. For this implementation, such values will instead be self-adaptive and subject to evolution. The reasoning behind this decision is that including reinforcement learning in GNP would make it a hybrid algorithm. One of the primary goals of this thesis is to determine if GNP has potential in the field of evolutionary robotics, and a non-hybrid GNP model will go further in determining that. When selecting genetic operators for GNP, it is important to select operators that don't disrupt inactive parts of the network (see 3.4). Because of this, the simplified crossover and mutation operators introduced in [Li et al., 2018] will be used for this implementation. A detailed description of the genetic operators is provided below:

1. **Simplified mutation** is a modified version of uniform mutation. Uniform mutation works by giving each connection an equal chance of being chosen for mutation, then changing the connections target destination to that of a randomly selected node. Simplified mutation is similar, but instead of selecting a random connection, it selects a random connection that was used at least once during the previous fitness evaluation. The chosen connection is able to target unused nodes in order to activate unused parts of the network. An example of simplified mutation can be seen in figure 4.1 where the second connection of node 4 (C_{4-2}) is selected to randomly target another node, in this case node 3.

2. **Simplified crossover** is a modified version of uniform crossover. In uniform crossover, two parents p_1 and p_2 are selected. A connection is selected at random, and its target node is swapped between p_1 and p_2 to form two children. Simplified crossover differs slightly from uniform crossover in the sense that only connections that were used by at least one of the parents during the previous fitness evaluation are considered for crossover. An example of simplified crossover can be seen in figure 4.2, where connection C_{2-1} is selected, and the target of the connection is swapped between the two parents in order to form two children.

3. **Simplified value mutation**. Certain processing nodes rely on a node-specific value V, which is modified by this operator. Simply put, a random processing node is chosen, then has its V-value reassigned to a random number within $[-1, 1]$. Like the other simplified operators, this operator is only applied to nodes that have been used at least once in the previous fitness evaluation.

4.1.3 Evolutionary Loop

Like most evolutionary algorithms, GNP uses a loop of evaluation, selection, reproduction and survivor selection. Most parts of the GNP loop follow standard EA procedures, but the simplified operators necessitate a slightly different architecture in the reproduction stage. Most EAs first select two parents, then create children from their genetic material, either by cloning them or by applying a crossover operator. The children (whether they are clones or not) then have a chance of undergoing a mutation. Classic EA architecture allows for four types of children: perfect clones, products of crossover, products of mutation, or products of both crossover and mutation . The latter type poses a problem when using simplified operators as children born

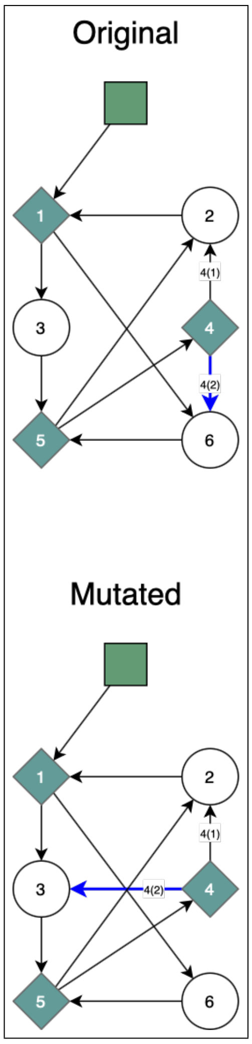


Figure 4.1: **Simplified mutation:** Node 4 and connection 2 (C_{4-2}) is selected for mutation and is randomly assigned to a new node, in this case mode 3.

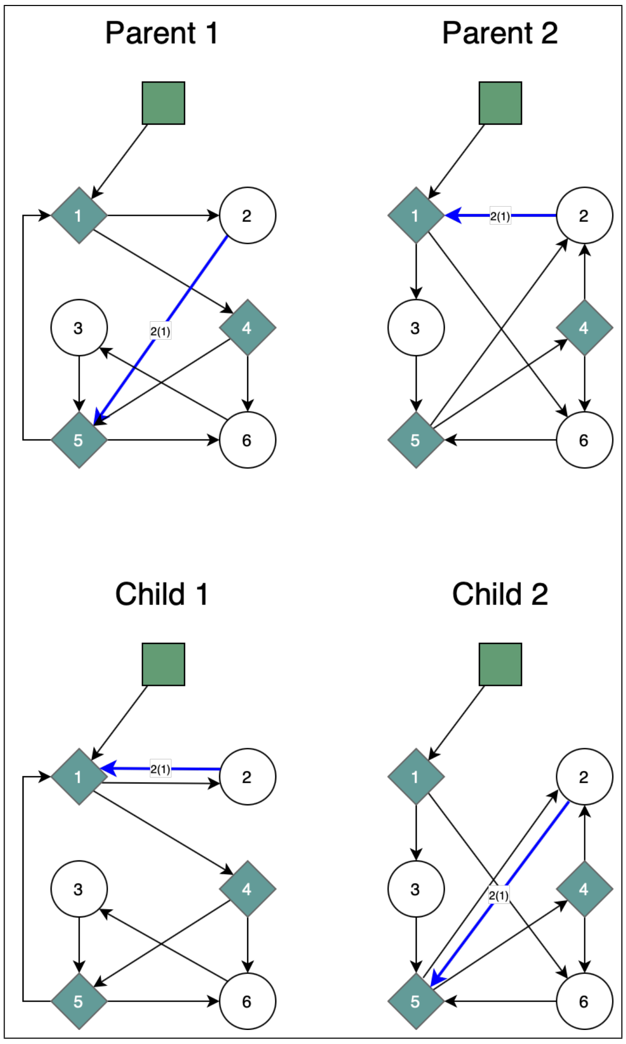


Figure 4.2: **Simplified crossover:** Connection 1 in node 2 (C_{2-1}) is selected for crossover and two parents exchange target node information to produce children.

from crossover have yet to be evaluated, and therefore have unknown node usage. To prevent this problem, a modified reproduction stage is used, where children are only allowed to go through one evolutionary operator. Figure 4.3 shows a flowchart of GNP's evolutionary loop, which can be contrasted to that of a ”normal” evolutionary algorithm (as depicted in 2.1), in that children can't undergo both crossover and mutation at the same time. If mutation is applied, there is an equal chance for any of the mutation operators to be chosen.

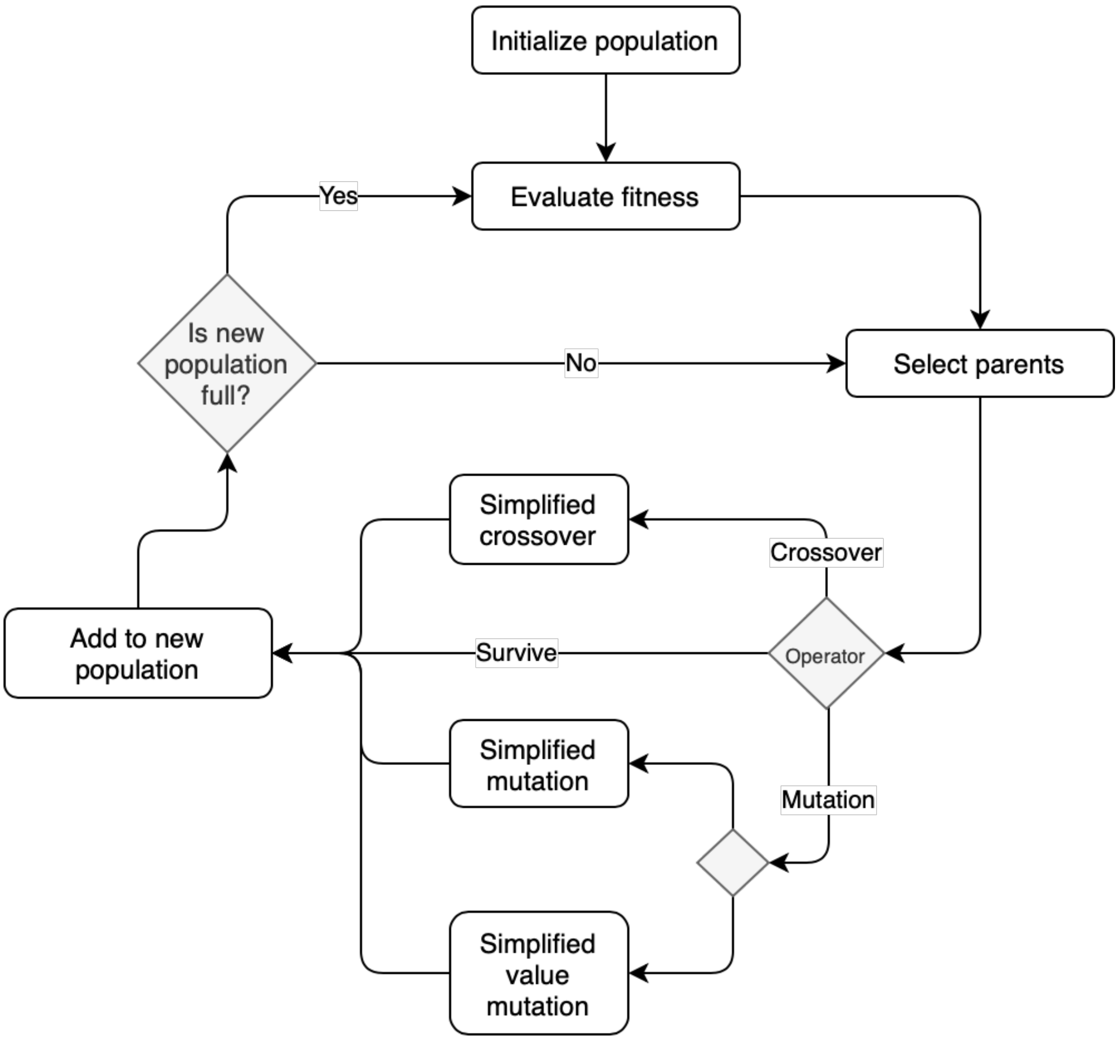


Figure 4.3: A flowchart visualizing the flow of the GNP evolutionary algorithm.

4.1.4 Gene Structure

As mentioned in the node library section (4.1.1), each GNP individual has their own individual node list. This list will be referred to as the *node chromosome*, and contains multiple copies of each available node, where the amount of copies is dictated by externally set hyperparameters. Individual node chromosomes are used as they allow for self-adaptive parameter control of embedded values in processing nodes, as well as enabling differently tuned copies of the same processing node to exist within the same GNP network. Elements in the node chromosome are objects containing descriptions of a node in the form of [K, ID, T, V], K being the node type (start, judgement or processing), ID referring to the nodes respective ID in the node chromosome, T being the time cost of the node and V being an evolved value used by certain processing nodes.

Each node has one or more outgoing connections. These connections are defined in the connection chromosome, and are identified through their respective node IDs. Take, for example, the judgement node in figure 4.4 with ID=2. The ID of the node is 2, therefore the corresponding list of outgoing connections can be found at index 2 in the connection chromosome. The list of outgoing connections is of the form [[ID,T], [ID,T]], where ID is the ID of the destination node, also known as the *target* of the connection, and T is the time cost of transiting the edge leading to the target. Once an edge has been chosen, the node with the specified ID in the node chromosome is chosen as the next node. Time costs for connections are present because they are a part of standard GNP, but the time cost of transiting an edge is always equal to zero, as is the case with most GNP implementations [Katagiri et al., 2000; Li et al., 2018; Roger and Jordan, 2015]. The reason connection time costs are always set to zero is because setting it to anything else has the same effect as changing the time cost of a judgement or processing node, so using connection time costs instead of directly changing judgement/processing time costs only hinders readability.

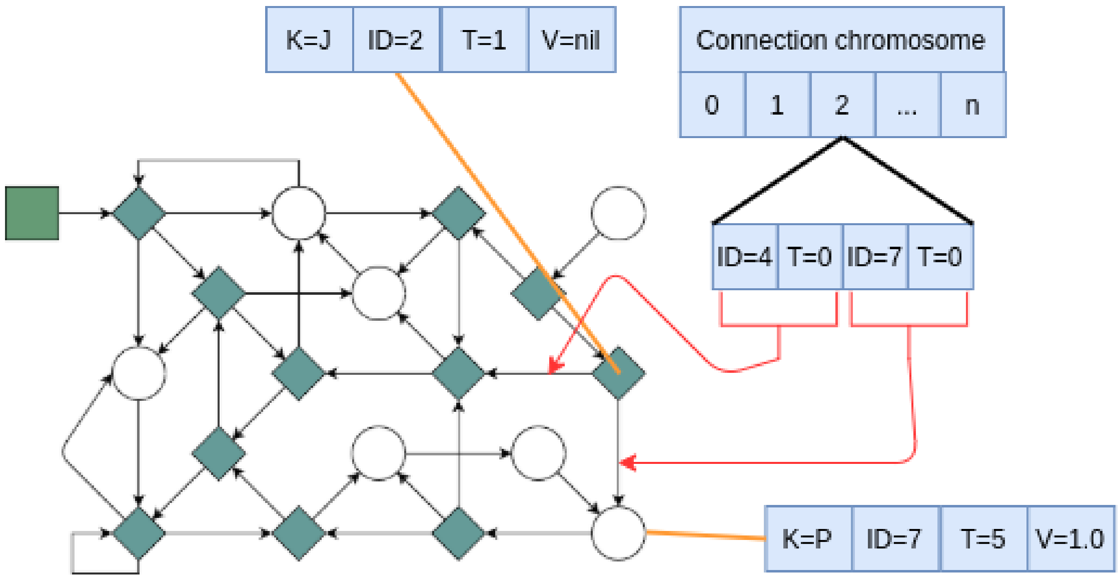


Figure 4.4: Example illustrating chromosome relations in the GNP model.

4.1.5 Hyperparameters

GNP adds multiple hyperparameters that affect performance and need to be optimized for high quality solutions to emerge:

- **Number of each processing node**: The node chromosome can contain multiple copies of each processing node. This dictates the number of copies.
- **Number of each judgement node**: The node chromosome can contain multiple copies of each judgement node. This dictates the number of copies.
- **Processing node time cost**: The time cost of using a process node.
- **Judgement node time cost**: The time cost of using a judgement node.

Too few processing/judgement nodes will limit the potential complexity of solutions, which can prevent high quality solutions to emerge. Too many nodes, however, widens the search space, slowing down innovation. How time costs affect performance is currently unclear (to the authors knowledge), but as mentioned in the background theory (2.3), time costs are generally low for judgement nodes and high for processing nodes.

4.2 GNP With Integrated NEAT

As mentioned in the chapter introduction (4), GNP and neural networks have complementary strengths. This section explores a hybrid approach between GNP and NEAT. NEAT was chosen because it is one of the most popular approaches to neural solutions in the world of evolutionary robotics, as well as being simple to use, due to NEAT evolving its own neural structure.

This paper contributes a novel hybridization of GNP and NEAT which will be referred to as *GNP with integrated NEAT*, or *GIN* for short. With GIN, a new node type called *NEAT-nodes* are introduced. NEAT-nodes are processing nodes that contain full NEAT-networks, which are evolved concurrently with the GNP part of the genotype, by an external NEAT evolver. When GIN processes a NEAT-node, the neural network inside the node receives all available sensory information, and uses it to control the agent. Specifically, for this model, it sets the agents translational and rotational velocities, as well as deciding whether or not a pheromone should be placed at the agents current position. Figure 4.5 shows an example GIN phenotype with two NEAT-nodes. Note how the NEAT-nodes are based on processing nodes, and therefore only have one output.

NEAT-nodes evolve concurrently with GNP connections by placing them in subpopulations based on their ID in the genome (see figure 4.6). At the end of each generation, the NEAT subpopulations are evolved by an external NEAT-evolver, using their respective GIN individuals fitness as their own. Meanwhile, the rest of the GIN genotype is evolved in the same way as the GNP implementation discussed in 4.1, treating NEAT-nodes like any other node.

A critical detail in GIN is that NEAT-nodes have constant IDs across the population. No matter which GIN individual is being looked at, the NEAT-node belonging to subpopulation #1 will always have the same ID, and the NEAT-node belonging to subpopulation #2 will have another ID, which is also consistent across the population. This architecture allows NEAT-nodes with certain IDs to specialize in decision making befitting the context that they are used in. For example, envision a hypothetical GIN population with a population size of 50 and two NEAT-nodes per individual, there are 50 NEAT-nodes with ID=7 and 50 NEAT-nodes with ID=8. One of each per individual. Since node connections are ID based, GIN can evolve connections that use the node with ID=7 when that is beneficial, the node with ID=8 when that is beneficial, or any other node if the situation does not call for the usage of a NEAT-node. The NEAT-nodes with ID=7 are evolved in their own subpopulation, and can therefore evolve towards behaviour that's appropriate for the context in which they are used in, while the NEAT-nodes with ID=8 evolve towards completely different behaviour. The GNP-part and the NEAT-part of GIN have a symbiotic relationship, as if GIN learns to use a NEAT-node in a certain situation, the fitness of that NEAT-node is dependant on being useful in that situation.

GIN makes use of the same simplified operators as GNP, but the contents of NEAT-nodes are always evolved, whether they're used or not, which is a potential weakness in the model. Since GIN uses a separate NEAT evolver to evolve each subpopulation, chance-based occurrences in the GNP part and the NEAT part of the genome are separate. For example, if two GIN individuals

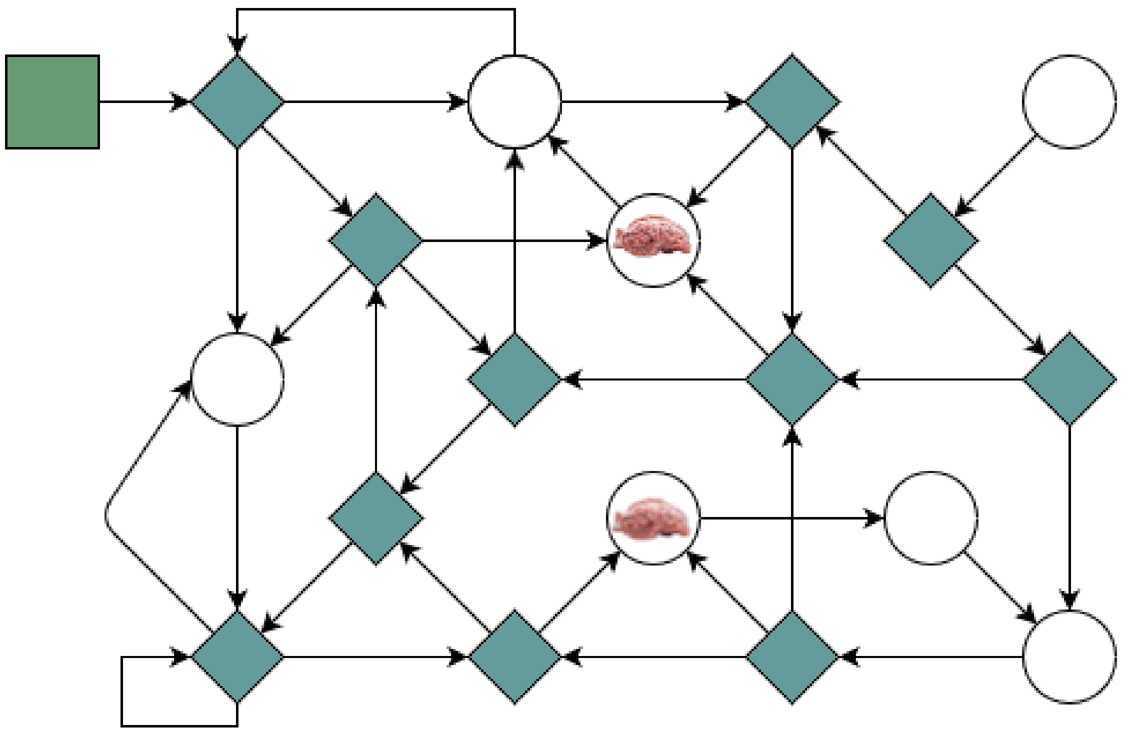


Figure 4.5: Example of a GIN phenotype with two NEAT-nodes. Sheep brain image courtesy of depositphotos.com

are selected for crossover, it does not necessarily mean that their NEAT-nodes will be selected for crossover, or even be selected as parents in the NEAT evolver at all.

In addition to the hyperparameters mentioned in the GNP chapter (4.1), GIN introduces two more hyperparameters. The first one is *Number of NEAT-nodes*, and it describes how many NEAT nodes each individual has access to. The second one is *NEAT-node time cost*, which dictates the time cost of using a NEAT node.

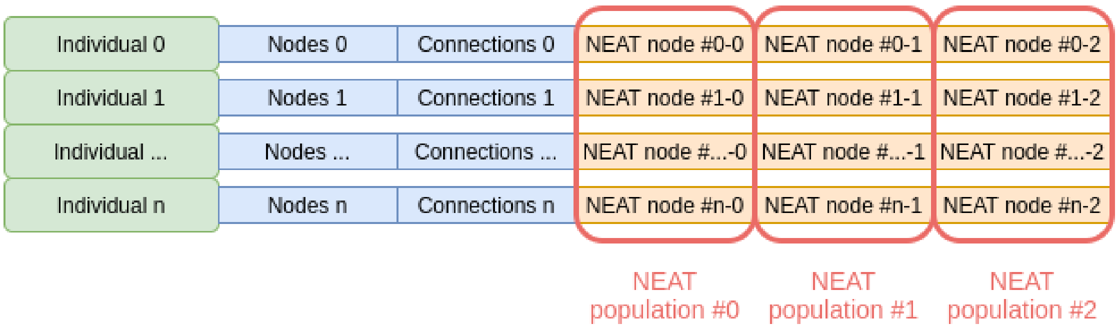


Figure 4.6: Example illustrating how multiple NEAT populations are formed from a single GIN population based on index. In this case there are three NEAT-nodes per GIN individual. Blue chromosome parts are present in GNP, whereas orange parts are new to GIN.

Chapter 5

Experiments and Results

This chapter will first present the simulated environment as well as information regarding the optimization of hyperparameters for the various algorithms featured in experiments. Afterwards, the types of graphs featured in the experiments will be explained. Experiments will be presented, along with motivation, hypothesis, results and discussion. Finally, the evolutionary behaviours of GNP and GIN are analyzed.

5.1 Foraging Simulator

One of the primary goals of this thesis is to determine if GNP has potential in the field of evolutionary robotics. Since GNP is rarely explored in neither hard, nor real valued domains, and has, to the authors knowledge, never been evaluated in the foraging domain, the design of the foraging arena was chosen with two goals in mind: normality and generality. Not to give GNP advantages, but to establish base comparisons between GNP and other methods in a basic foraging version that most swarm intelligence researchers already have an understanding of.

The foraging implementation was built on top of the 3rd version of the *Roborobo!* simulator [Bredèche et al., 2013], which is a 2D simulator made specifically for swarm robotics. Roborobo! uses real values for most information including coordinates, angles and distance sensors, as well as binary sensors where appropriate (eg: if an object is of a certain type). The simulator does not feature realistic physics, but features simple, physics-based constraints on robot movement speed and rotation speed, as well as causing robots to get stuck in each other when they crash, necessitating the evolution of collision avoidance in controllers.

Agents have access to sensory information received via a combination of Roborobo! sensors and a custom nest sensor. A summary of sensors included in the project is provided below:

- **Distance sensors**: Each robot has 8 distance sensors spread evenly in a circular pattern on their body. These sensors detect any object or wall in range.
- **Resource sensors**: For each distance sensor, there's a boolean sensor detecting whether or not the object is a collectable resource.
- **Agent sensor**: Like the resource sensors, but detects other agents.
- **Nest sensor**: Provides the robots orientation relative to the nest position.

- **Pheromone sensor**: Detects pheromone strength on the ground at the agents current location.

For interacting with the environment, all agents make the following three decisions every time step:

- How fast should I move? Range: [-1,1], which is then multiplied by a agent maximum movement speed parameter.
- What should my angular velocity be? Range: [-1,1], which is then multiplied by a agent maximum angular velocity parameter.
- Do I want to place a pheromone at my current location? [True, false]

The foraging environment was designed to resemble the environment used in [Ericksen et al., 2018]. Specifically, exactly 256 non-respawning, collectable resources are placed in the world. There are 3 different configurations dictating how the resources are placed: clustered, semi-clustered and random. In the clustered configuration, as shown in figure 5.1, the resources are placed in clusters of size 8x8 and 4x4, randomly located around the environment. Figure 5.2 shows the semi-clustered configuration where the resources are placed both randomly and in clusters of size 8x8, 4x4 an 3x3. In the random configuration (figure 5.3), the resources are placed at random coordinates. The 3 configurations propose different challenges when it comes to the task of foraging. When resources are clustered, communicating the locations of clusters via pheromones becomes important, whereas when resource are placed randomly, efficient search patterns are rewarded more heavily. Intuitively, the semi-clustered configuration requires a mixture of both behaviours.

Pheromones in the simulator always have a strength of 1 once they're placed. Every time step, a certain percentage of the pheromone strength evaporates, until a threshold is reached where the pheromone completely dissipates. If an agent attempts to place a pheromone on top of another pheromone, the strength of the original pheromone is set to its initial strength instead of a new pheromone being placed. In nature, the pheromone strength would go beyond its initial value if multiple agents deposited at the same spot. Foraging papers generally don't specify how their pheromones work in that respect [Ericksen et al., 2018; Lu et al., 2016b; Hecker and Moses, 2013; Fricke et al., 2016], so a decision was made to reset pheromones to initial strength instead of increasing them, which is contrary to nature. The reason for this is that agents tend to place as many pheromones as they can on their way back to the nest after picking up a resource. If pheromones increase in strength when this happens, it becomes harder to leave predictable duration pheromones as agents strengthen their own pheromones multiple times while building their pheromone trail. The number of times a pheromone is strenghtened by the same agent is dependent on the particular agents translational and rotational speed, which is why resetting to initial strength was deemed more dependable. Table 5.1 shows the simulation parameters that will be used in simulations, unless otherwise stated. The pheromone parameters allow pheromones to persist on the ground for a total of 1496 time steps before completely dissipating.

5.1.1 Fitness Evaluation

The goal of foraging is to collect as many resources as possible within the allotted time frame. As such, the fitness function used is the following:

$$f = \sum_{n \in N} r_n + \frac{1}{|A|} \sum_{a \in A} r_a \tag{5.1}$$

Arena size	1400x800
Time steps	Every evaluation is allotted 15 000 time steps.
Resource clustering method	Semi-clustered
Number of resources	256
Resource radius	2
Nest radius	30
Number of agents	20
Agent size	5x5
Agent max movement speed	1 per time step
Agent max angular velocity	30 degrees per time step
Agent carrying capacity	1
Pheromone initial strength	1
Pheromone evaporation percentage	0.002
Pheromone dissipation threshold	0.05
Pheromone radius	4

Table 5.1: Simulation parameters. All sizes describe circles

Where N is the set of nests, A is the set of agents, r_n and r_a are the amount of resources currently held by the respective nest or agent. Note that agents can only hold a single resource at a time, so r_a is always either 0 or 1. The second part of the fitness function is there to reward stepping stones in evolution, decreasing the bootstrap problem. A deposited resource should, however, always be better than picking up resources without depositing, which is why that part of the fitness function is divided by the number of agents. Graphs presented later on in this paper will show how many resources were collected instead of fitness, so picked up, but not deposited resources only impact fitness, not performance graphs.

5.2 Comparison bases

Several algorithms were chosen as comparison bases for different reasons. MPFA was included because it represents the state of the art in multiple place foraging. NEAT was included because of its widespread use in evolutionary robotics. NEAT with novelty search was included because it was interesting and easy to add. HyperNEAT was originally a part of the project plan as well, but had to be dropped from experiments due to running extremely slow, being estimated to finish experiment 1 in more than 3 years of running time. (see table 5.6) To ensure fairness, every algorithm involved in the experiments have had their hyperparameters optimized by a third party algorithm called *irace* [López-Ibáñez et al., 2016]. Note that every algorithm included in this project makes use of evolutionary algorithms, and that irace is used to optimize the evolutionary algorithms themselves.

In the following sections the available hyperparameters for all algorithms will be listed and explained. Note that along with algorithm-specific hyperparameters, every method will have independently optimized values for the following:

- **Crossover rate:** The chance of a crossover operator being applied. If it isn't applied, children will be copies of their parents. Crossover operators are algorithm-specific.
- **Mutation rate:** The chance of a mutation operator being applied to a child. Mutation operators are algorithm-specific.

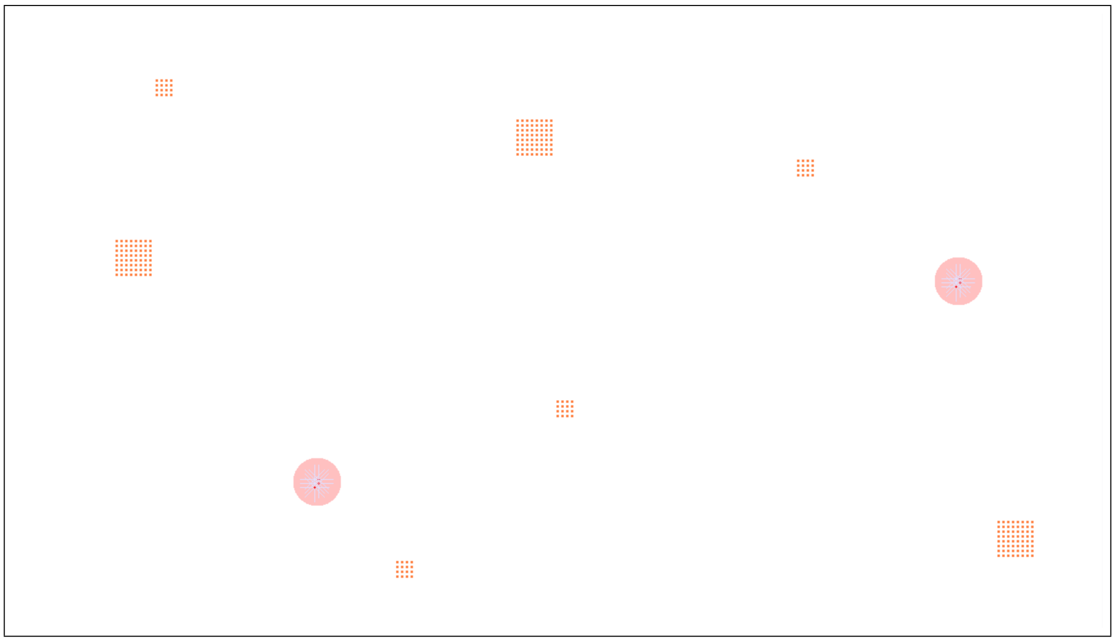


Figure 5.1: **Clustered:** An environment where collectable resources (orange) are placed in clusters of 8x8 and 4x4 objects.The red circles are nests.

- **Tournament size:** Every algorithm uses tournament selection for parent selection. This parameter dictates the number of competing genomes.

In addition, a constant population size of 50 (genome population in EAs, not the amount of agents in simulations) is enforced for each algorithm. This is done to ensure each algorithm has the same amount of fitness evaluations per generation to work with, making generational comparisons more fair.

5.2.1 MPFA

The implementation for the MPFA was adapted from the source code submitted with the original MPFA paper [Lu et al., 2016b]. The original version was made for the ARGoS simulator, so certain changes were necessary to ensure well functioning behaviour in Roborobo!. First, to prevent agents getting stuck in each other or in walls, agents will start moving erratically if they get stuck. Second, if agents fruitlessly keep moving into the same wall, they will select a new target. It is worth reiterating that MPFA simulations use different pheromones than other controllers. Whereas ordinary pheromones simply exist on the ground and have a certain strength to them, MPFA pheromones are stored in the nest and provide coordinates to any other agents visiting the nest. MPFA pheromones are still subjected to pheromone decay and evaporation parameters.

MPFA chromosomes in this implementation consist of 6 values with 6 static ranges. The ranges constrain their respective value, and the values control MPFA parameters. Crossover is performed via single-point crossover (see 2.1). Mutation is done via gaussian distribution. That is, a random number is selected from a gaussian distribution with mean value 0, and standard deviation equal to a percentage of the range of the value being mutated. The percentage is the

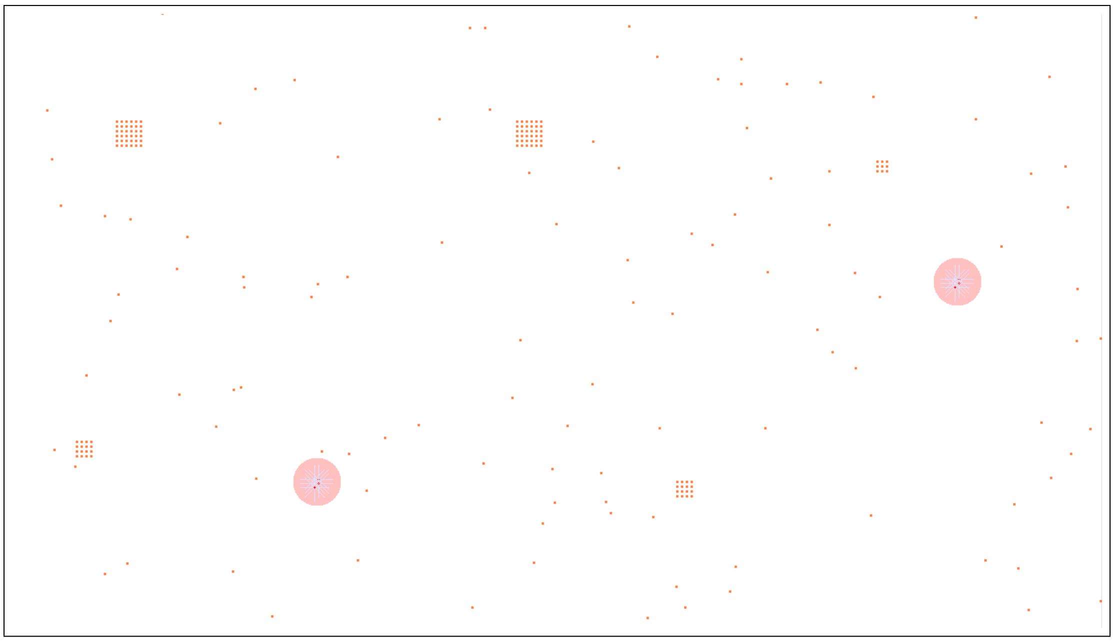


Figure 5.2: **Semi-clustered:** An environment where some collectable resources are located in clusters (8x8, 4x4 or 3x3), and some are randomly placed. The red circles are nests.

only MPFA-specific parameter which is subject to hyperparameter optimization. The ranges can be seen in table 5.2. Note that the "Rate of pheromone decay" parameter is missing, despite

Probability of switching to search	[0, 1]
Probability of returning to nest	[0, 1]
Uninformed search variation	[0, 1]
Rate of informed search decay	[0, 0.2]
Rate of laying pheromones	[1, 10]
Rate of site fidelity	[1, 10]

Table 5.2: Parameter ranges in the MPFA.

being a part of the evolver in the original MPFA paper [Lu et al., 2016b]. The reasoning behind this decision is that each algorithm should be evaluated in as similar as possible circumstances, and the other algorithms do not feature pheromone parameters in their optimizers / evolvers. Evolving pheromone parameters in other algorithms' evolvers was attempted as an alternate solution, but during preliminary testing it was found to converge to low pheromone lifespans in the early stages of evolution, where teamwork is not present, failing to evolve cooperative behaviour like trail following in the later stages of evolution. By forcing long duration pheromones, smart pheromone usage can emerge in the later stages of evolution.

5.2.2 NEAT

The NEAT (and HyperNEAT) implementation was supplied by *MultiNEAT*, a neuroevolution library created by Peter Chervenski and Shane Ryan. MultiNEAT has been in development since

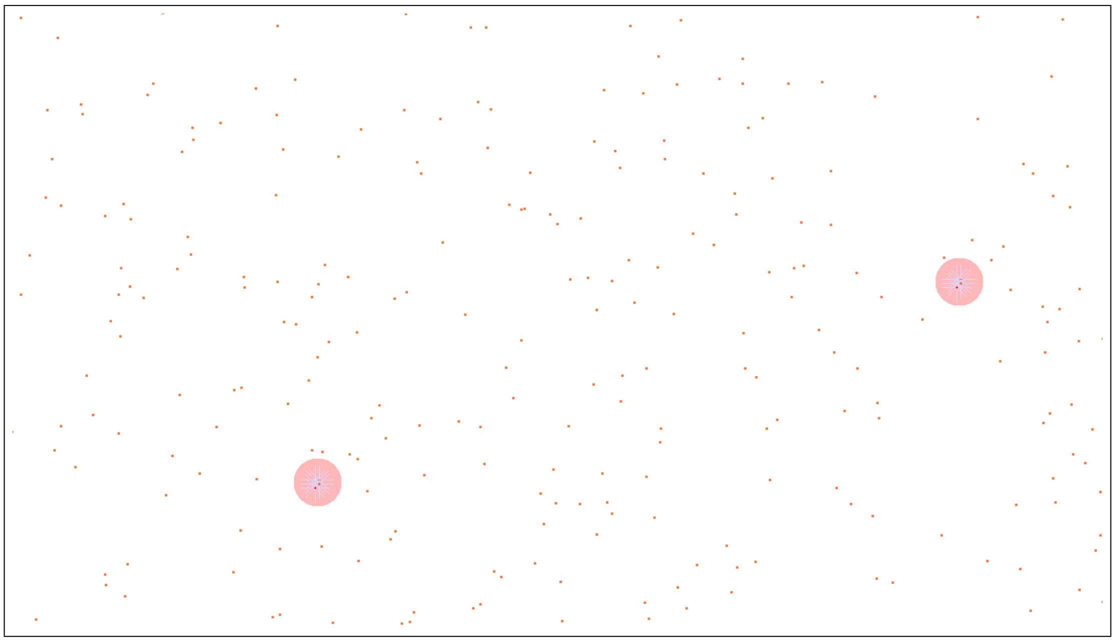


Figure 5.3: **Random:** An environment where all collectable resources are scattered uniformly at random. The red circles are nests.

2007 and supports many features proposed for NEAT over the years. MultiNEAT has over 100 hyperparameters to set, but for this project only a handfulof them are subject to hyperparameter optimization. The hyperparameters subject to optimization are as follows:

- **Young age threshold:** If an innovation is younger than this threshold (in generations), it is considered young.
- **Young age fitness boost:** Young genomes gain a slight fitness boost in the form of a multiplier. This parameter is that multiplier.
- **Old age threshold:** If an innovation is older than this threshold (in generations), it is considered old.
- **Old age fitness penalty:** Old genomes are subjected to a fitness penalty. This parameter is that penalty.
- **Phased searching:** Whether or not NEAT should use phased searching.

5.2.3 Novelty Search Based NEAT

The archiving solution as well as the NEAT controller used both came from the MultiNEAT library. *Sampled average state* (see 3.3) was used to measure novelty because of its generality and promising results in [Gomes and Christensen, 2013] . The hyperparameters subject to optimization are as follows:

- **Sampled average time window:** The size of the time window used in sampled average state.

	GNP	GIN	NEAT	NoveltySearch	MPFA
Mutation rate	0.8775	0.2275	0.5415	0.5415	0.8301
Crossover rate	0.6861	0.3973	0.4189	0.8759	0.9281
Tournament size	8	9	6	6	11
Gaussian standard deviation ratio					1.0431
Young age threshold			11		
Young age fitness boost			1.3099		
Old age threshold			34		
Old age penalty			0.8362		
Phased searching			false		
Sampled average time window				63	
Novelty search K				2	
Novelty search P_{min}				0.1992	
Dynamic P_{min}				false	
Number of each processing node	9	6			
Number of each Judgement node	5	8			
Number of NEAT nodes		3			
Processing node time cost	4	5			
Judgement node time cost	1	1			
NEAT node time cost		2			

Table 5.3: Hyperparameters resulting from the irace optimization.

- **The K constant:** Number of similar behaviours used to calculate average behaviour distance.
- **Recompute sparseness interval:** How often to recalculate sparseness for all individuals.
- P_{min}**:** Sparseness threshold for adding an individual to the archive.
- **Dynamic** P_{min}**:** whether or not P_{min} should be dynamically adjusted over the course of the run, based on how frequently individuals are classified as novel. If most of a new generation are classified as novel, selection pressure is most likely too low, so P_{min} is raised. If too many evaluations pass without finding novel behaviour, P_{min} is lowered.

5.3 Parameter Tuning

In order to ensure fair comparisons, the hyperparameters for all algorithms have been tuned by an external parameter tuning algorithm known as irace [López-Ibáñez et al., 2016]. The hyperparameters used for all the algorithms is shown in table 5.3

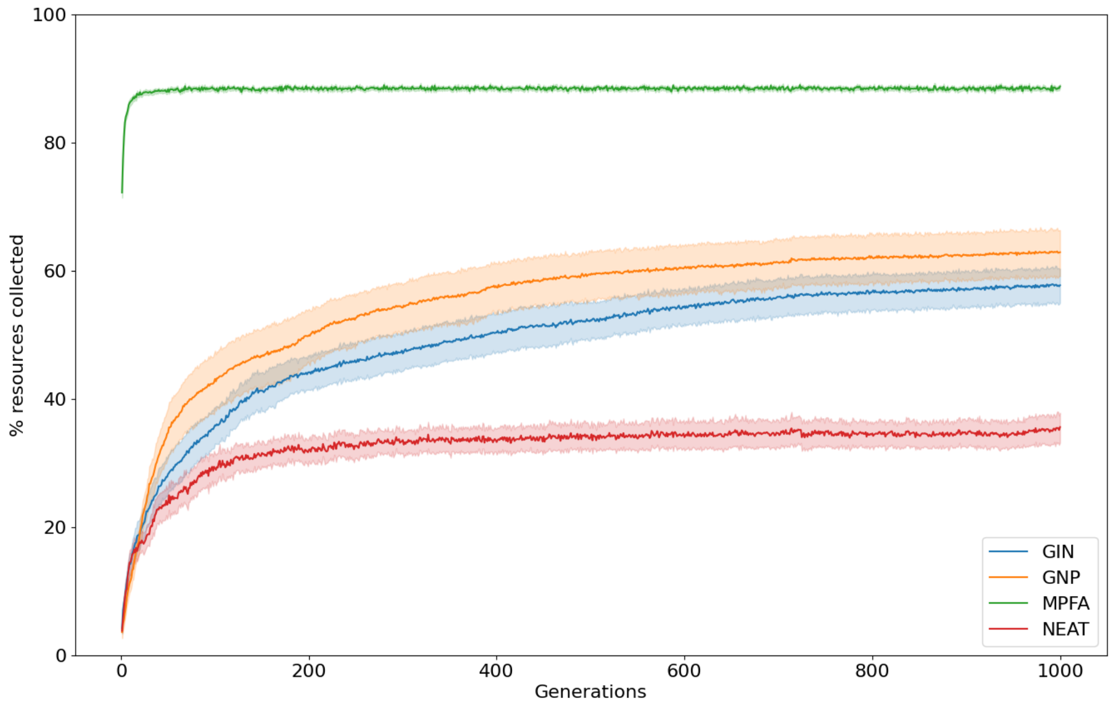


Figure 5.5: Example of a linechart. The center lines show average values and thick outline shows the 95% confidence interval.

5.4 Visualization Tools

There are two primary graph types which will be featured in the experiments. First and foremost, there is the boxplot, whose main responsibility in this thesis is to convey detailed information regarding the distribution of fitness values, which is primarily done through the "median", "quartile" and "extreme" parts of the graph. The anatomy of a boxplot can be seen in figure 5.4. Simply put, the zone between the upper extreme and the upper quartile is the range where the top 25% of values are found. The zone between the upper quartile and the median contains values in the 25-50th percentile. Between the median and the lower quartile is the 50-75th percentile of values, and the worst 25% of values can be found between the lower quartile and the lower extreme.

The second graph type which will be prominent in experiments is the classic line chart,

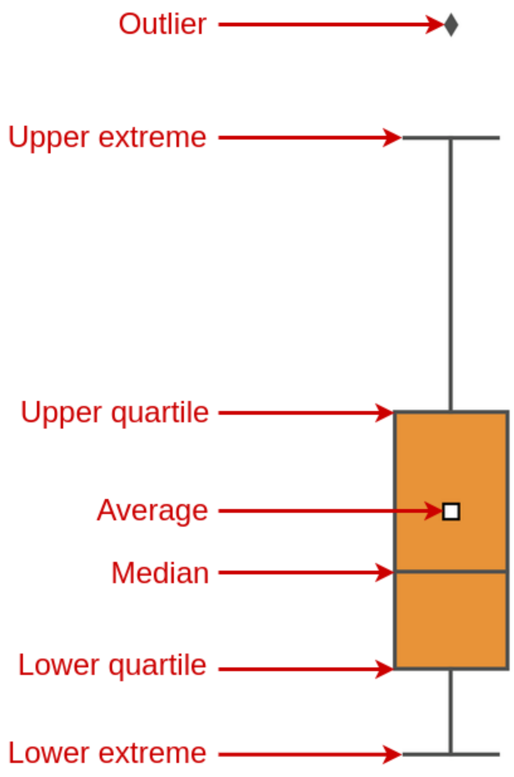


Figure 5.4: An example showcasing the various parts of a boxplot.

an example of which can be found in figure 5.5. Line charts will be used in cases where only one of the values in a boxplot needs to be shown, as line charts allow for more points to be shown and convey relative performances in a clearer manner. Line charts will show the average value, as well as a 95% confidence interval.

Data featured in graphs is calculated using the best individual in each generation. Each data point is created from 30 runs, so there will be 30 values per generation to base graph construction on. Line charts allow presenting data for each generation, so a line chart will have 30 values per data point. Box plots can't show data for every generation, and as such work on a group basis. Take figure 5.6a for example. There are 5 groups: 200, 400, 600, 800 and 1000. The box for generation 400 is built by using, for each run, the best individual within generation 200 and 400. This totals up to 30 values per box as well, but those values are selected from a much wider pool. Foraging performance is stochastic, so individual performances are highly varying on a generation to generation basis. Because of this phenomena coupled with the boxplots wider selection pool of data, performances will in general seem to be better in the boxplots than the linecharts. For example, the averages in a boxplot are actually the averages of the very best evaluations in many generations, not the average of the best in each generation.

5.5 Experiment 1

The primary research goal of this thesis is to investigate GNP's potential in evolutionary robotics by evaluating its performance in a complex domain, foraging. To do that, it is important to compare the relative performances of GNP, the state of the art and other popular methods within evolutionary computation. Comparisons against methods within evolutionary computation are important as they allow for direct comparisons between general problem solving tools that can be used in any domain, whereas comparisons against the MPFA highlights performative differences between evolutionary approaches and human-designed approaches, contextualizing results.

By comparing algorithms in a multitude of scenarios, each emphasizing different skills and behaviours, a picture is formed of the relative strengths and weaknesses of compared algorithms. As mentioned in section 3.1, different resource distributions emphasize different skills in foraging agents, which is why the first experiment in this thesis will be a comparison between all featured algorithms in four different scenarios:

- **Randomly positioned resource distribution:** Collectable resources are scattered randomly throughout the environment. Figure 5.3 shows an example.
- **Clustered resource distribution:** Collectable resources are partitioned into clusters. See figure 5.1 for an example.
- **Semi-clustered resource distribution:** A combination of clusters and randomly scattered singular resources. See 5.2 for an example.
- **Combined evaluation:** Each genome is evaluated in all three configurations, their fitness becoming the sum of all three individual fitness scores.

Each algorithm was run 30 times for 1000 generations with 50 genomes per population, for a total of 50 000 fitness evaluations per run, or 1 500 000 total fitness evaluations.

5.5.1 Hypothesis

Based on the theoretical advantages and disadvantages of each algorithm, it is expected that the relative performances of each algorithm stays somewhat similar in all four scenarios, as no individual algorithm has any obvious advantage in evolving neither efficient search patterns nor pheromone communication. GIN is expected to have an advantage over GNP due to the increased flexibility of having access to NEAT nodes, and both GNP and GIN are expected to have an advantage over NEAT due to GNP's innate advantage in dealing with the bootstrap problem. Based on preliminary testing, it is assumed that the MPFA will be close to unbeatable, whereas Novelty Search's performance will be comparable to that of a fish on land.

5.5.2 Results and Discussion

An overview of the results can be seen in table 5.4. Plots showing the evolutionary progress of each algorithm for each resource distribution can be seen in figures 5.6, 5.7, 5.8 and 5.9. The results indicate that GNP generally outperforms GIN, which in turn generally outperforms NEAT. Unsurprisingly, the MPFA wins clearly in all configurations, consistently getting close to 100% in both the random and semi-clustered configuration. Even when it isn't getting top scores (clustered), the MPFA has below 10% spread in performances, indicating that the MPFA is very robust. In contrast, GIN and GNP both managed to reach top performances in their upper extremes for the random configurations (see boxplot 5.6a), but had much worse average performances than the MPFA. GNP's average results are consistently 5-10% better than GIN's, which is unexpected as GIN should be able to ignore NEAT-nodes, if necessary, and evolve the same behaviour as GNP. Surprisingly, GNP and GIN show different performance spreads between configurations. GIN has a wider performance spread for the random configuration, but GNP has a wider one in the semi-clustered configuration. In the clustered configuration they have very similar spreads.

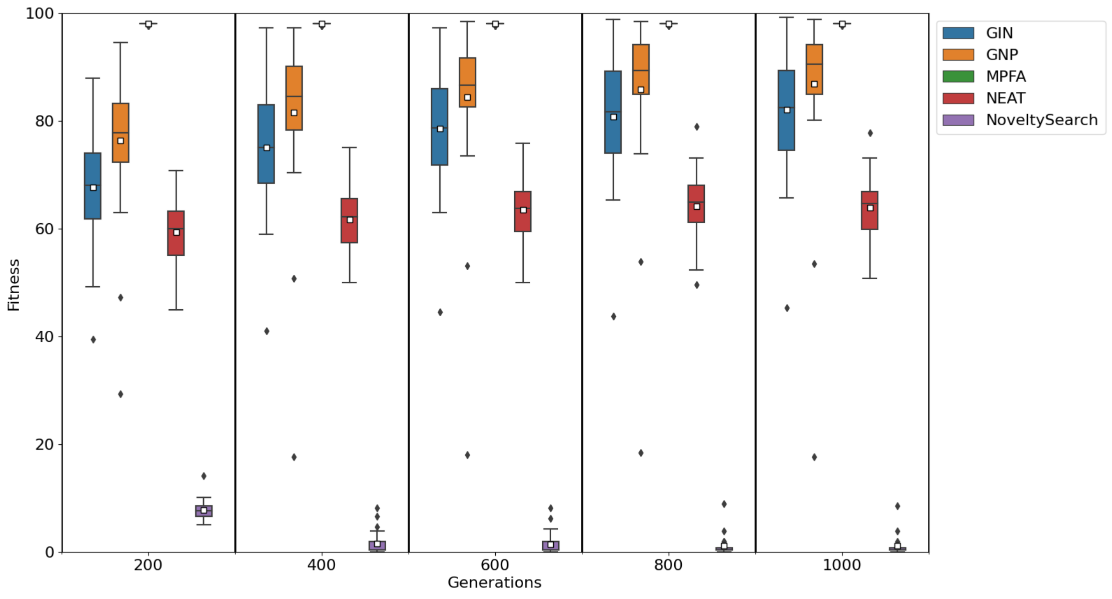
Novelty search ignores fitness, and will therefore vary in performance from generation to generation due to it abandoning positive innovations as quickly as it finds them. Because of this, it is better to look at novelty searches top performances than its averages. Even when looking at outliers however, novelty search never collects more than 14% of resources (see boxplot 5.6a), which could be caused by any combination of domain complexity, suboptimal model choices or unknown implementation issues. As mentioned in 3.3, novelty search has seen positive results in earlier works, and because of that, it's hard to confidently say anything about novelty search given the results.

[Ericksen et al., 2018] found the semi-clustered approach to be the hardest, as it requires proficiency in both efficient search patterns and communication. According to experiment 1's results however, the clustered configuration is harder than the semi-clustered configuration. Interestingly, the difficulty difference is much larger for the non-MPFA algorithms than for the MPFA, suggesting that evolving efficient search patterns is easier than evolving intelligent pheromone usage. A possible reason for the difference in difficulty gap lies in the particularities of the pheromone solution used by the non-MPFA algorithms, as the MPFA has its own pheromone solution. Pheromone sensors for other algorithms can only detect if an agent is currently on top of a pheromone, not if there's a pheromone close by the agent. The moment an agent moves off

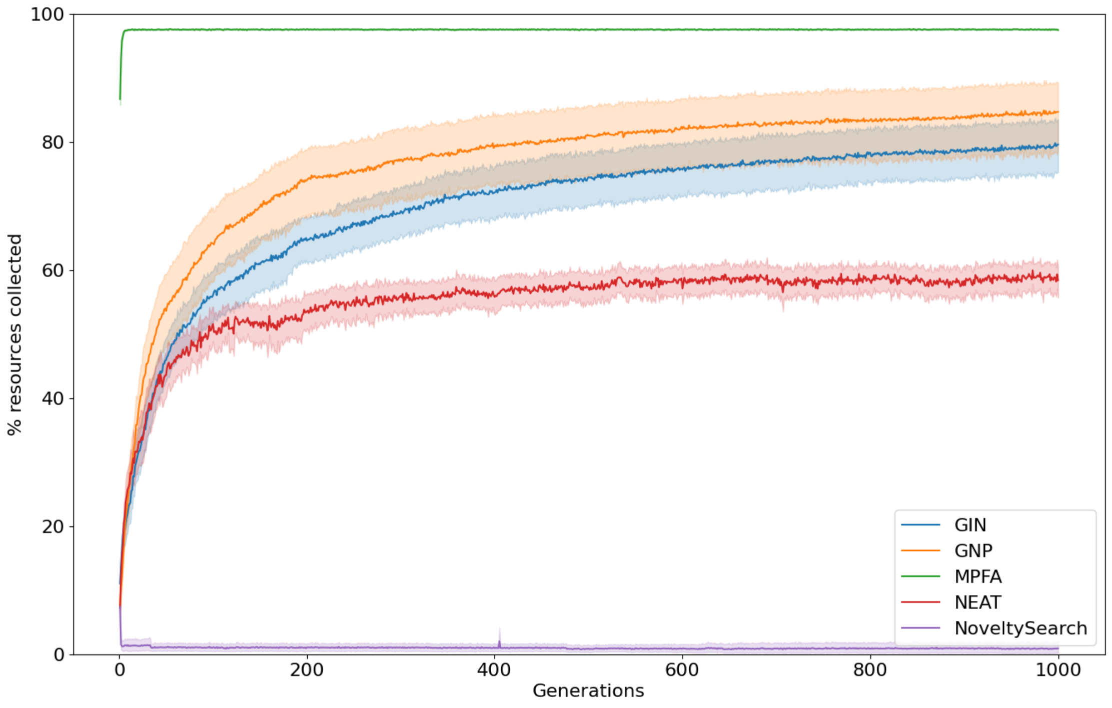
Configuration	**Algorithm**	**Median**	**Mean**	**SD**	**CI=0.95**
Random	GIN	82.62	82.25	11.28	77.97 - 86.54
	GNP	90.43	87.37	13.68	82.18 - 92.56
	NEAT	65.82	65.20	5.68	63.04 - 67.35
	MPFA	98.05	98.05	0.00	98.05 - 98.05
	NoveltySearch	7.62	7.81	1.73	7.16 - 8.47
Semi-clustered	GIN	66.74	66.04	9.38	62.48 - 69.60
	GNP	74.33	73.11	9.21	69.61 - 76.61
	NEAT	48.44	48.53	7.41	45.71 - 51.34
	MPFA	99.11	99.11	0.00	99.11 - 99.11
	NoveltySearch	4.24	4.33	0.89	3.99 - 4.67
Clustered	GIN	36.72	38.97	9.41	35.40 - 42.54
	GNP	44.14	44.57	8.00	41.53 - 47.61
	MPFA	85.94	86.41	1.43	85.86 - 86.95
	NEAT	32.23	32.83	3.49	31.50 - 34.15
	NoveltySearch	4.30	4.97	2.32	4.09 - 5.85
Combined	GIN	64.65	63.33	8.37	60.15 - 66.51
	GNP	67.28	66.10	10.43	62.14 - 70.06
	MPFA	92.81	92.77	0.64	92.52 - 93.01
	NEAT	39.40	41.20	8.67	37.91 - 44.49

Table 5.4: Statistics from experiment 1. SD is the standard deviation and CI=0.95 is the 95% confidence interval.

of a pheromone trail, it then needs to turn either left or right to get back to the trail, but agents have no way of knowing whether to go left or right, a fact that might hamper the evolution of communication too much for good pheromone usage to emerge. Theoretically, it is possible to evolve sophisticated movement patterns that can find the trail no matter which direction it is in, but such behaviour can be complex to evolve.

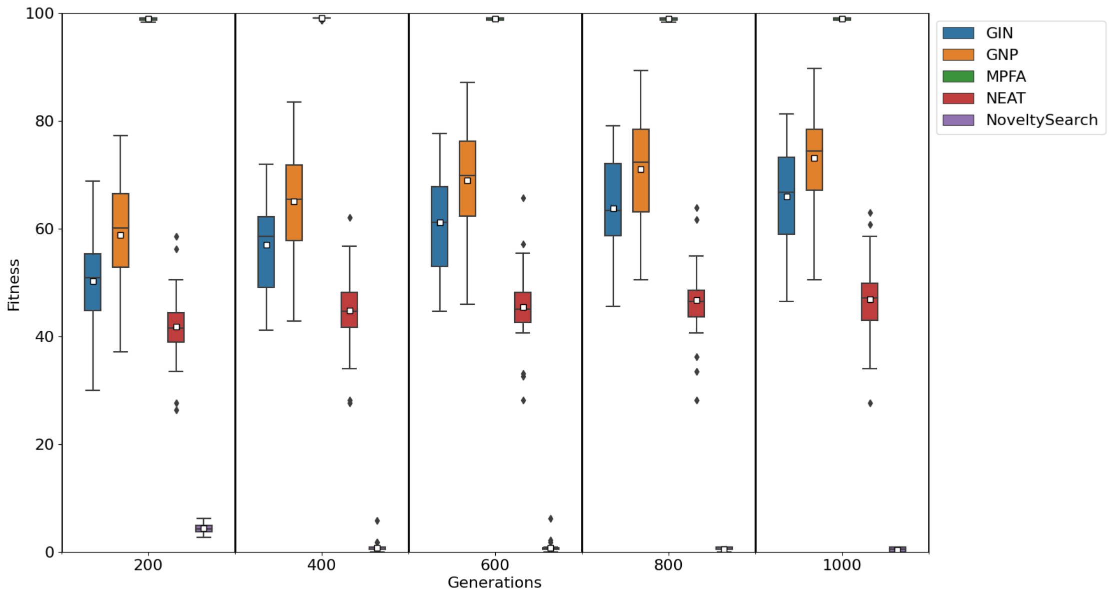


(a) Boxplot

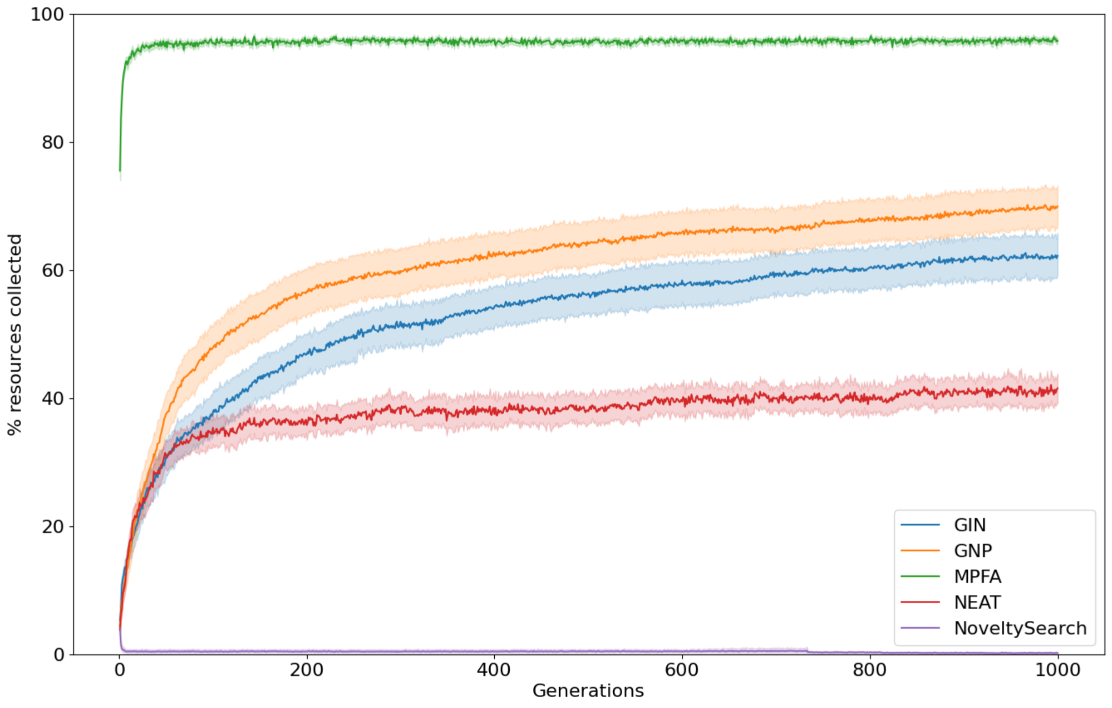


(b) Linechart

Figure 5.6: **Random:** Graphs showing performance of algorithms in experiment 1 using the random configuration. Both graphs were created using the same data.

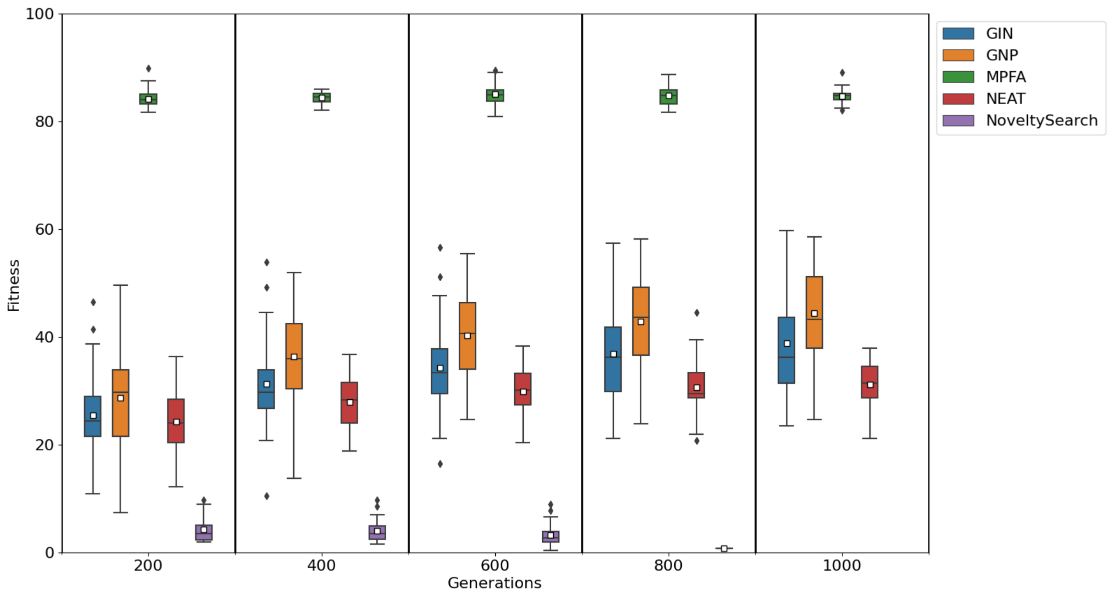


(a) Boxplot

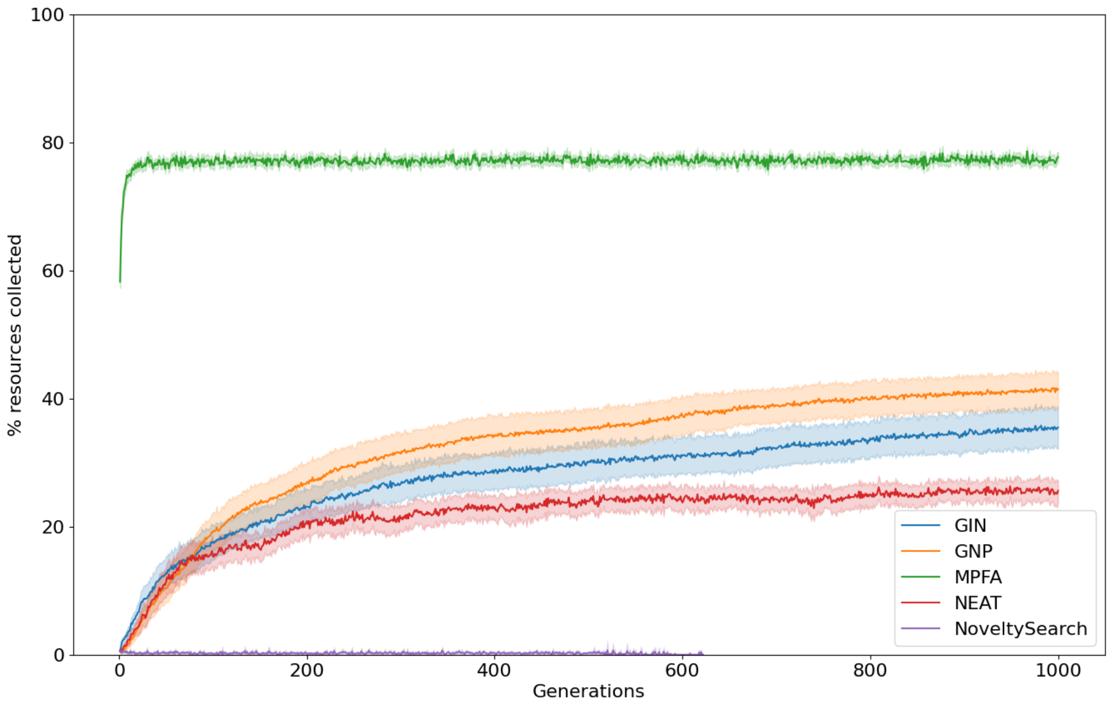


(b) Linechart

Figure 5.7: **Semi-clustered:** Graphs showing performance of algorithms in experiment 1 using the semi-clustered configuration. Both graphs were created using the same data.

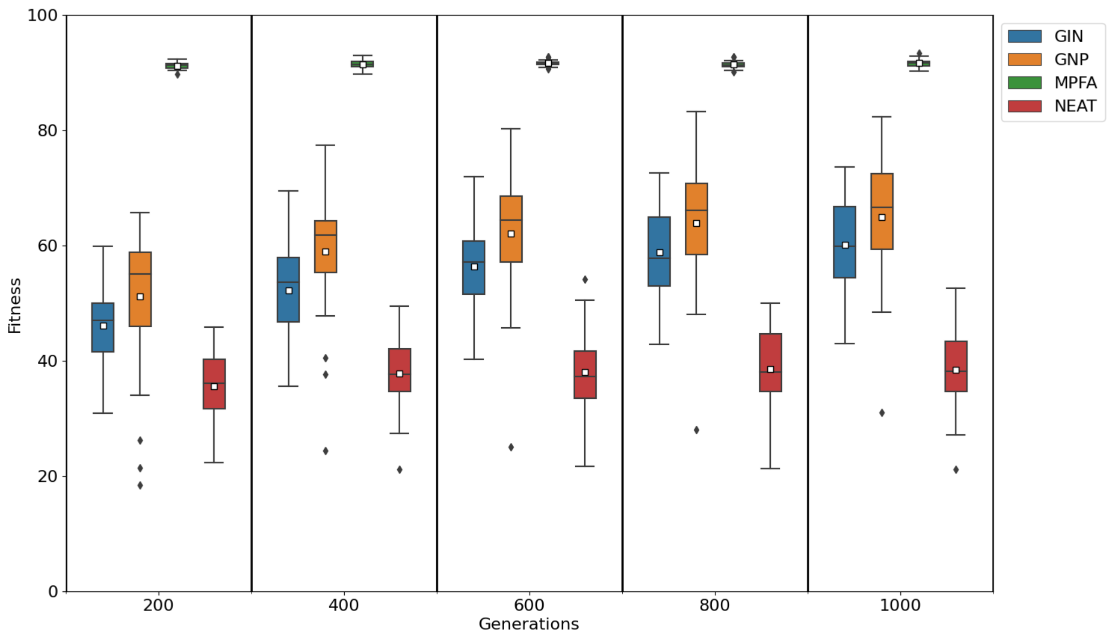


(a) Boxplot

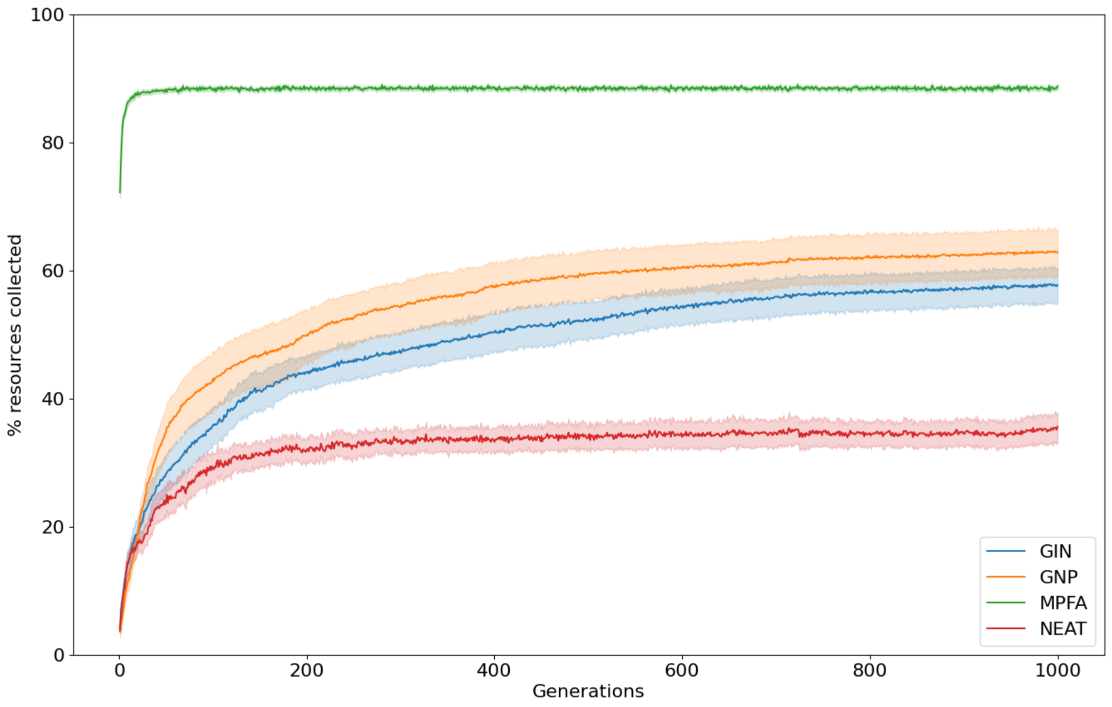


(b) Linechart

Figure 5.8: **Clustered:** Graphs showing performance of algorithms in experiment 1 using the clustered configuration. Both graphs were created using the same data. Novelty search did not finish the evaluation within its allotted 100 hours, and therefore stops after 610 generations.



(a) Boxplot



(b) Linechart

Figure 5.9: **Combined:** Graphs showing performance of algorithms in experiment 1 using the combined configuration. Both graphs were created using the same data. Novelty search is not included due to time related issues, which will be elaborated on later, in section 5.8.

5.6 Experiment 2

Swarm robotics is interesting as a problem solving tool because of its distributed and freely scalable nature. If a problem is being solved too slowly, it can often be easily remedied by adding more robots to the swarm. This aspect of swarm robotics is a big attraction, so the scalability of different algorithms is important when comparing them. In this experiment, every algorithm is compared using 6 swarm sizes: 5, 10, 20, 30, 40 and 50. Swarm sizes above 50 were avoided in order to not trivialize the problem too much. To reveal if swarm sizes have an impact on communication or search patterns, each algorithm and swarm size will be tested on all three resource distribution configurations as well. A table of all permutations featured in this experiment can be found in table 5.5.

Algorithms	**Configurations**	**Swarm sizes**
GNP	Random	5
GIN	Semi-clustered	10
NEAT	Clustered	20
MPFA		30
		40
		50

Table 5.5: List of algorithms, configurations and swarm sizes which will be combined with each other to form experiment runs for experiment 2. All algorithms will be tested on all configurations with all swarm sizes, for a total of $5 * 3 * 5 = 75$ permutations.

5.6.1 Hypothesis

The most obvious consequence of using a large number of agents is that more resources will be collected. This is expected for all algorithms, but the scaling factor of this increase is expected to be different. Larger swarm sizes should increase the importance of collision avoidance and dispersion, but should lessen the importance of smart pheromone usage, as a higher percentage of clustered resources will be picked up by pure chance as multiple agents find the same cluster.Smart pheromone usage is not only expected to be less important, but also to be harder to evolve, as more swarm members will be leaving pheromones on the map, potentially leading to information overload. In addition, as algorithms approach 100% collection rates, they are expected to stagnate in performance, as finding the few remaining resources becomes increasingly difficult. Because of the mentioned differences, the gap between MPFA and the other algorithms is expected to decrease as swarm size increases. No algorithm is thought to have an advantage in evolving dispersive behaviour, and none of the algorithms are believed to be disadvantaged in evolving pheromone usage, so the relative performance increase for non-MPFA algorithms is expected to be similar.

5.6.2 Results and Discussion

Plots showing the performance of algorithms as functions of swarm sizes on random, semi-clustered and clustered resource configurations are shown in figures 5.10, 5.11 and 5.12, respectively. As expected, all algorithms experience an increase in performance as their swarm size increases, but the MPFA quickly maxed out on performance by collecting everything, making it difficult to say anything about it in terms of scaling.A trend that's visible in all configurations is the fact that performance increases slow down as more resources are collected, with particularly

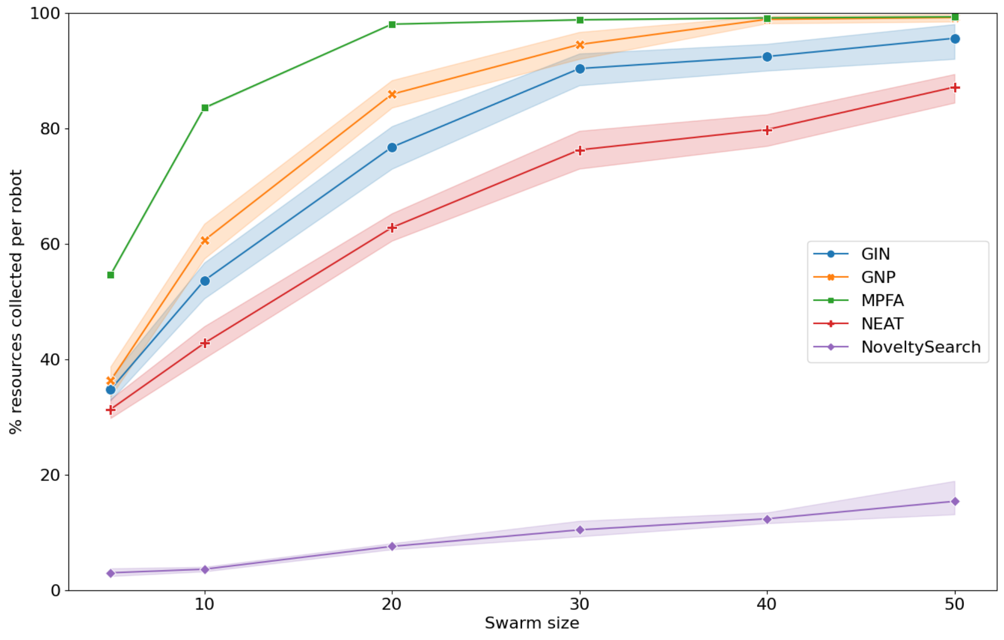


Figure 5.10: **Random:** Performance of algorithms as a function of swarm size using the random resource distribution configuration.

sharp declines appearing after 80% of resources have been collected. (see 5.10 as compared to 5.12, or GNP compared to GIN and NEAT in 5.11). This most likely happens because as more resources are picked up, it becomes harder to find the remaining resources.

Interestingly, there are minute differences in how well the non-MPFA algorithms scale relative to each other in different resource distributions. For instance, GNP appears to scale much better than GIN and NEAT in the clustered configuration (fig 5.12), suggesting that GNP has an easier time evolving smart pheromone usage than the other evolution-based algorithms. A similar story can be found in the semi-clustered configuration, where GNP scales better until the swarm size reaches 30, at which point GIN gets a sharper performance curve. GIN scaling better than NEAT in after swarm size 30 in the semi-clustered configuration (5.11) is thought to be caused by the GNP part of its genome enabling better pheromone usage, but the fact that GIN starts scaling better than GNP is most likely a consequence of the ”few remaining resources” problem outlined in the previous paragraph, which GNP starts struggling with after reaching 80% collection ratio at swarm size 30, while GIN is still at 60%, limiting the problems impact on performance improvement.

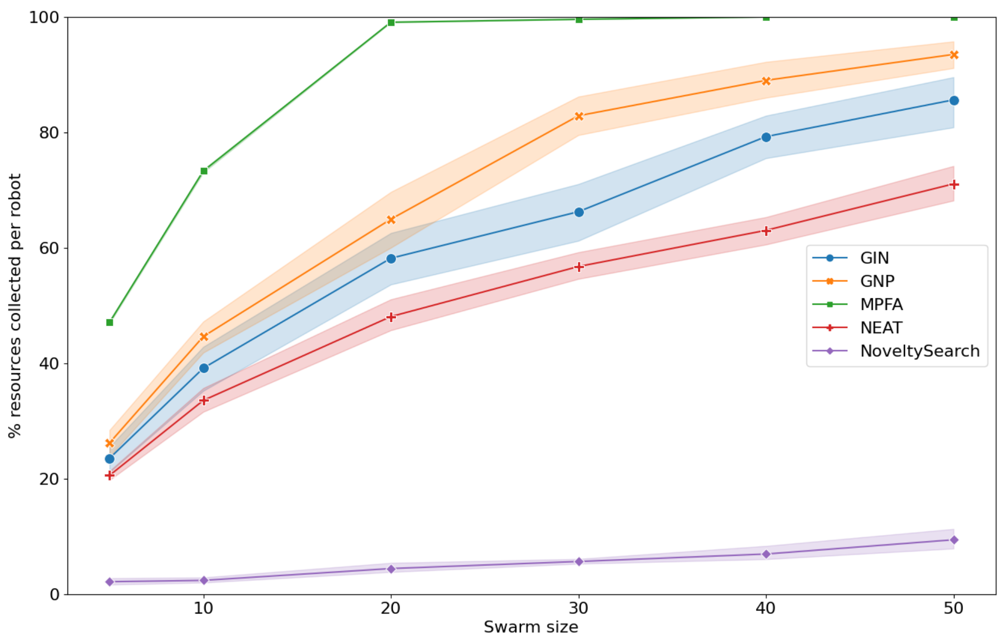


Figure 5.11: **Semi-clustered:** Performance of algorithms as a function of swarm size using the semi-clustered resource distribution configuration.

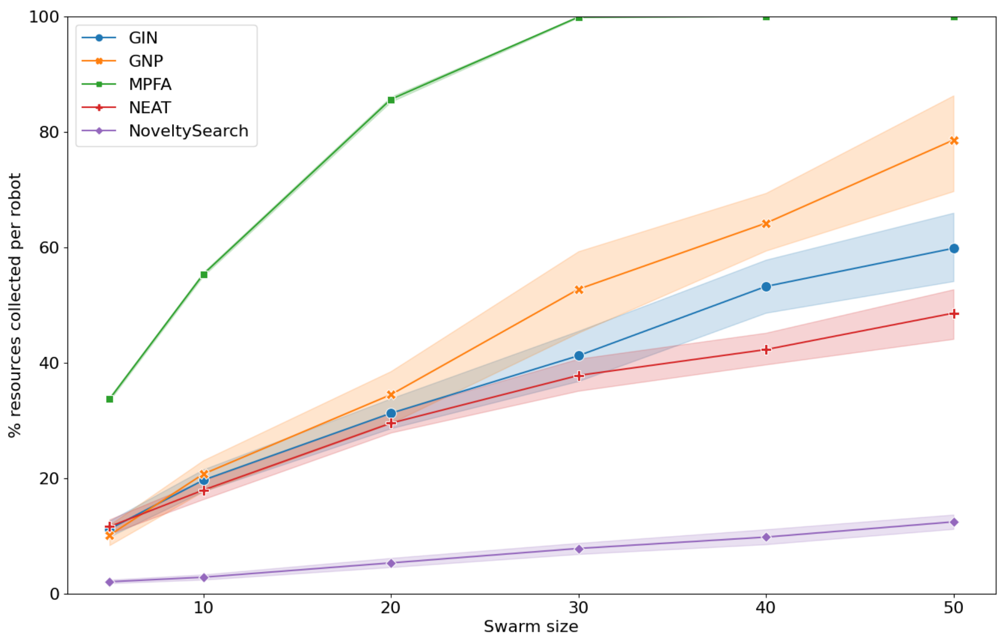


Figure 5.12: **Clustered:** Performance of algorithms as a function of swarm size using the clustered resource distribution configuration.

5.7 Experiment 3

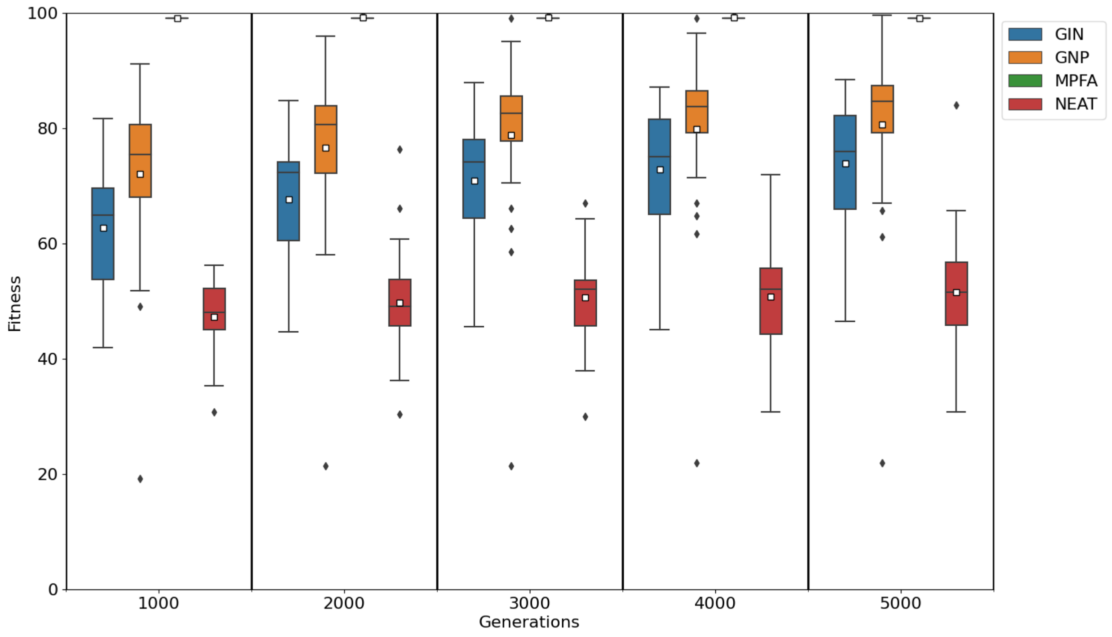
Previous experiments have been run for 1000 generations per run each. Performance curves flatten out to a certain degree over the course of those 1000 generations, but, except for the MPFA, they never fully plateau. In this experiment, all algorithms are run for 5000 generations in order to see their top performances. The semi-clustered configuration was chosen for this experiment as it was deemed the most interesting. Novelty search is not featured in this experiment due to its performance in the previous experiments couple with its slow execution time (see 5.6).

5.7.1 Hypothesis

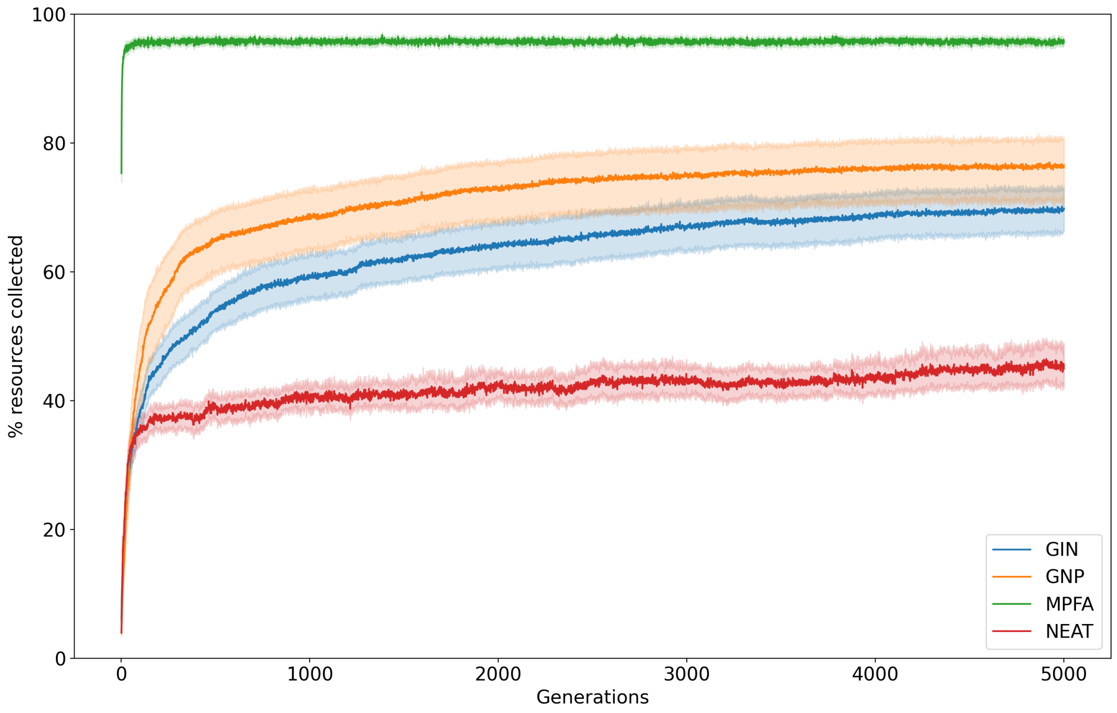
In experiment 1, GNP reached an upper extreme of 90% resources collected, whereas GIN managed a bit over 80%. By letting them evolve for a much longer time, GNP is expected to reach 100% in its upper extremes, but to still have lower average performances than the MPFA. In the semi-clustered graph from experiment 1, GIN seems to lie a constant percentage behind GNP, but as less resources are available to GNP, it should be harder for it to preserve the lead. Because of this, GIN is expected to start narrowing the gap as generations go by. NEAT is expected to make slight performance improvements, but not nearly as substantial as what GNP and GIN should experience.

5.7.2 Results and Discussion

Results of the experiment can be found in figure 5.13. GNP continued scaling at a good pace until around generation 2000, where performance improvements started slowing down until it plateaued at 75% average number of collected objects at generation 4000. The boxplot shows that GNP managed to reach 100% collected resources in its upper extremes in the final part of the experiment, which is more than the MPFA managed. The MPFA is a lot more consistent in its performance, but this result indicates that GNP has the potential to be competitive in the foraging domain, given that further improvements are made. The gap between GNP and GIN becomes smaller over time, but by very small amounts. In addition, GIN's average performance plateaus after 4000 generations, indicating that the gap between their average performances should stay at 5% even if the experiment was ran for more generations. Interestingly, NEAT never plateaued, unfortunately rendering its peak performance unknown, at least in this foraging environment.



(a) **Experiment 3:** Boxplot



(b) **Experiment 3:** Linechart

Figure 5.13: **Experiment 3:** Plot depicting performances in experiment 3 on the semi-clustered configuration.

5.8 Time Complexity

The real world applicability of swarm robotics depend on the individual robots being cheap, as budgets naturally restrict projects. If robots are cheap, one can use more of them in order to improve performance. Faster processors are more expensive than slow ones, so the time complexity of algorithms is an influential statistic. Table 5.6 shows runtime for all algorithms on experiment 1, measured in hours. Algorithms only had 100 hours to complete experiment 1, which unfortunately was not enough time for novelty search to complete 1000 generations on the clustered configuration. HyperNEAT clocked in at 34 hours per generation, which means it would take about 3, 88 years for it to complete a run.

	Random		**Semi-clustered**		**Clustered**	
	average	(std)	average	(std)	average	(std)
MPFA	8	(0)	8	(0.25)	8	(0.78)
GIN	19	(1.78)	18	(2.06)	19	(2.62)
NEAT	30	(1.57)	22	(2.42)	19	1.18
GNP	26	(5.3)	26	(5.06)	25	(5.67)
NoveltySearch	61	(22.18)	69	(22.01)	100+	
HyperNEAT	34 000	Estimated value based on one generation.				

Table 5.6: Execution times for various algorithms running experiment 1, measured in hours, sorted by time. Std is short for standard deviation.

Execution times are inconsistent partly due to the computing cluster used, IDUN, having many different types of CPU's in it. Evidence of this can be seen in, for example, NEAT and novelty search. for NEAT, the random configuration was the slowest, whereas for novelty search, the random configuration was the fastest. Despite these inconsistencies, there are clear differences between some algorithms that cannot be explained by CPU differences. First and foremost, the MPFA is state of the art in more than simply behaviour, boasting a much faster execution time than its competition. Second, novelty search runs much slower than the others. This is most likely due to the heavy computational complexity of the novelty measure used, as well as the extra overhead of maintaining a novelty archive. HyperNEAT was also extremely slow, due to its tendency to evolve very large neural networks.

5.9 GNP and GIN Behavioural Analysis

GNP is interesting as a phenotype for a multitude of reasons, one of which being the fact that it's easily analyzable. By looking at overall node usage, the relative importance of nodes can be highlighted, providing information regarding which considerations are important when developing controllers for foraging or similar domains. The difference in performance beetween GNP and GIN is interesting because GIN should theoretically be able to evolve the exact same behaviour as GNP by simply ignoring NEAT-nodes. To investigate this difference as well as answer the question of which nodes are important for foraging, this section explores the node usages of GNP and GIN for the various resource configurations.

5.9.1 Hypothesis

The rotation processing node is expected to be the most used node as it is the primary response mechanism for agents. It is important for setting up good search patterns, steering towards resources, steering towards the nest, collision avoidance, and following pheromone trails. The speed node is not expected to be used often, as moving at maximum speed should be the best option most of the time. The final processing node, pheromone deposition, is expected to be used a decent amount, as most agents place a lot of pheromones when returning to nests with resources.

As for judgement nodes, agents have 8 directional sensors, which are partitioned into 4 directional judgement nodes. Forward sensing judgement nodes are not expected to be used much, as this system enables left and right pointing sensors to detect wall collisions, even if the agent is facing a wall directly (in which case the detection happens post-collision). There are two situations that are detectable only through the forward sensing judgement node. The first is when a resource is in front of the agent, which is not a situation that needs to be reacted to, as left and right sensing nodes can correct the course in case the resource is not picked up. The second situation is agent on agent collisions, which should be reacted to. The backwards facing judgement node is expected to be mostly useless, and thus be used the least out of all judgement nodes. Because forwards and backwards sensing judgement nodes have so few necessary uses, the left and right pointing judgement nodes are expected to be used much more frequently than them. Judgement nodes for sensing pheromones and nest directions are expected to be used a similar amount, as they are important guidance nodes for finding and returning with resources, respectively. The judgement node that detects whether or not an agent is currently carrying a resource is interesting, because it can either be used a lot, or very rarely depending on how intertwined the GNP network is. A highly separated GNP network should be able to detect that it has a resource, then move to the nest without needing to recheck the carrying node, as GNP's program flow will be in a subnetwork that assumes a resource is being carried. On the other hand, intertwined GNP networks will need to check the node more often as they can't make the same assumptions.

As for GIN and its NEAT-nodes, they are expected to be used a medium amount. NEAT-nodes control actions based on all sensors, so as NEAT-nodes are used more, both judgement and processing node usage should decrease accordingly. NEAT-nodes are expected to develop solutions for specific situations, whether they are general or niche. Because of this, situational nodes like pheromone nodes and front sensing judgement nodes are expected to be especially vulnerable to having their jobs overtaken by NEAT-nodes.

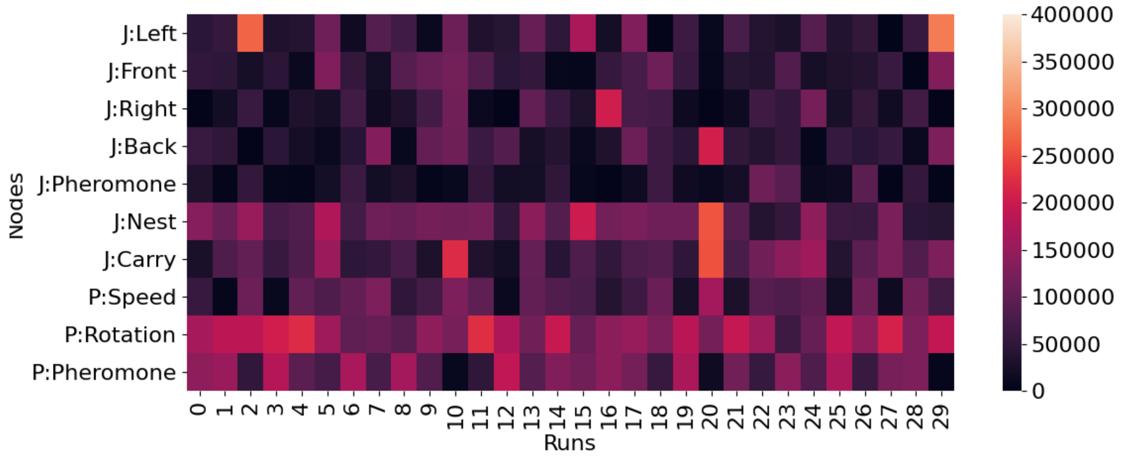
5.9.2 Behaviour Analysis

Figures 5.14, 5.15 and 5.16 depict heatmaps of node usage from experiment 1. Each box in the heatmaps shows how many times a particular node was used by the best individual in a run. Each column represents one of the 30 runs. All heatmaps show the same general pattern for processing nodes, in which the rotation node was the most used processing node, followed by pheromone placement, then the speed node. This is in line with expectations, but judgement node usage however, holds a decent amount of surprises. While there are some that do, very few solutions make heavy use of both left-sensing and right-sensing judgement nodes, instead opting for one of them, then building a solution around that choice (example: half of 5.16a). Most solutions, be they NEAT, GNP, or GIN, settle into spiral-based search patterns after a while, which could explain the tendency to only use one of the side-sensing judgement nodes. If an agent constantly turns towards the left, resources are most likely to appear on the right side of the agent.

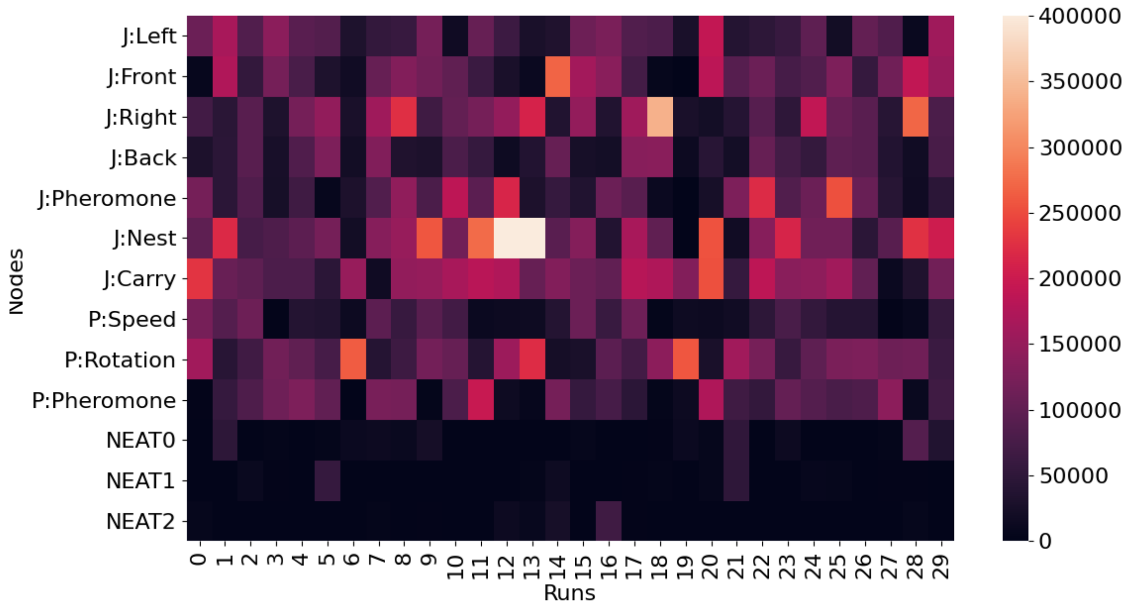
Interestingly, for GNP, directional sensors in networks are rarely used in solutions developed for the random configuration (5.14a), but are used more often in solutions developed for the clustered configuration (5.16a). GIN does not share this pattern, making use of many directional judgement nodes in all three configurations. Interestingly, GIN also appears to use the pheromone judgement node and the nest judgement node a lot more than GNP, hinting towards more pheromone usage. Both algorithms make heavy use of the nest node in all configurations, indicating that it is more useful than previously thought. Theoretically, networks can learn to use the nest sensor not only to go to the nest, but to ensure they are moving away from the nest whilst searching or following a pheromone trail. With a couple of exceptions, the general trend for carry nodes seems to be that GNP uses them a lot less than GIN, indicating that GNP evolves less intertwined networks, making better use of GNP's ability to evolve state machines. As for NEAT-nodes, only one GIN individual (#19 in the clustered configuration)used a NEAT-node actively, which is surprising. Interestingly, that individual doesn't use a single judgement node, so the NEAT-network is its only method for reacting to sensor values. Another surprising discovery is that of the ones who used NEAT-nodes at all, most of them only used a single one, despite having three available. A potential explanation for the low NEAT-node usage lies in GNP's advantage in dealing with the bootstrap problem. If NEAT-nodes are useless for too many generations, GIN populations might learn to avoid them too early, not giving them a chance to evolve useful neural networks.

In general, GIN seems to to have more variety in node usage from run to run, whereas GNP is more predictable in its node usage. It's hard to say why this happens, but NEAT-nodes certainly introduce chaos. NEAT-nodes should alter node usage as it takes over the jobs of other nodes, but NEAT-nodes are rarely used according to the heatmaps, and GIN in general sees worse performances than that of GNP, so why is this? A potential explanation is that when NEAT-nodes are used, they change what the "optimal" network structure should be, depending on the content of the NEAT-node. As a result, children of NEAT-node-using networks naturally evolve towards different network structures than those of individuals who don't rely on NEAT-nodes, leading to a natural split in the GIN population. Children created by crossover between these two hypothetical subpopulations will then have higher chances of being bad, slowing down evolution. Eventually, one subpopulation should start dominating the population, but even then, a mutation/crossover that brings NEAT-nodes back can then start rewarding destructive changes. If this theory is correct, it could be regarded as a digital case of reinforcement speciation (ex: horses and donkeys), which would be interesting. Figure 5.18 shows a snapshot of node usage in a single GIN population at generation 475. In the figure, one can see that NEAT-node usage varies among the population, with some using the first NEAT-node more than others (IDs #8, #22, #40, #48), some including the second one as well (IDs #27, #39, #45) and some barely using NEAT-nodes at all (IDs #17, #35). The variety present in the population could indicate that there's substance behind the theory presented above, but more evidence is needed to make conclusive statements.

Figure 5.17 shows the best individual in every generation over a single 500-generation GIN run. The figure itself is handpicked from a selection of 30 preliminary tests, but it showcases trends that can be seen across most of the preliminary tests. Common trends include seemingly random micro changes in node usages from generation to generation. Foraging behaviour is stochastic, so individuals will slightly change their node usage each time they are evaluated. This phenomena can not only lead to such small changes in node usage, but also change which individual is the best one for that particular generation, in which case larger node usage changes are visible (see generation 48, 128, 320). Another common phenomena that happened 1-4 times per run in the data from preliminary tests, is a somewhat drastic change that lasts for many generations, caused by another individual consistently surpassing the performance of the previously highest performing individual (see gen 176, 416). This indicates that multiple niches in the evolutionary search space are being explored, which is good as it means the algorithm isn't prematurely converging. More evidence of GIN not prematurely converging can also be seen in figure 5.18, as multiple different node usage patterns are present in the population at generation 475. Another visible trend is that NEAT-nodes are primarily being used in the earlier stages of evolution, lending credence to a previously described theory, that NEAT's poor bootstrapping speed is negatively affecting GIN.

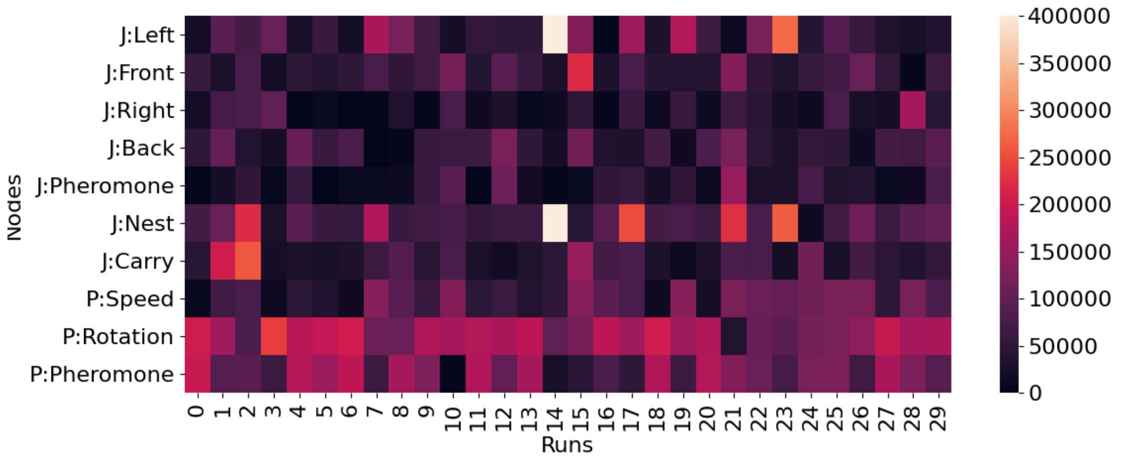


(a) GNP

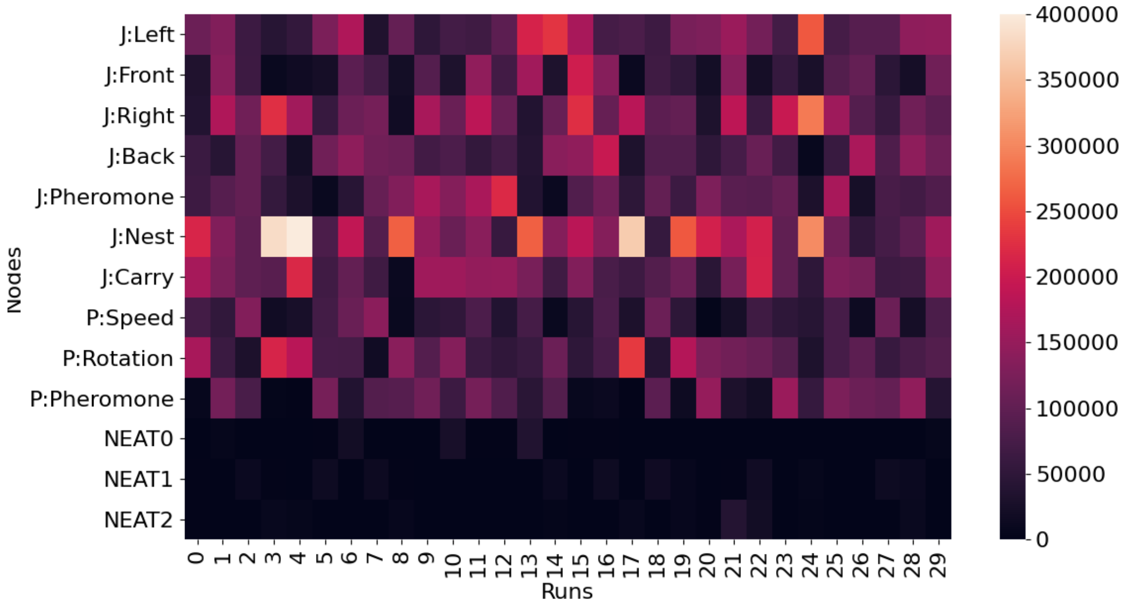


(b) GIN

Figure 5.14: **Random:** Heatmaps showing node usage of GNP and GIN in experiment 1 on the random resource distribution. Each box shows node usage from the best individual from that run.

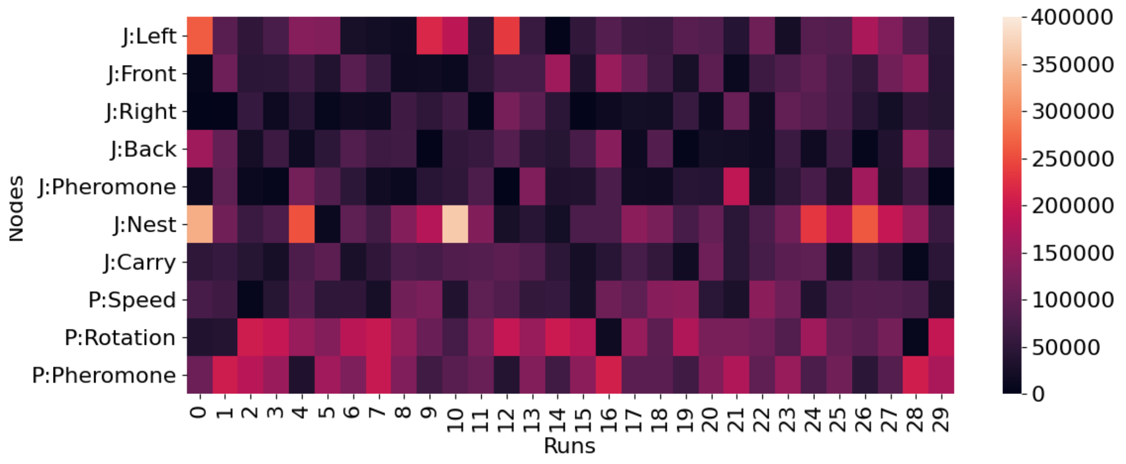


(a) GNP

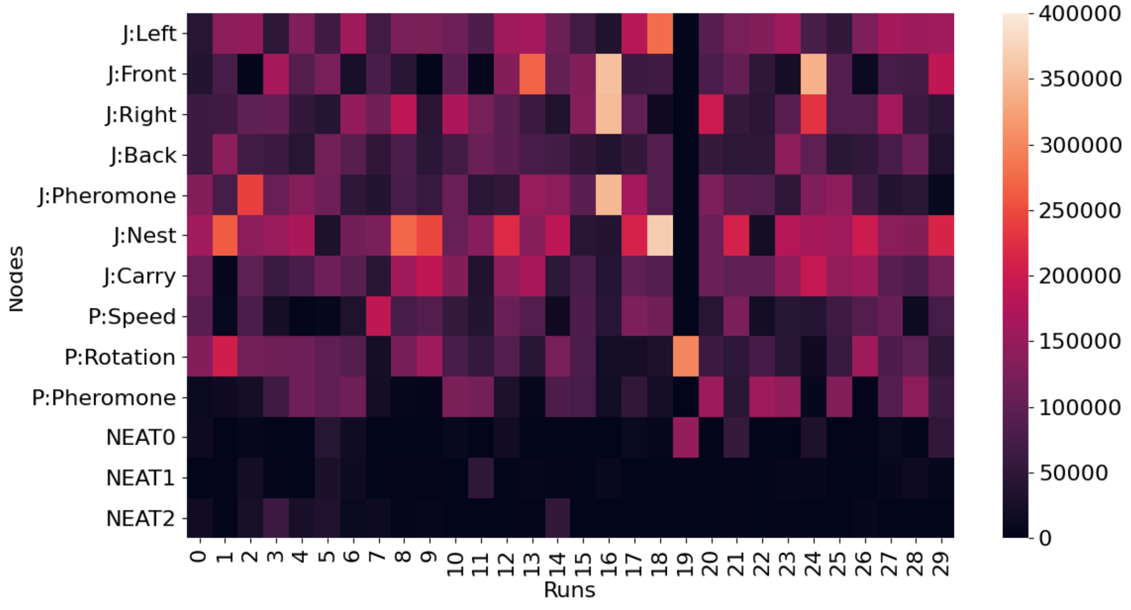


(b) GIN

Figure 5.15: **Semi-clustered:** Heatmaps showing node usage of GNP and GIN in experiment 1 on the semi-clustured resource distribution. Each box shows node usage from the best individual from that run.



(a) GNP



(b) GIN

Figure 5.16: **Clustered:** Heatmaps showing node usage of GNP and GIN in experiment 1 on the clustered resource distribution. Each box shows node usage from the best individual from that run.

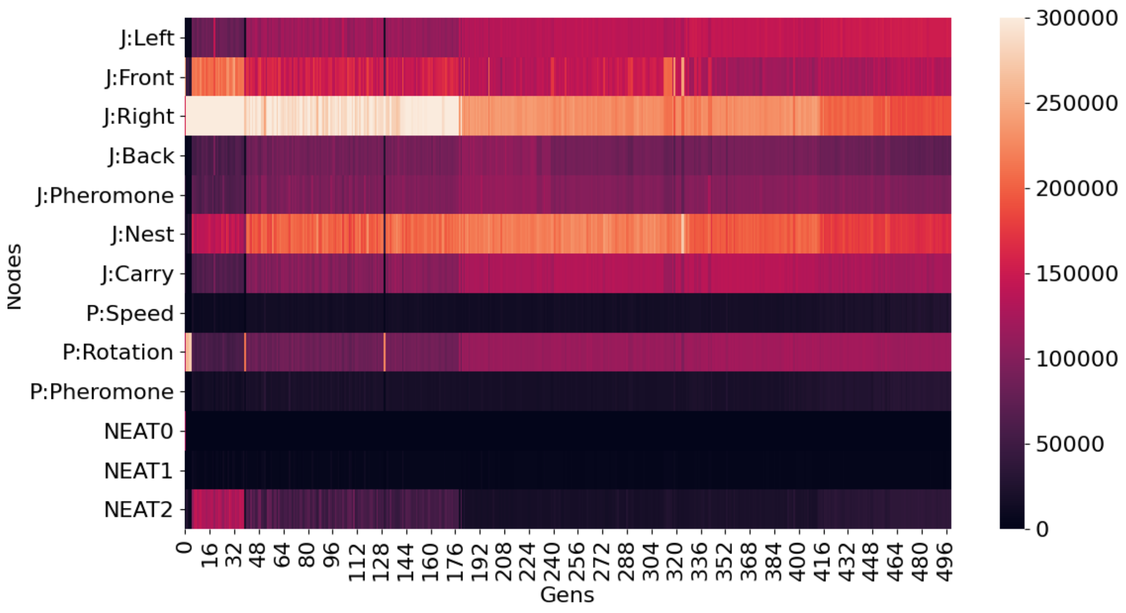


Figure 5.17: Node usage of the best individual in a GIN run, measured every generation. The run was carried out on the semi-clustered resource placement configuration.

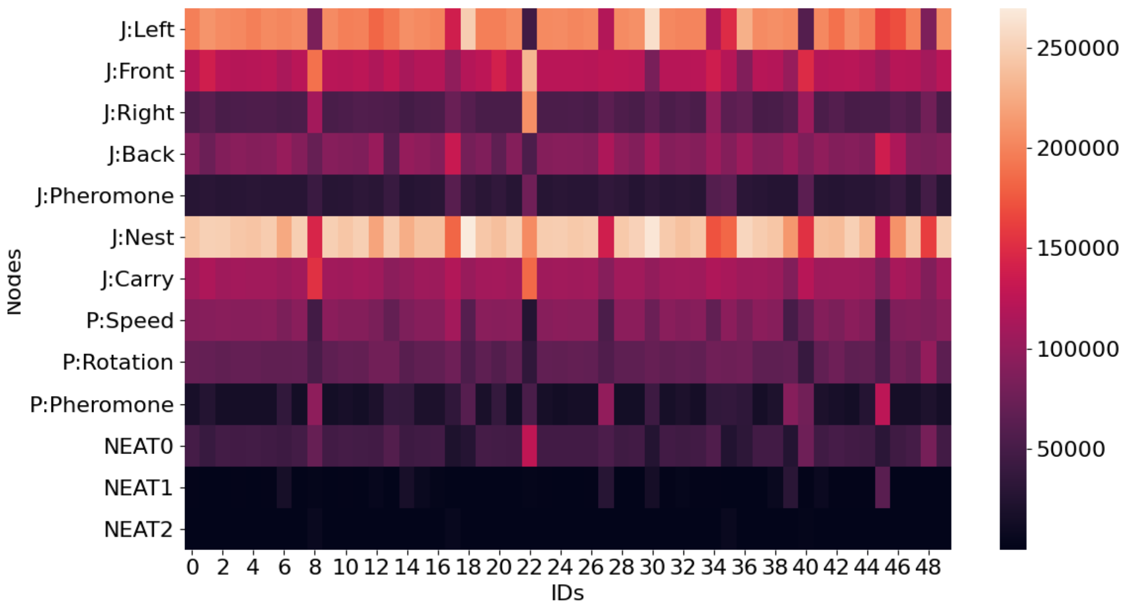


Figure 5.18: Snapshot depicting the node usage of a GIN population at generation #475.

Chapter 6

Evaluation and Conclusion

In this final chapter, the original goals of the thesis are revisited, looking at successes, failures, strengths and weaknesses of the work presented. A list of potential directions for future work is presented as well as concluding statements.

6.1 Evaluation

This thesis had two primary goals. **First:** to investigate GNP's potential in evolutionary robotics by comparing it to state of the art methods within the foraging domain. **Second:** to investigate a potential improvement to GNP, by hypridizing it with NEAT. To reach these goals, 7 research questions were formulated. This section takes an individual look at the answer to each of these questions, then uses that information to evaluate the primary goals.

RQ1 - Is GNP competitive in the foraging domain when compared to the current state of the art?

Based on the results of the experiments, the answer to this question is twofold. When compared to the state of the art within foraging in general, including human-designed approaches, GNP as featured in this thesis is not competitive. When compared to state of the art approaches in automatic design however, GNP is definitely competitive, doubling NEAT's performance in some cases.

RQ2 - How do resource clustering methods affect GNP compared to the state of the art?

The answer to this question comes in pieces spread throughout experiment 1, experiment 2, and the behavioural analysis section. Every compared algorithm reacted to resource placement configurations in similar ways, but some had more extreme reactions than others. GNP becomes more reliant on directional sensors as resources become more clustered, and suffers a large performance loss compared to that of the MPFA. The large difference in loss could be caused by a number of factors, including the fact that the MPFA's top performance isn't really visible in the experiments due to it managing to collect 100% of resources.

RQ3 - How does GNP scale with swarm size compared to the state of the art?

All algorithms scale similarly in the random resource placement configuration, but differences appear in the semi-clustered and clustered configurations. Notably, the MPFA scales much better than its competition, but GNP scales much better than both NEAT and GIN, especially in the clustered configuration, suggesting that GNP is better at evolving smart pheromone usage than its competition.

RQ4 - Which nodes are important for GNP in the foraging domain?

Based on node usage heatmaps and behavioural analysis, all nodes are important to some degree. The front and back sensing judgement nodes however, seem less important. Additionally, the processing node for setting speed is not used much, indicating that one does not need many copies of it.

RQ5 - How does hybridizing GNP with NEAT affect performance?

It worsens performance, at least when taking the approach to hybridization that was done in this thesis. For theories regarding why this is, see the behavioural analysis section (5.9).

RQ6 - How does node usage in NEAT-enhanced GNP differ from that of normal GNP?

When NEAT-nodes are included, node usage becomes less predictable. Surprisingly, this is not only because of NEAT-nodes taking over the jobs of other nodes, as many nodes increase in usage when compared to GNP. As for NEAT-nodes themselves, they are not used much, at least by the best individuals in the population.

Goals

The first goal of this thesis, investigating GNP's potential in evolutionary robotics, was a sounding success. It's performance in both search and pheromone communication has been clarified to some degree, as well as the relative scaling of GNP and other algorithms. The future work section will elaborate on many potential improvements to both the simulated environment and GNP itself, and given the results of this thesis, GNP is definitely worthy of further research.

As for the second goal, hybridizing GNP with NEAT, at least in the way it was done in this thesis, did not improve GNP at all. Despite the poor results, a lot of information was collected from the hybrid, contributing to a deeper understanding of GNP, and highlighting important considerations when designing improvements to GNP. The second goal was to investigate a potential improvement, wording which allows for labeling the goal "successfully reached", but how successful is debatable, as NEAT-nodes added too much chaos to properly analyze.

6.2 Future Work

This work features extensive experiments, resulting in both novel knowledge and the reinforcement of old knowledge. The experiments also highlight many weaknesses in the model and directions for further research, will will be briefly listed here. Primarily, any method that can import some of the MPFA's strengths to GNP or other neuroevolution methods are of interest, like coordinate-based control instead of using direction + rotation as a steering mechanism. Another major advantage in the MPFA's design is its smart pheromone usage. Digital solutions aren't restricted by the same things that ants are restricted by, and because of that, there's no reason not to take advantage of digitalization by using MPFA style pheromones. On the topic of MPFA and pheromones, evolving pheromones might also be positive, seeing as that's normally a part of the MPFA.

Another problem with the model was that the foraging arena was too simple for a proper evaluation of the MPFA in many of the experimental setups, as it was already at, or too close to 100% resources collected. This can be solved by either increasing the size or the arena, lowering agent movement speed, or increasing the number of resources that are present in it. Making any of those changes would also allow more swarm sizes to be tested, allowing for better insight into algorithm scaling. On the topic of algorithms not being able to show their peak performance, novelty search requires a more thorough investigation, as it should be able to perform better than what was seen in this thesis, based on its performances in other literature (see 3.3).

As for GNP, the relationship between time costs and performance is still unknown, and merits investigation. The GNP implementation designed for this paper could also be improved further, so importing concepts like automatic network sizing from the state of the art could lead to even better performance. One of GNP's strengths is that it's analyzable, but the analysis methods used in this thesis are just touching the surface of what's possible. In particular, one should be able to generate 3D models depicting evolved networks over multiple generations, which would go a long way in telling how GNP evolves its connections over time.

This thesis's method for integrating NEAT-nodes into GNP networks resulted in worse performance, but that doesn't mean all attempts at a hybrid between the two will end the same way. Outside of fundamentally different approaches, there are many potential improvements that can be made to improve GIN's performance, like using simplified operators for NEAT-evolution, changing the NEAT-evolver to work better with GIN, partly freezing GNP evolution to allow NEAT-nodes to bootstrap, using NEAT-nodes as judgement nodes instead of processing nodes, replacing absolutely all nodes with NEAT-nodes, or simply using other algorithms than NEAT.

6.3 Conclusion

To conclude, GNP seems to have a lot of potential in the world of evolutionary robotics and deserves to be the subject of further research. Further refinements to both foraging setup and the GNP model can lead to improvements, and may even lead to GNP surpassing the state of the art some time in the future. A novel hybridization between GNP and NEAT dubbed GIN was introduced, showcasing better performance than NEAT, but worse performance than GNP, the reasons for which have been analyzed to a degree. Rudimentary analysis of GNP and GIN was performed, resulting in knowledge regarding GNP's strengths and weaknesses as well as considerations to take when designing improvements to GNP have been revealed, laying foundations for further research. There is certainly much more to investigate in both the world of evolution based foraging approaches and that of GNP, and it is our sincere hopes that this thesis inspires more researchers join the effort.

Bibliography

Alfeo, A. L., Ferrer, E. C., Carrillo, Y. L., Grignard, A., Pastor, L. A., Sleeper, D. T., Cimino, M. G., Lepri, B., Vaglini, G., Larson, K., Dorigo, M., and Pentland, A. S. (2019). Urban swarms: A new approach for autonomous waste management. *Proceedings - IEEE International Conference on Robotics and Automation*, 2019-May:4233–4240.

Balaprakash, P., Birattari, M., and Stützle, T. (2007). Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *International workshop on hybrid metaheuristics*, pages 108–122. Springer.

Beni, G. (1988). Concept of cellular robotic system. pages 57–62.

Beverly, B., Mclendon, H., Nacu, S., Holmes, S., and Gordon, D. (2009). How site fidelity leads to individual differences in the foraging activity of harvester ants. *Behavioral Ecology*, 20:633–638.

Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K., et al. (2002). A racing algorithm for configuring metaheuristics. In *Gecco*, volume 2.

Bredèche, N., Montanier, J., Weel, B., and Haasdijk, E. (2013). Roborobo! a fast robot simulator for swarm and collective robotics. *CoRR*, abs/1304.2888.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47(1):139 – 159.

Chapman, C., Chapman, L., and McLaughlin, R. (1989). Multiple central place foraging by spider monkeys: Travel consequences of using many sleeping sites. *Oecologia*, 79:506–511.

Chen, Y., Mabu, S., Shimada, K., and Hirasawa, K. (2009). A genetic network programming with learning approach for enhanced stock trading model. *Expert Systems with Applications*, 36(10):12537 – 12546.

Del Ser, J., Osaba, E., Molina, D., Yang, X. S., Salcedo-Sanz, S., Camacho, D., Das, S., Suganthan, P. N., Coello Coello, C. A., and Herrera, F. (2019). Bio-inspired computation: Where we stand and what's next. *Swarm and Evolutionary Computation*, 48(April):220–250.

Efremov, M. A. and Kholod, I. I. (2020). Swarm Robotics Foraging Approaches. *Proceedings of the 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, EIConRus 2020*, pages 299–304.

Ericksen, J., Moses, M., and Forrest, S. (2018). Automatically evolving a general controller for robot swarms. *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017 - Proceedings*, 2018-January:1–8.

Feng, G. (2006). A survey on analysis and design of model-based fuzzy control systems. *IEEE Transactions on Fuzzy Systems*, 14(5):676–697.

Ferrante, E., Turgut, A. E., Duéñez-Guzmán, E., Dorigo, M., and Wenseleers, T. (2015). Evolution of Self-Organized Task Specialization in Robot Swarms. *PLoS Computational Biology*, 11(8):1–21.

Fricke, G. M., Hecker, J. P., Griego, A. D., Tran, L. T., and Moses, M. E. (2016). A distributed deterministic spiral search algorithm for swarms. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4430–4436.

Gomes, J. and Christensen, A. L. (2013). Generic behaviour similarity measures for evolutionary swarm robotics. *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference*, (April):199–206.

Hecker, J., Stolleis, K., Swenson, B., Letendre, K., and Moses, M. (2013). Evolving Error Tolerance in Biologically-Inspired iAnt Robots. pages 1025–1032.

Hecker, J. P. and Moses, M. E. (2013). An evolutionary approach for robust adaptation of robot behavior to sensor error. *GECCO 2013 - Proceedings of the 2013 Genetic and Evolutionary Computation Conference Companion*, pages 1437–1444.

Hecker, J. P. and Moses, M. E. (2015). Beyond pheromones: evolving error-tolerant, flexible, and scalable ant-inspired robot swarms. *Swarm Intelligence*, 9(1):43–70.

Hiraga, M., Wei, Y., and Ohkura, K. (2019). Evolving Collective Cognition of Robotic Swarms in the Foraging Task with Poison. *2019 IEEE Congress on Evolutionary Computation, CEC 2019 - Proceedings*, pages 3205–3212.

Hirasawa, K., Okubo, M., Katagiri, H., Hu, J., Murata, J., and Discov, P. A. (2001). Comparison between Genetic Network Programming(GNP) anid Genetic Programming(GP). *IEEE.*

Hoff, N., Wood, R., and Nagpal, R. (2013). *Distributed Colony-Level Algorithm Switching for Robot Swarm Foraging*, pages 417–430. Springer Berlin Heidelberg, Berlin, Heidelberg.

Holland, J. H. (1962). Outline for a Logical Theory of Adaptive Systems. *Journal of the ACM (JACM)*, 9(3):297–314.

Holland John, H. (1975). Adaptation in natural and artificial systems. *Ann Arbor: University of Michigan Press.*

Igor Chuxlancev (2015). Antbridge crossing. `https://commons.wikimedia.org/wiki/File:AntBridge_Crossing_08.jpg`. [Online; accessed March, 2021].

Isaacs, J. T., Dolan-Stern, N., Getzinger, M., Warner, E., Venegas, A., and Sanchez, A. (2020). Central place foraging: Delivery lanes, recruitment and site fidelity. *2020 IEEE International Conference on Autonomous Robot Systems and Competitions, ICARSC 2020*, pages 319–324.

Katagiri, H., Hirasawa, K., and Hu, J. (2000). Genetic network programming - application to intelligent agents. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 5:3829–3834.

Katagiri, H., Hirasawa, K., Jinglu Hu, and Murata, J. (2003). Comparing some graph crossover in genetic network programming. pages 1263–1268.

Lehman, J. and Stanley, K. O. (2008). Exploiting open-endedness to solve problems through the search for novelty. *Artificial Life XI: Proceedings of the 11th International Conference on the Simulation and Synthesis of Living Systems, ALIFE 2008*, pages 329–336.

Lehman, J. and Stanley, K. O. (2011). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–222.

Li, B., Li, X., Mabu, S., and Hirasawa, K. (2011). Variable size genetic network programming with binomial distribution. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 973–980.

Li, B., Yu, S., and Hirasawa, K. (2013). Usage of frequently used node in variable size genetic network programming. *Proceedings - 2013 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2013*, pages 174–179.

Li, X., He, W., and Hirasawa, K. (2014). Adaptive genetic network programming. *Proceedings of the 2014 IEEE Congress on Evolutionary Computation, CEC 2014*, pages 1808–1815.

Li, X., Yang, G., and Hirasawa, K. (2015). Evolving directed graphs with artificial bee colony algorithm. *International Conference on Intelligent Systems Design and Applications, ISDA*, 2015-January:89–94.

Li, X., Yang, H., and Yang, M. (2018). Revisiting Genetic Network Programming (GNP): Towards the Simplified Genetic Operators. *IEEE Access*, 6:43274–43289.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., and Birattari, M. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

Lu, Q., Hecker, J., and Moses, M. (2016a). The mpfa: A multiple-place foraging algorithm for biologically-inspired robot swarms. pages 3815–3821.

Lu, Q., Hecker, J. P., and Moses, M. E. (2016b). The MPFA: A multiple-place foraging algorithm for biologically-inspired robot swarms. *IEEE International Conference on Intelligent Robots and Systems*, 2016-November:3815–3821.

Lu, Q., Hecker, J. P., and Moses, M. E. (2018). Multiple-place swarm foraging with dynamic depots. *Autonomous Robots*, 42(4):909–926.

Mabu, S. (2007). A Graph-Based Evolutionary Algorithm Genetic Network Programming (GNP) and Its Extension Using Reinforcement Learning. *Evolutionary Computation*, 15(3):369–398.

Mabu, S., Hatakeyamay, H., Thu, M. T., Hirasawa, K., and Hu, J. (2006). Genetic network programming with reinforcement learning and its application to making mobile robot behavior. *IEEJ Transactions on Electronics, Information and Systems*, 126(8):1009–1015.

Maron, O. and Moore, A. W. (1997). The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11(1):193–225.

MERCER, R. E. and SAMPSON, J. R. (1978). ADAPTIVE SEARCH USING A REPRODUCTIVE META-PLAN. *Kybernetes*, 7(3):215–228.

Mouret, J. B. and Doncieux, S. (2012). Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary Computation*, 20(1):91–133.

Murata, T., Nakamura, T., and Nagamine, S. (2005). Performance of genetic network programming for learning agents on perceptual aliasing problem. *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, 3:2317–2322.

Murata, T. and Okada, D. (2006). Using genetic network programming to get comprehensible control rules for real robots. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1983–1988.

Nguyen, P. T. H. and Sudholt, D. (2020). Memetic algorithms outperform evolutionary algorithms in multimodal optimisation. *Artificial Intelligence*, 287:103345.

Ratnieks, F. L. and Anderson, C. (1999). Task partitioning in insect societies. *Insectes Sociaux*, 46(2):95–108.

Ray Norris (2005). Raynorris termite cathedral mounds. `https://commons.wikimedia.org/wiki/File:RayNorris_termite_cathedral_mounds.jpg`. [Online; accessed March, 2021].

Robots, M. (1996). Evolving Mobile Robots in Simulated and Real Environments. 434(1995):417–434.

Roger, A. and Jordan, J. (2015). Evolutionary networks for multi-behavioural robot control.

Sendari, S., Mabu, S., and Hirasawa, K. (2011). Fuzzy genetic network programming with reinforcement learning for mobile robot navigation. *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, pages 2243–2248.

Shimada, K., Hirasawa, K., and Jinglu Hu (2005). Genetic network programming with acquisition mechanisms of association rules in dense database. In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, volume 2, pages 47–54.

Silva, F., Duarte, M., Correia, L., Oliveira, S. M., and Christensen, A. L. (2016). Open issues in evolutionary robotics. *Evolutionary Computation*, 24(2):205–236.

Sims, K. (1995). Behavior by Competition. *Artificial Life*, 372:353–372.

Själander, M., Jahre, M., Tufte, G., and Reissmann, N. (2019). EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Terrio, M. D. and Heywood, M. I. (2002). Directing crossover for reduction of bloat in gp. In *IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No.02CH37373)*, volume 2, pages 1111–1115 vol.2.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.

Wang, W., Reyes, N. H., Barczak, A. L. C., Susnjak, T., and Sincak, P. (2015). Multi-behaviour robot control using genetic network programming with fuzzy reinforcement learning. In Kim, J.-H., Yang, W., Jo, J., Sincak, P., and Myung, H., editors, *Robot Intelligence Technology and Applications 3*, pages 151–158, Cham. Springer International Publishing.

Winfield, A. F. (2009). Towards an engineering science of robot foraging. *Distributed Autonomous Robotic Systems 8*, (3):185–192.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Yu, L., Mabu, S., Zhou, J., Shimada, K., Ye, F., Hirasawa, K., and Markon, S. (2008). Double-deck elevator system using genetic network programming with genetic operators based on pheromone information. *GECCO'08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation 2008*, pages 2239–2244.

Yu, L., Zhou, J., Mabu, S., Hirasawa, K., Hu, J., and Markon, S. (2007). Elevator group control system using genetic network programming with ACO considering transitions. *SICE*, pages 1330–1336.

Zhou, H., Wei, W., Mainali, M., Shimada, K., Mabu, S., and Hirasawa, K. (2008). Class association rules mining with time series and its application to traffic prediction.

Şahin, E. (2005). Swarm robotics: From sources of inspiration to domains of application. *Lecture Notes in Computer Science*, 3342:10–20.

Appendices