Evaluating the effectiveness and applicability of Novelty Search and other Quality-Diversity algorithms compared to niching

Master Thesis - ICA-4145275

Midas Schonewille

Utrecht University - Game and Media Technology

Supervisor: Dirk Thierens

Utrecht University - Intelligent Systems

September 3, 2020

Abstract

Novelty Search is a promising new evolutionary algorithm, which claims to outperform traditional evolutionary algorithms in some cases. The idea is that sometimes, pursuing the objective, like in traditional evolutionary algorithms, may prevent the objective from being reached. In these cases, it might be better to explore solutions that are inherently different than previous ones instead of solutions that have a higher fitness value. Imagine walking through a maze towards a goal, but first having to walk away from it to eventually reach it. The question is how effective this method is compared to niching algorithms, another methods that tries to increase both quality and diversity. In this paper we will subject novelty search and niching algorithms to the classic maze experiment, in order to find out exactly how effective novelty search is when compared to niching and in what cases.

1 Introduction

In most cases when finding a solution for a computer science problem, you are only interested in the single most optimal solution. However, it might be desirable to try and find a collection of solutions that all perform reasonably well. Looking for multiple locally optimal solutions is called Multimodal Optimisation

and is most often solved by using an Evolutionary Algorithm. This is because in these algorithms, a population of possible solutions is already being generated, so it makes sense to also use this data in the output of the algorithm. This can be useful when searching in a model of an actual, more complicated application. This collection of solutions can then be manually filtered and tested afterwards or used as a whole.

When trying to find high quality solutions in a search space, it is important to consider a diverse array of solutions. This is because, without prior knowledge about the search space, one does not know where the optimal solutions lie within this space. Various niching techniques have been developed to ensure multimodal optimisation and they have excelled in their simplicity. While its primary focus is still the optimisation of a fitness function, rules are put in place which prevent the population from converging and losing all its diversity.

However, when just focusing on the quality of the solution, one might get stuck exploring only local optima. The Quality-Diversity algorithms aim to combat this problem by focusing on both quality and diversity in their search of solutions, focusing primarily on diversity. The idea is that considering a wide range of solutions is just as important as chasing high quality solutions. [1, 7]

2 Research Question

Novelty Search is an algorithm that tries to accomplish something similar to niching, but the way in which it does is different from the niching algorithms that have been developed up until this point. It tries to find multiple optima in multimodal landscapes by diversity alone. Lehman and Stanley [5] state that it outperforms traditional evolutionary algorithms this way, but niching algorithms are also very adept at exploring multimodal landscapes. This raises the following question for us:

What are the characteristics of a problem that help us determine if it is better solved by niching or QD algorithms?

We will answer this question in this paper, which is laid out in the following way. In section 3 we will discuss some preliminaries. In sections 4 through 8 we will introduce and compare the relevant algorithms. The method used for comparison will be explained in section 9. From section 11 to section 15 results will be presented and discussed. Lastly, a conclusion will be drawn in section 16.

3 Preliminaries

For this paper, we will assume knowledge of evolutionary algorithms and their workings. The following are some definitions which we will use throughout this study. The collection that represents the population will be denoted by P and its size as N. The parent of new offspring is p and o is the offspring. n is the dimensionality of the problem and thus also the size of the individuals.

We will now define a basic Evolutionary Algorithm which we will use as a basis of all other algorithms discussed. A standard EA might work as follows. Initialize a population P with N individuals chosen uniformly at random. Then, while the stopping criterion is not met, we create new individuals. For some techniques, we can use recombination. With recombination we select two individuals p_1 and p_2 at random. We then create an individual o where each bit has an equal chance of being from p_1 or p_2. If for a technique it is desirable to have only one parent p per offspring o, we can just choose an individual p at random and copy it to create an individual o. After the creation of o we mutate it by flipping each bit with a probability of $1/n$ and calculate its fitness. Lastly we replace an individual with the worst fitness from P if o performs at least as good.

4 Finding multiple solutions

The search for multiple solutions has been a challenge for a long time in computer science. Sometimes one is not looking for a single solution to a problem. This can be the case in real-world applications of solutions for example, where there are always real-world errors to keep in mind. This means that the best solution on paper might not be the best in practice. To combat this, you can generate multiple good solutions and try them out to see which one works best in practice. One might also just want to make a collection of possible actions a system can take and use this collection as a whole to, for example, control a robot.

Multimodal function optimization was one of the first ways to tackle this problem [7]. This technique would find local optima of a multidimensional function. However, one might not be interested in only the optima. The goal might be to find a diverse set of solutions instead of only the best ones. Especially when there is no notion of good or bad, it makes sense to instead focus on improving diversity within a set of solutions.

5 Niching

Niching is one of the ways in which multimodal optimisation can be accomplished. When finding solutions in a multimodal landscape, the base EA will quickly converge to a single optimum, ignoring the fact that a landscape might have multiple optima. To find multiple optima, multiple diversity-preserving mechanism have been created, which try to find multiple optima simultaneously. This technique is called niching, or the creating of niches within the population. Because there are so many niching techniques, we will have to select a few of them to use as comparison material in this study. To find out which niching algorithms are the most effective, we will now enumerate several of them and highlight their properties [4].

1. No Genotype Duplicates (NGD) and No Fitness Duplicates (NFD) will

disallow o to enter P if it has the same genotype or the same fitness respectively as an individual already present. However, these mechanism do not prove effective to keep enough diversity in the population [11].

2. Fitness Sharing derates the original fitness of an individual by an amount that represents its similarity to other members of the population. There are two versions, population-based (PFS) and individual based (FS). PFS considers every subset of size N of the parents and offspring, calculates the fitness of all the individuals within this subset using their similarity to other members of this subset and selects the subset with the highest fitness as the next P. This is, however, computationally a very expensive approach and not feasible for problems with larger dimensions and population sizes. FS, on the other hand, calculates the similarity of an individual to the rest based on the entire population and thus one does not have to calculate the fitness of every subset. This is computationally a lot less expensive, but does not perform as well [9, 3].

3. When you use crowding, you let o only compete with p. There are two types of crowding. Probabalistic Crowding (PC) chooses if o or p gets added to the next generation with a probability proportional to their fitnesses, while Deterministic Crowding (DC) just chooses the one with higher fitness, giving priority to the o in the case of a tie. PC does not perform very well, because o will typically differ very little from p and it is these tiny changes that add up and drive evolution. If the fitnesses do not differ by much, the choice basically gets random. DC, however, capitalizes on these tiny changes and performs very well. It also preserves the diversity in a population, because new offspring only competes with its parent and thus the existence of multiple niches is supported [3].

4. In Restricted Tournament Selection, o only competes with a certain random sample from P. Of this random sample of a certain size w, it is compared with the one with the lowest distance from it and replaces it if it is at least as good. This ensures that individuals will only replace others that are relatively similar to it and thus having the option to create multiple niches within the population. It performs very well [12].

5. When applying Clearing, first o gets added to P. Then, P gets iterated over from the highest fitness to the lowest. Every individual encountered will be called a winner of its niche. Then, all but κ individuals that have a distance lower than σ to the winner will be removed from the collection. This makes sure that only the k best individuals per niche are preserved in P [10].

6 Quality-Diversity algorithms

Quality-Diversity (QD) algorithms are a class of algorithms that create a collection of solutions instead of just a single one. This collection should contain

a diverse group of high quality solutions. QD algorithms are characterized by 3 features: a Container, a Selection Operator and the considered value to optimize. In this paper, when we mention QD algorithms, we really mean the subset QD algorithms that focus on novelty instead of fitness making use of a behaviour metric, as these are the more interesting cases to consider. Novelty Search is one such algorithm.

6.1 The Container

The Container is used to store a diverse collection of all the best solutions encountered so far. This is also the collection that will be given as the output of the algorithm. During the algorithm, each new solution is considered for entry in the Container. Generally, it is entered if the container does not contain a solution already similar to it according to a distance metric, or if it has a higher fitness value than the solution similar to it that is already in the Container. The Container is also used to determine the novelty of newly created solutions, comparing its distance to its nearest neighbours.

6.2 The Selection Operator

The Selection Operator determines how the next batch in the population is generated. In general, the current population and the Container are used to generate the next batch. That depends, however, on the exact operator you are using.

6.3 The considered value

Some Selection Operators select on a specific value. Traditionally, fitness is the value being selected on, but new values like Novelty and Curiosity are now being introduced to select on. There are also Selection Operators that do not select on a specific value whatsoever.

6.4 The Algorithm

QD algorithms start with a random initialization of the population and an empty container. In each iteration, the following 3 steps are taken:

1. A set of parents is selected from the population and/or container, depending on the type of selection you are using.
2. The new population is generated by randomly varying the parents, for example using crossover and/or mutation.
3. Each newly created individual is potentially added to the container, dependent on the individuals already present.

These steps can be seen in Algorithm 1.

Algorithm 1 Quality-Diversity

```
Population ← RANDOM()
Container ← ∅
while Some stopping criterion is not met do
    Parents ← SELECTION(Population, Container)
    Population ← VARIATION(Parents)
    for all p ∈ Population do
        Container.ADD(p)
    end for
end while
```

6.5 Novelty Search

Novelty Search is a genetic algorithm originally devised by Lehman and Stanley [5], which focuses solely on the diversity of solutions to eventually find ones with higher fitness. The idea is that, if you just keep creating solutions that perform differently from all previous solutions, at some point you have to reach solutions of a higher quality. Novelty Search works by keeping track of a Novelty Archive. This is a collection of the most diverse solutions found which gets updated as new solutions are being explored. When looking for new solutions, their diversity gets compared against this archive to determine if it is a novel solution. This is especially effective in environments where there are multiple different solutions that have the same fitness. In the case of a standard EA, the population will converge to a single solution, but it might be that the best solution is another one. How Novelty Search distinguishes itself from niching is that it uses a whole new metric based on the behaviour of a solution to determine its similarity to others. This is based on the phenotype, and thus only indirectly on the genotype, of that solution. It is the diversity of this behaviour that gets optimized within a population in the hope that novel behaviour eventually leads to high fitness. The behaviour of an individual has to be defined on a problem-by-problem basis. It is a vector of numbers that describe how a certain individual performs the problem task. For example, if one were to teach a robot to walk, the behaviour might be the collection of coordinates of a certain part of the robot taken every second. A collection is kept of all behaviours that have occurred and if a new behaviour is located in a sufficiently sparse area of the behaviour space, it is added to this collection. The sparseness ρ at a certain point x is given by the average distance to the k nearest neighbours:

$$\rho(x) = \frac{1}{k} \sum_{i=0}^{k} dist(x, \mu_i)$$

Where μ_i is the i-th nearest neighbour to x. If this sparseness value is higher than a certain threshold value, the new behaviour is considered novel enough and is added to the collection. This allows for a lot more behaviours to be explored, even if they might not be optimal. At first, most of the behaviours will be very simple and of low fitness, but after the most simple ones have been explored, the incentive to find novel behaviour will encourage new behaviours

to explore more complex ones. Eventually, the idea is that behaviours become more and more complex and they have to include high quality ones just because the low quality ones run out.

Even though Novelty Search has been proven to work in some cases, adding some desire for quality seems like a good idea to hone in on better solutions faster. This is where Novelty Search with Local Competition (NSLC) improves upon classic Novelty Search. Where originally solutions are not added to the archive if they are too similar to an already existing solution, NSLC does add this solution if it has a higher quality than the solution that is already present in the archive. As a criterion to be added to the collection, it uses exclusive ϵ-dominance. This uses the two measures Quality and Novelty. A solution gets added if the positive difference in one measure is at least as large as the negative difference in the other measure, up to a predefined maximum of ϵ. This means that a small decrease in Quality is accepted if there is a big increase in Novelty and vice versa [1].

6.6 MAP-Elites

Another popular QD algorithm is MAP-Elites [8]. Instead of a continuous collection, this algorithm discretizes the behaviour space into a grid. This introduces a very intuitive sense of sparseness, namely, whether or not a particular cell in the grid is filled. Just like with NSLC, there is local competition, but in this case it just means that a higher quality solution will replace another one if it corresponds to the same cell in the grid.

Another difference that MAP-Elites introduces is the removal of the population. Whereas Novelty Search keeps track of a population to create new generations, MAP-Elites takes a random sample for the archive itself. This approach can also be applied to Novelty Search, which produces better results [1].

6.7 Why do they work

Now that we have explored how these algorithms work, we will take a look at why they work and produce a better understanding of the processes involved. Typically, QD algorithms are used to evolve agents, autonomous entities that act in an environment based on their information of said environment. This produces a very natural procedure to determine the behaviour of an individual, namely, to derive it from the states encountered by the agent. Agents naturally move through an environment encountering a series of different states, this series of states is characteristic for the associated individual and thus is able to define a behaviour.

QD algorithms are basically a search process in the behaviour space and the assumption is that they are doing this uniformly and randomly [2]. Classically, evolutionary algorithms are visualized as navigating through the peaks and valleys of a fitness landscape, following the gradient upwards to reach a higher fitness, hoping this will lead to the global optimum. However, this gradient

might be deceiving if it leads to a local optimum. This is why QD algorithms focus on novelty rather than fitness, but this leads to the traversal of a changing landscape. After all, adding new individuals to our container changes the novelty of previously encountered individuals.

Doncieux et al. [2] have defined the different spaces and functions related to Novelty Search, these definitions can be extended to all QD algorithms and are as follows.

- S is a state space, it contains all states an agent can be in.
- A is an action space, it contains all the actions an agent can take.
- m is a transition function, it describes how performing an action on a state changes that state.
- π is a policy, it describes what action and agent will take in a given state.
- G is a genotype space, it contains all individuals the algorithms can directly sample, with each individual defining a policy π.
- $\mathcal{B}$ is a behaviour space, it contains all possible behaviours that individuals in G can create.
- $o_{\mathcal{B}}$ is an observer function. it turns a series of states into a behaviour, oftentimes drastically reducing the dimensionality while still remaining descriptive.
- f_g is a goal function, it determines if a certain behaviour satisfies a particular goal, returning a boolean.
- $\mathcal{T}$ is a task space. A task space is a space in which a task can be naturally expressed, such that for any point in this space, you can tell if it has completed the task. $\mathcal{B}$ can be seen as a task space as f_g defines a task, with some members of that space failing at the task and others succeeding.
- $\mathcal{G}$ is a goal space, it contains all the behaviours that satisfy the goal defined by f_g.
- $\phi_{\mathcal{B}}$ is a behaviour function, it is an implicit function from G to $\mathcal{B}$ that maps every genotype to its corresponding behaviour. It is an abstraction of the whole interaction of the agent with the environment, just focusing on the resulting behaviour.

Figure 1 gives an overview of these different spaces and functions, taken from Doncieux et al. [2].

It is very important for $\mathcal{B}$ to be a task space $\mathcal{T}$ and thus be aligned with the task at hand. This means that from the descriptors of $\mathcal{B}$ you can tell if the task is completed or not. If this is the case and a member of $\mathcal{B}$ is a solution, we are actually able to tell that it is. If there are members of $\mathcal{T}$ for which f_g is True and QD algorithms are performing a uniform random search in $\mathcal{T}$, a solution

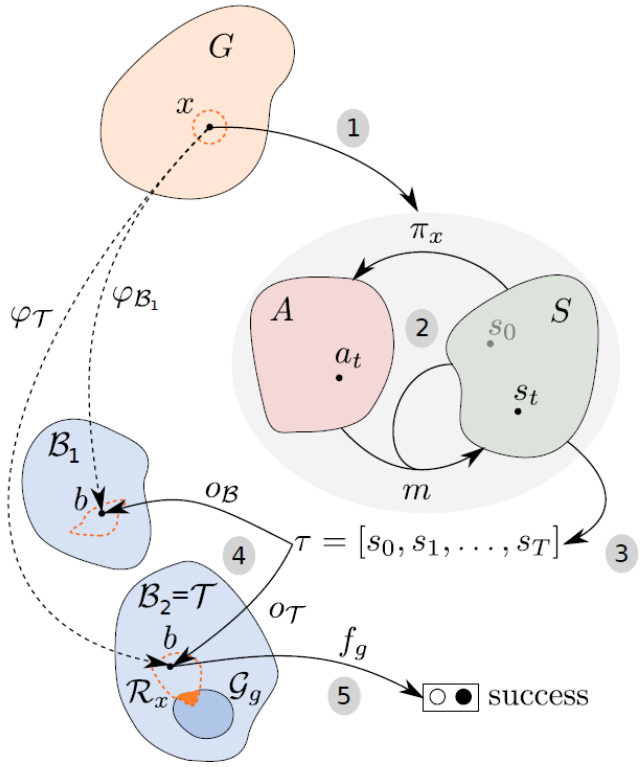

Figure 1: Overview of the different spaces and functions related to Novelty Search. The Genotype space G describes a policy π (1) that determines actions in A to apply in a given state $s \in S$ (2). The sequence of states describes the robot behaviour (3) that is projected to lower dimension behaviour spaces $\mathcal{B}_1$ or $\mathcal{B}_2$ (4). Note that $\mathcal{B}_2$ can also be called $\mathcal{T}$ as it defines a task. In this case, $\mathcal{G}_g$ describes the goal to be reached (5) and $\mathcal{R}_x$ the area of the task space $\mathcal{T}$ that is within a single mutation range from the genotype x. Figure by Doncieux et al. [2].

will eventually be found. Of course this depends greatly on the sizes of both $\mathcal{T}$ and $\mathcal{G}$. Another thing one has to take into account is the reachability of the goal from the individuals. Of course QD algorithms do not exactly perform a uniform random search. It is impossible to sample $\mathcal{B}$ directly, behaviour emerges from the genotype in a non-obvious way, so the only way to sample $\mathcal{B}$ is to try out different genotypes.

Each individual is only able to lead to a select group of individuals for the next generation. The search process will thus probably be slower than the theoretical maximum, depending on the reachability of the goal and the intermediate individuals.

6.8 When do they work

Now that we have explored how QD algorithms work, the question remains as to when they should be used. There are problems that lend themselves better to be solved by these algorithms than others, but how do we identify them?

First and foremost, QD algorithms will need some observer function $o_{\mathcal{B}}$ to define a behaviour space $\mathcal{B}$. $\mathcal{B}$ needs to be aligned with the task, but this is not at all trivial to create. One could always define $o_{\mathcal{B}}$ to just take the genotype of an individual as its behaviour, but this would not create any new space to search in. This is why QD algorithms are generally more applied to problems where an agent has to act in an environment. This gives us a more intuitive sense of defining a behaviour, namely, by looking at the series of states it has been in while attempting to execute its task. Only if a natural notion of behaviour can be defined, typically through looking at an agent's states, can QD algorithms be applied.

When a behaviour is defined, the effectiveness of the algorithm will depend on the size and reachability of $\mathcal{B}$. When $\mathcal{B}$ is too big, it will take too long to find the behaviour you are interested in. There must also be a clear way to progress through these behaviours, from the random starting population to your goal. If this path does not exist or is too long, $o_{\mathcal{B}}$ might not be set up right, or the problem might not be suitable to be solved by QD algorithms.

6.9 A Modular Framework

Cully and Demiris [1] have developed a framework in which to unify the QD algorithms. Both MAP-Elites and NSLC can be formulated as an algorithm within this framework. This framework, in general, creates a lot of different algorithms by combining different Containers, Selection methods and considered values. This allows us to easily define the QD algorithms that we will be using in this paper. They are laid out in Table 1.

Variant name	Container	Selection Op.	Considered Value	Related approach
arch_no_selection	archive	noselection	-	Random Search / Motor Babbling
arch_random	archive	random	-	-
arch_pareto	archive	Pareto	Novelty & Local Quality	-
arch_fitness	archive	Score-based	Fitness	-
arch_novelty	archive	Score-based	Novelty	MAP-Elites with Novelty [13]
arch_curiosity	archive	Score-based	Curiosity	-
arch_pop_fitness	archive	Population-based	Fitness	Traditional EA
arch_pop_novelty	archive	Population-based	Novelty	Novelty Search [5]
arch_pop_curiosity	archive	Population-based	Curiosity	-
grid_no_selection	grid	noselection	-	Random Search / Motor Babbling
grid_random	grid	random	-	MAP-Elites [8]
grid_pareto	grid	Pareto	Novelty & Local Quality	-
grid_fitness	grid	Score-based	Fitness	-
grid_novelty	grid	Score-based	Novelty	-
grid_curiosity	grid	Score-based	Curiosity	-
grid_pop_fitness	grid	Population-based	Fitness	Traditional EA
grid_pop_novelty	grid	Population-based	Novelty	-
grid_pop_curiosity	grid	Population-based	Curiosity	-
NSLC	grid	Population & archive based	Novelty & Local Quality	Novelty Search with Local Competition [6]

Table 1: Different combinations of Container, Selection Operator and considered value.

7 The difference

Because niching and QD algorithms do a lot of similar things, the question remains, what are their differences? Whereas niching performs selection on either the genotype or the fitness, novelty search does this on an intermediate metric, its behaviour.

Individuals that have differing genotypes might very well have very similar phenotypes. Especially in the case of robots being controlled by neural networks, there is a high chance for redundancy in the connections within this neural network. It is also the case that, when the robots do not have a very wide range of actions, they have a higher chance of acting alike. Even though these individual behave very similarly, their genotypes differ. Most niching techniques select individuals that are distinct in their genotype, but differing individuals might still map to the same phenotype and thus fitness. This might lead to a lot of unnecessary exploration of the genotype space.

Ideally, one would prefer to explore the entire genotype space to find the individual that they are looking for. When this can be done exhaustively, it is certain that the desired individual will be found if it exists. However, most of the time this is not feasible to do because of the sheer size of this space. This is why Novelty Search introduces a new space to search in: the behaviour space. This space is used as an intermediary space between genotype and fitness. It tries to be a numeric representation of the phenotype.

The idea is that a search will be performed in this behaviour space, which can be a lot smaller than the genotype space. Of course there is a trade-off between speed and exhaustiveness to be made here. Individuals with an already seen behaviour will not be explored, so speed will increase, but important and/or

interesting individuals might be dismissed. This is why it is important for the behaviour function to make sure that individuals with a similar behaviour metric do not differ too much in their phenotype. If they do not, the search space will be condensed without losing too much exhaustiveness.The question is, however, whether an effective behaviour function always exists in each search space. And if it does not, in what kind of situations does it exist?

One might imagine that there are a lot of situations where this function exists, after all, there is almost always a lot of redundancy in the genotype space. One solution is to make the behaviour an aggregation of all the states the robot has found itself in, however, this can very quickly become a very large space as well, especially if there are a lot of time steps. This is why the behaviour is more often defined as a subset of the traversed states, usually separated by a fixed interval. This is a natural way to shrink the size of the behaviour space while still accurately describing the phenotype. The amount of states incorporated within the behaviour can be tweaked at will, lowering it if you want more speed and raising it if you want two individuals to have a more distinct behaviour value. It must be figured out on a problem by problem basis if the right balance exists between these two factors within the behaviour function, or if more traditional niching algorithms are more efficient.

8 Time complexity analysis

Algorithms that use an Archive have a running time of at least $\mathcal{O}(NC)$ per generation where C is the size of the container and N is the size of the population. This is because every individual in the population has to be tested against every individual in the Archive to determine if it is to be added or not. When using a grid, the running time is taken down to $\mathcal{O}(N)$ per generation, because the correct grid cell can be sampled directly.

When selecting randomly from the container, this has a time complexity of $\mathcal{O}(N)$ per generation, but if selecting from the population based on novelty, the k nearest neighbours need to be found each time. When using a grid, this takes $\mathcal{O}(Nk)$ per generation, but when using an archive, this takes $\mathcal{O}(NCk)$.

The standard fitness based EA with roulette wheel selection has a running time of $\mathcal{O}(N^2)$ per generation, just like Clearing. Deterministic crowding has a running time of $\mathcal{O}(N)$ per generation and Restricted Tournament Selection does $\mathcal{O}(Nw)$ per generation.

9 The method

To get a deeper understanding of the differences between the discussed algorithms, we will conduct an experiment where we will analyse their performances and make some comparisons.

As our experimental setup we will recreate the maze experiment used by Lehman and Stanley [5] in their first paper about Novelty Search. In this

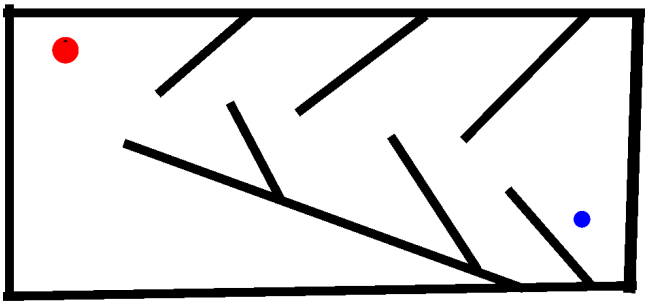

Figure 2: The maze as created by Lehman and Stanley [5]. The red circle is the robot and the blue circle is the goal. Black lines are walls.

problem, we need to find a neural network that will navigate a robot towards a goal within a maze. This neural network has ten input nodes and two output nodes. Six of the ten input nodes are range finders, one for each orthogonal direction and two at a 45 degree angle from the forward facing range finder, that indicate the distance to the nearest wall in that direction. The other four input nodes are radars that turn on when the goal is in a 90 degree slice, again one pointing in each orthogonal direction. Of the two output nodes, one is for moving the robot forwards or backwards and one is for rotating the robot left or right. In Figure 2 you can see the maze that is being used. In this figure, the red circle is the robot and the blue circle is the goal.

This neural network will be evolved using NeuroEvolution of Augmenting Topologies (NEAT) developed by Stanley and Miikkulainen [14]. This is an algorithm that starts out with simple neural structures, but can increase its complexities. This forces our algorithms to first encounter simple behaviours before encountering more complex ones. The code base was written to be able to evolve such NEAT genes.

The rest of the code was written from scratch in C# using the Unity engine. The maze was manually recreated within Unity to resemble the original one as much as possible. The modular framework by Cully and Demiris [1] was used as reference to create a system in which the container type and selection operator can be combined easily. All the niching algorithms were also written and implemented separately.

10 Determination of parameters

To make sure there is a fair comparison between all the different algorithms, we need to find the correct parameters to use for each of them. For this purpose, a preliminary test was done to determine which parameters to use in the official experiment. For this test, a wide range of parameters was tested for each

algorithm. Each combination of algorithm and parameters was run 5 times on the maze and the progression of the average and maximum fitness was traced throughout the run, after which we plotted the average of these 5 runs in Figure 3.

For the archive based QD algorithm, the tested parameter is the maximum distance two individuals may have in the container and it seems that a distance of 1 performs the best. This also happens to be the robot's diameter. For the grid based QD algorithm, the optimal value seems to be 1 as well, this value is the width of the grid's cells and is thus closely related to the parameter of the archive. For Clearing, the values of both σ and κ need to be set. Values of σ were tested between 5 and 20 and values of κ were tested between 4 and 8. $\sigma = 10$ and $\kappa = 6$ seem to work very well,as well as $\sigma = 20$ and $\kappa = 8$. For restricted tournament selection the value of w was tested between 4 and 8, $w = 8$ seemed to give the best results.

11 Maze Results

For the parameters of this experiment we will be using a feed-forward only network with a proportion of initial connections of 50%.

The following algorithms will be considered and compared to one another (see Table 1):

- `arch_random` with a threshold of 1.
- `arch_pop_novelty` with a threshold of 1 and a k value of 5.
- `grid_random` with a grid size of 1.
- `grid_pop_novelty` with a grid size of 1 and a k value of 5.
- The niching algorithm Clearing. One version with $\sigma = 10$ and $\kappa = 6$, and one version with $\sigma = 20$ and $\kappa = 8$.
- The niching algorithm Deterministic Crowding.
- The niching algorithm Restricted Tournament Selection with $w = 8$.
- A standard fitness based EA.
- A random search as a baseline.

These algorithms were chosen as a fair representation of both QD algorithms and niching algorithms. Novelty Search and MAP-Elites are both represented as well as two closely related algorithms. The chosen niching techniques are the techniques that were found to perform the best in section 5.

For this experiment we run each algorithm on the maze problem, stopping it at 200 generations or if any individual reaches the goal. We then run all the algorithms again, repeating the process for a 12 hour period. This has resulted

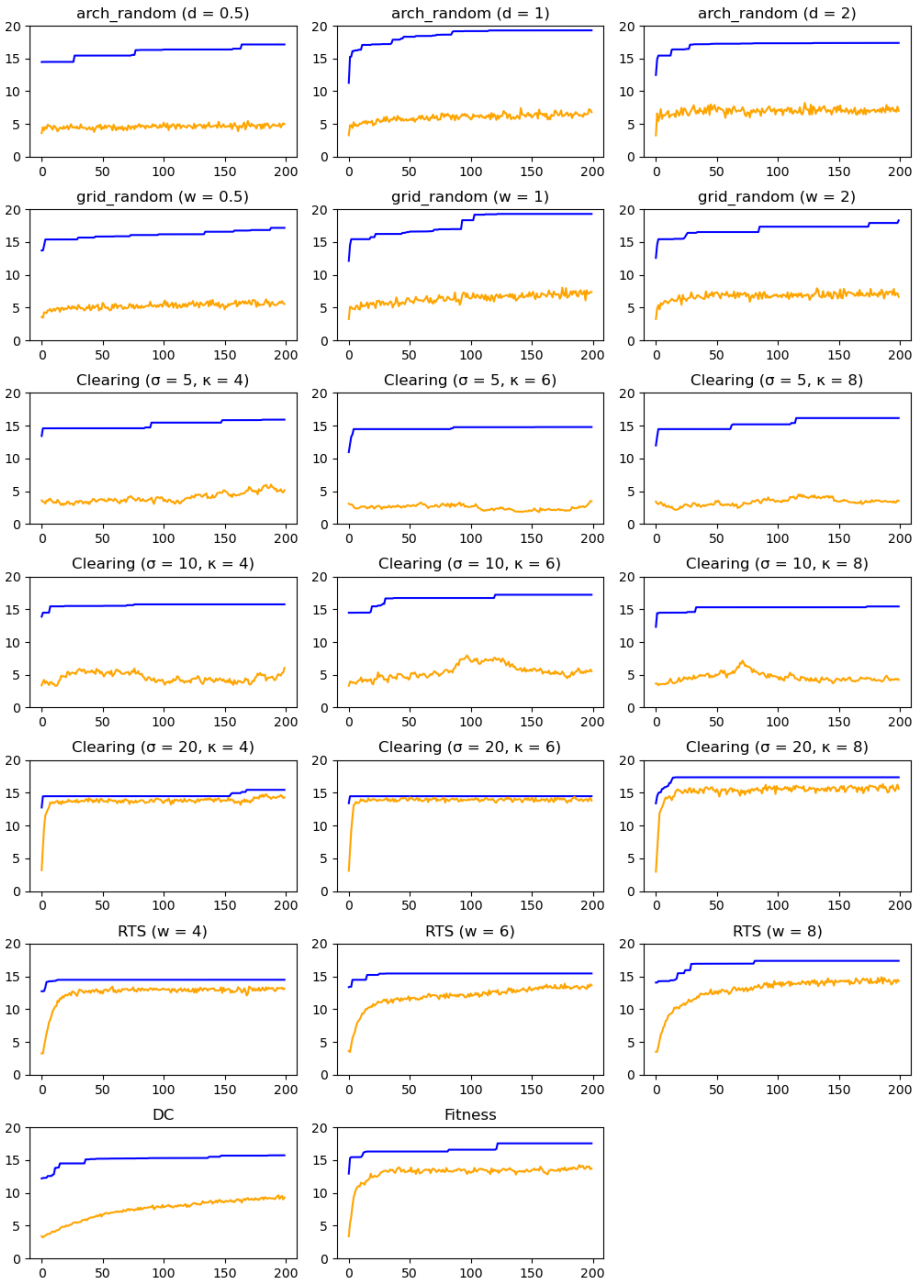

Figure 3: The average fitness (in orange) and maximum fitness (in blue) over time averaged over 5 runs for each combination of algorithm and parameters. Deterministic Crowding and a basic fitness based EA were also run, but these do not have specific parameters to set up.

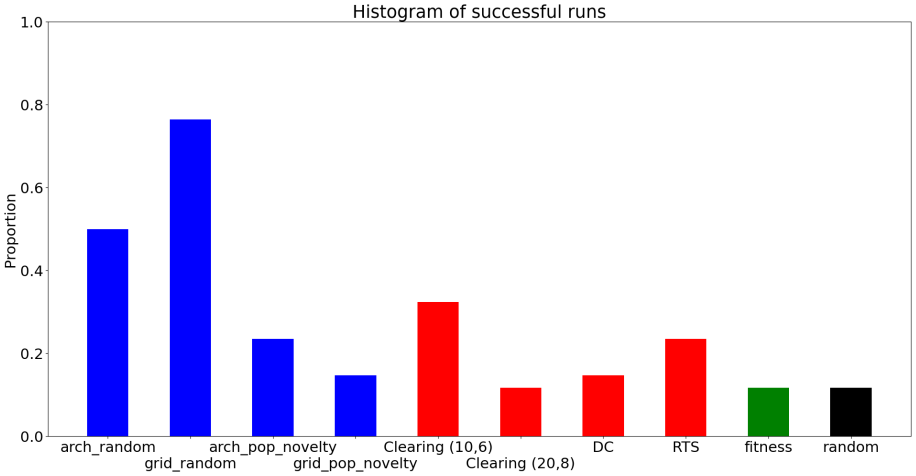

Figure 4: A histogram of the proportions of successful runs by various algorithms on the regular maze problem.

in 34 runs per algorithm. For each run we record if the goal was reached and plot the resulting proportion of successful runs. These results can be seen in Figure 4.

12 Maze Discussion

From the results we can see that `grid_random` has the best performance among the tested algorithms. `grid_random` is equivalent to MAP-Elites, which agrees with the results from Cully and Demiris [1]. After this, `arch_random` has the best performance, which is a variation on MAP-Elites where the grid is replaced with an archive. After these two QD algorithms, Clearing ($\sigma = 10$, $\kappa = 6$) actually outperforms the two population based QD algorithms. These probably perform so badly because their populations are converging to individuals that end up at the various walls of the maze. This happens because these behaviours have less neighbours and thus are more novel. The population of the Clearing algorithm tends not to converge because of the large amount of niches that gets preserved.

In general, however, QD algorithms tend to outperform niching algorithms in this experiment, with most of the niching algorithms doing barely better than the standard fitness based EA and even the fully unguided random search. Apparently, it is hard to create niches within the genotype space of NEAT neural networks. Different individuals can still inhibit fairly similar behaviour, but when focusing on the genotype space, their similarity is missed. When focusing on an intermediate behaviour space, like the QD algorithms are doing, it is possible to distinguish more distinct niches of actual different behaviour.

Another aspect to consider is the parameters that these algorithms have, which need to be set up correctly. Having a lot of parameters that are not

trivial to set up can make it very hard to apply an algorithm correctly. Clearing actually has two parameters that heavily influence its performance. These are quite abstract and not trivial to determine, so tests will be needed to determine these values. Container based algorithms, both archive- and grid-based, will need a parameter for the spacing between individuals, but this value is much less abstract and thus easier to determine without running too many tests. Novelty Search itself, however, has another parameter that determines the k nearest neighbours that are used to calculate novelty. Restricted Tournament Selection also uses a more abstract parameter, but Deterministic Crowding does not. However, Deterministic Crowding has not proven to be much better than a random search as can be seen in Figure 4.

It is remarkable to see how much better the grid container performs than the archive container when comparing `arch_random` to `grid_random`. This might be because each individual within a cell represents its own niche, and each niche is able to improve towards the goal within this cell. On the other hand, if an individual within an archive improves slightly towards the goal, chances are high that it has also come close to another individual within the archive, and thus will not be entered.

Another advantage that grid-based algorithms have is their faster running times. Whereas archive-based algorithms need to compare against every one of their individuals, in a grid you only have to check against one. It is worth noting that Clearing performs better than RTS, which in turn performs better than DC, which is exactly in the order of decreasing time complexity. This would suggest that more complex and involved algorithms with a higher running time perform better, which makes sense.

13 A bigger maze

What is interesting about this particular maze problem is that the starting location is in the top left and the goal is in the bottom right. This means that, at least at the start, if an individual exhibits novel behaviour, it will always be moving towards the goal. This gives the QD algorithms an unfair edge above other algorithms, which are focusing on the goal directly. To do a fair comparison between these algorithms, we have constructed a second maze. We have mirrored the original maze horizontally and vertically to create a bigger maze with the starting location in the middle and the goal still in the bottom right corner. This creates an environment where an increase in diversity does not necessarily leads to an increase in fitness. It will be interesting to see how QD algorithms deal with this change in landscape.

14 Big Maze Results

For this second experiment, we have kept all parameters the same, except the maximum amount of generations. This is doubled to 400, to accommodate for

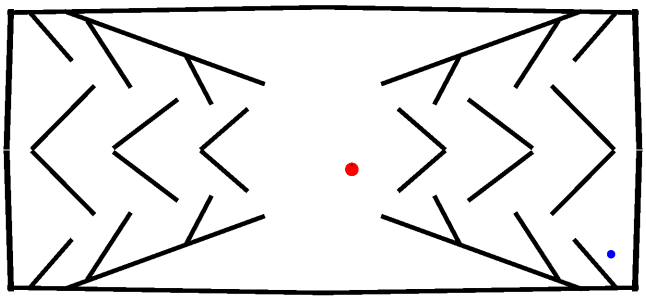

Figure 5: The bigger maze, mirrored both horizontally and vertically. The red circle is the robot and the blue circle is the goal. Black lines are walls.

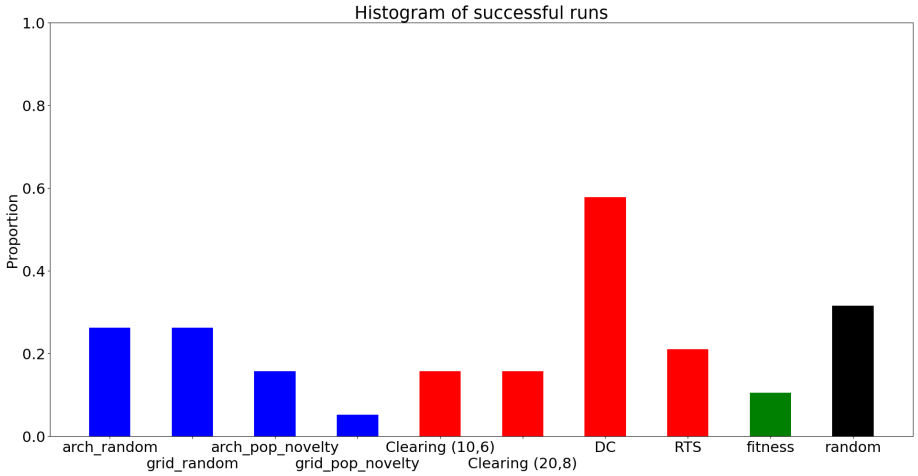

Figure 6: A histogram of the proportions of successful runs by various algorithms on the big maze problem.

the bigger size. The bigger maze is displayed in Figure 5.

After running this experiment for a 12 hour period, each algorithm was run 19 times and the results can be found in Figure 6.

15 Big Maze Discussion

In the big maze, Deterministic Crowding has outperformed the rest of the algorithms. The addition of more space, and thus increasing $\mathcal{B}$, has hampered the performance of the QD algorithms. This confirms the idea that creating a good behaviour space is very important. This is especially evident in scenarios that provide a natural definition of behaviour where a change in behaviour typically indicates an increase in fitness, such as moving away from the top left starting

position in the maze towards the bottom right. It is in these scenarios that QD algorithms can be applied successfully. In situations where there are too many directions to take in the behaviour space, QD algorithms become inefficient and not better at finding solutions than typical niching algorithms.

It appears that the conditions for QD algorithms to be applied successfully are quite sensitive and perturbing one of them, like in this case the maze size, can have a profound impact on its performance. Sensitive conditions like these are generally undesirable in algorithms, as implementation can be very hard. Another downside is that when applied to problems that are similar to ones that QD algorithms performed well on, there will not be a guaranteed success, as demonstrated with these two maze problems.

16 Conclusion

In this paper we have compared Quality-Diversity algorithms, and in particular Novelty Search, to various niching algorithms. Whereas niching algorithms perform diversification on the genotype space, QD algorithms do this on an intermediate space, the behaviour space. This can be both helpful and detrimental, depending on the features of this space. When this space is not too big and neatly aligned with the goal, it can outperform niching algorithms, however it will perform worse if the space is too big or unaligned.

In the case of the maze experiment, QD algorithms perform well if the starting and finishing positions are in opposite corners. This creates an implicit drive towards the goal and thus more fit individuals, even though QD algorithms explicitly do not focus on fitness. This happens because these algorithms search for novel behaviour and, especially in the beginning, novel behaviour can only be found towards the goal. When QD algorithms are faced with our altered maze experiment, with the starting position in the middle, it severely drops in effectiveness. In the case of this altered maze, other directions away from the goal are also novel, so there is no implicit drive for fitness anymore. In this case, niching algorithms perform better, because they focus primarily on fitness and thus easily dismiss the other "wrong" directions.

When considering to use a QD algorithm, you have to make sure that your behaviour space facilitates traversal to the desired state. This has to be done by not making it too big, otherwise there will be too many behaviours to explore and it will take too long to reach ones that actually work towards the goal. Another thing to consider is the alignment of your space. You should be able to tell to what extent an individual has completed the desired task from looking at its behaviour. Only in this way will effective individuals be found, because the behaviour space is the one being explored after all. Lastly, it is important to consider the reachability of your behaviour space. There should be a possible evolutionary path from the behaviour of any random starting individual to the behaviour of an individual that fulfills the task. This path should be along other behaviours that are reachable with a single mutation. In other words, the behavior space needs to be completely connected.

When all these conditions are met, QD algorithms can indeed be a very effective way to solve problems. However, these are a very delicate set of conditions and when they are not met perfectly, basic niching algorithms are often simply better. The question remains if QD algorithms can actually be applied to real-world problems, or if their expertise remains with experiments specifically fabricated to contain the ideal set of conditions.

17 Code Base

The code base used and more specific information about the experiments for this project can be found at `https://github.com/Zupami/quality-diversity-niching`

References

[1] A. Cully and Y. Demiris. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation*, 22 (2):245–259, 2018.

[2] S. Doncieux, A. Laflaquière, and A. Coninx. Novelty search: a theoretical perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 99–106, 2019.

[3] T. Friedrich, P. S. Oliveto, D. Sudholt, and C. Witt. Analysis of diversity-preserving mechanisms for global exploration. *Evolutionary Computation*, 17(4):455–476, 2009.

[4] J. Horn, N. Nafpliotis, and D. E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Proceedings of the first IEEE conference on evolutionary computation, IEEE world congress on computational intelligence*, volume 1, pages 82–87. Citeseer, 1994.

[5] J. Lehman and K. O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[6] J. Lehman and K. O. Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218, 2011.

[7] S. C. Maree, T. Alderliesten, D. Thierens, and P. A. N. Bosman. Real-valued evolutionary multi-modal optimization driven by hill-valley clustering. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 857–864. ACM, 2018.

[8] J.-B. Mouret and J. Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[9] P. S. Oliveto, D. Sudholt, and C. Zarges. On the runtime analysis of fitness sharing mechanisms. In *International Conference on Parallel Problem Solving from Nature*, pages 932–941. Springer, 2014.

[10] E. C. Osuna and D. Sudholt. Analysis of the clearing diversity-preserving mechanism. In *Proceedings of the 14th ACM/SIGEVO Conference on Foundations of Genetic Algorithms*, pages 55–63. ACM, 2017.

[11] E. C. Osuna and D. Sudholt. Empirical analysis of diversity-preserving mechanisms on example landscapes for multimodal optimisation. In *International Conference on Parallel Problem Solving from Nature*, pages 207–219. Springer, 2018.

[12] E. C. Osuna and D. Sudholt. Runtime analysis of probabilistic crowding and restricted tournament selection for bimodal optimisation. *arXiv preprint arXiv:1803.09766*, 2018.

[13] J. K. Pugh, L. B. Soros, P. A. Szerlip, and K. O. Stanley. Confronting the challenge of quality diversity. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 967–974, 2015.

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