A Simple Bucketing Based Approach to Diversity Maintenance

Amit Benbassat
Sapir Academic College
M. P. Hof Ashkelon, Israel
amitbenb@mail.sapir.ac.il

Yuri Shafet
Ben-Gurion University of the Negev
Beer-Sheva, Israel
shafet@post.bgu.ac.il

ABSTRACT

We present an approach to diversity maintenance based on separating the population into buckets based on similarity and biasing selection to keep individuals from all buckets in the population. We look at two approaches to bucketing. The first uses a locally sensitive bucketing function on individuals. The second uses the K-Means clustering algorithms to divide the population. We focus our research on a family of deceptive problem domains which we dub Tricky Keys and analyze how the using bucketing methods changes evolutionary search results for problem instances of varying difficulty. Our results show that both bucketing by function and bucketing by clustering methods show an increase in probability of finding a good solution and in number of good solutions found.

CCS CONCEPTS

•Computing methodologies → Genetic algorithms; •Mathematics of computing → Evolutionary algorithms;

KEYWORDS

Evolutionary Algorithms, Diversity

ACM Reference format:
Amit Benbassat and Yuri Shafet. 2017. A Simple Bucketing Based Approach to Diversity Maintenance. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082528

1 INTRODUCTION

Maintaining diversity is a common goal in *Evolutionary Computation* (EC) research [2, 10, 13, 14]. It is noteworthy that though it is quite easy to generate and maintain genotypic diversity by continuously introducing new random individuals, such new individuals will unlikely be helpful in any problem space of significant size and difficulty. In some cases (e.g. [7]) it may make sense to measure diversity at a higher level such as phenotype or behavior, however that may not always be practical or useful.

Another common characteristic that *Evolutionary Algorithms* (EAs) usually have and can be relevant to diversity maintenance is fitness. When using diversity maintenance we may still be interested in results expressed by high fitness. By encouraging a more comprehensive search of the problem space diversity maintenance techniques are designed to improve results in two distinct ways.

The first improvement one expects from maintaining population diversity is quite simply to find higher fitness individuals. In a diverse population there are more opportunities to get around the deceptive areas of the problem space and find the best individuals. The second improvement is finding a more diverse set of good solutions. Some problem spaces may contain a set of several different locally optimal high quality solutions and finding as many of those solutions as possible may be desirable.

If either of these two improvements occurs we can say that not only was high diversity achieved, but it was in fact the right kind of diversity that we were looking for. We refer to population diversity that displays one or both of these fitness improvements above as *Effective Population Diversity*.

Section 2 contains a short summery of some of the previous work in diversity maintenance techniques and clustering that relates to this work. In Section 3 we present both variants of our bucketing approach. The family of problem domains that we use in our simulations is discussed in Section 4. We present our experiment setup and runtime parameters in Section 5. Our results appear in Section 6 where we examine the success of our methods in finding high fitness points. Finally in Section 7 we conclude and discuss our results.

2 RELATED WORK

A popular method of subdividing the population is the Island Model [11]. In this approach, the population is divided into subsets or *Islands*, each of them running an independent EA with occasional migration events between islands. Every island's initial population is typically chosen at random and population differences are a result of initial differences and their interactions with evolutionary dynamics.

Petrowski [12] suggested the rather crude but effective *Clearing* approach in order to coerce the EA to maintain a diverse population. Using Clearing only one dominant individual is allowed to exist in every small area of the problem space, forcing all neighboring lower fitness individuals into immediate extinction and preventing the population from devolving into tight clusters around already discovered local optima.

In work by one of the authors of this paper [1] selection was manipulated to limit the number of neighbors each individual in the population can have.

Speciation [10] is another measure used to encourage diversity by separating the population into *species* based on some criterion (usually genetic similarity, phenotype similarity, the existence of common recent ancestry or some mixture of these factors). In

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082528

the case of speciation individuals reproduce only with members of their own species, though they often still compete with members of other species during selection phase or through co-evolution. Stanley and Miikkulainen [16] chose to adapt Speciation to evolving *Artificial Neural Networks* (ANNs) in their NEAT system. This decision carried through to it successor HyperNEAT that also uses Speciation for diversity [3, 15].

In later work Lehman and Stanley [7] suggested *Novelty Search*, a method that completely replaces standard fitness with a novelty function that is calculated based on past behavior from previous iterations of the EA recorded on a special database.

2.1 K-Means

James MacQueen was the first to use the term *K-Means* [9]. The standard algorithm was first proposed by Stuart Lloyd in 1957 as a technique for pulse-code modulation, though it wasn't published outside of Bell Labs until 1982 [8]. Forgy discovered the idea independently in 1965 [4].

3 BUCKET-BASED SELECTION

Bucket-Based Selection maintains diversity by assigning each individual in the population to a one of k buckets based on some locality sensitive procedure and then applying the standard selection method (tournament selection in our case) on each bucket separately. This selection method selects an equal number of $popSize/k$ individuals from each bucket. Figure 1 contains a simplified schematic visualization of the process of bucket-based selection. Algorithm 1 shows the pseudocode for the bucket based selection phase.

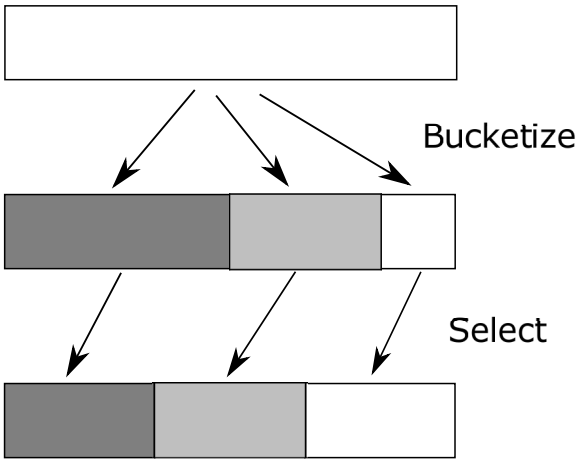

Figure 1: Schematic description of bucket-based-selection on a population of size n. The population is first divided into 3 buckets of different sizes. Next selection is employed on each bucket to separately select $n/3$ parents for the parent population, resulting in a population containing the same number of individuals from each of the buckets

Algorithm 1 Bucket Selection (Population P, k)

for each individual I in P **do**
  Assign I to bucket $i \in \{1...k\}$
end for
$NewP \leftarrow \{\}$
for each bucket B in P **do**
  Use selection method to select $size(P)/k$ individuals from B and add them to $NewP$
end for

3.1 Locality Sensitive Function Bucketing

Perhaps the simplest approach to dividing the population into k buckets is to have some sort of bucketing function f which, given an individual I, assigns it to a bucket $f(I) \in \{1..k\}$. This method gives a straightforward interpretation of bucket assignment as described in Algorithm 1. The exact nature of the bucketing function is domain-dependent. This method has the potential of being very fast as all it requires is $PopSize$ calls to f. This bucketing method is static. An individual I will always be assigned to the same bucket $f(I)$. One can get around this by using a sequence $\{f_i\}$ of bucketing functions and using f_i to assign individuals to buckets on the i'th generation.

3.2 Clustering Based Bucketing

Another way to go about bucket assignment is to use a clustering algorithm beforehand. The clustering algorithm divides the population into k clusters based on a nonnegative distance function d. Individuals are then assigned a bucket based on the cluster they are in. An interesting attribute of this approach to bucketing is that it is dynamic. As the clustering algorithm runs once every generation and as its results depend upon the distribution of the current generation's individuals in the search space, bucket associations differ on successive generations.

Algorithm 2 shows the pseudocode for the K-Means algorithm. This Algorithm serves us as a fast efficient way to subdivide a set of feature vectors into k compact clusters on using a distance function as above for some predefined k. We then take the clusters in the aforementioned set as our the division of the population into k buckets.

Algorithm 2 K-Means (Population P, k)

Randomly guess a set of k mean values for the k clusters:
  $M \leftarrow \{m_1...m_k\}$
repeat
  $C \leftarrow$ A list of k empty sets
  for each individual I in P **do**
    Assign I to set i such that the distance between I and m_i is minimal
  end for
  for $i \leftarrow 1$ **to** k **do**
    $m_i \leftarrow$ The mean of items in element i of C
  end for
until There are no more changes in M
return C

4 THE TRICKYKEYS DOMAIN

In order to test our bucketing based approach to diversity maintenance we use a simple hand-crafted family black-box optimization problems which we dub *TrickyKeys* problems. This set of problems has several qualities that make it very useful for our purposes.

(1) It is relatively simple to explain, conceptualize, and implement in code.
(2) It allows for fast fitness calculations, linear or near linear in the length of the genome even for problem instances that contain a large number of local and global optima.
(3) It is designed primarily to define deceptive search problems.
(4) It is very tunable using problem parameters, allowing the experimenter to control search space size, degree of deceptiveness and number of local and global optima.
(5) It lends itself to additions and alterations with ease.

In a TrickyKeys problem instance each individual genome consists of a vector of integers. There are however several problem parameters to be set. One such parameter is a natural number b. Individual genomes in TrickyKeys are made of base b strings. Though the problem can easily be expanded to include search spaces with variable length strings, in this work we limit ourselves to genomes of constant length. A number of keys exist that have optimal fitness. A number of trick keys with locally optimal but globally suboptimal fitness also exist. If the number of trick keys ≥ 1 the problem instance is deceptive.

4.1 Fitness

The fitness function calculates a different fitness value for each one of the keys and trick keys and then returns the maximal of those values. Let f_k be the value calculated for exactly matching one of the keys and let f_t be the value calculated for exactly matching the trick key t.

$$\forall t \in Trick\_Keys, \quad f_t < f_k \tag{1}$$

In order to derive the fitness score element in respect to a trick key t of length n, we count the number of in-place matches between the individual's genome and t. We then multiply the result by a constant factor (in our experiments below this factor is 0.8). A perfect match with t gives us a fitness score of $f_t = 0.8n$ (e.g. for trick key 1111111111 the individual 1002101131 gets a fitness score of $5 \cdot 0.8 = 4$).

In order to increase deceptiveness we apply a more stringent approach to evaluating fitness according to keys. Rather than counting the number of in-place matches we find the longest in-place substring match with the key and return its length (e.g. for key 1111111111 the individual 1102111131 gets a fitness score of 4 because of the 7 matches 4 are consecutive).

4.2 Permutation and Segmentation

As defined up to this point TrickyKeys problem domains produce strong tight building blocks and fit very well with standard k-point crossover operators as well as mutation operators that make use of locality. Since we want the problem to be hard to solve we wish to do away with these building blocks. To solve this issue each TrickyKeys problem instance may be associated with a permutation π. When fitness is evaluated π is applied to the genome before compering it with the different keys. The permutation is an optional feature which we turn off due to time considerations when running simulations that do not use any genetic operators that take advantage of genome locality.

An attribute of TrickyKeys problems is that due to high problem deceptiveness it very quickly becomes very difficult to find an optimal solution as the problem size grows. A way to get around this prohibitive hardness is to segmenting the keys. Suppose we have a problem instance with length 60 genomes. We may decide to treat each of the keys and trick keys as segmented into 3 length 20 keys. The fitness function will then evaluate fitness separately for every segment and return the sum of these evaluations.

Algorithm 3 contains the pseudocode of the fitness calculation method.

Algorithm 3 Calculate Fitness ($Genome$, π $SegmentRanges$, $Keys$, $TrickKeys$, $TrickFactor$)

```
fit ← 0
PermutatedGenome ← π(Genome)
for each Range in SegmentRanges do
  subfit ← 0
  for each Key in Keys do
    temp ← Maximal length of an in-place substring of
           Key[Range] in PermutatedGenome[Range]
    subfit ← MAX(temp, subfit)
  end for
  for each Key in TrickKeys do
    temp ← Number of in-place matches of
           Key[Range] in PermutatedGenome[Range]
    subfit ← MAX(TrickFactor · temp, subfit)
  end for
  fit ← fit + subfit
end for
return fit
```

The notions of using a permutation on genome locations to decimate building blocks as well as problem segmentation was previously suggested and implemented by Goldberg et al. in their study of messy genetic algorithms [5, 6].

5 EXPERIMENTS

We run experiments in simulation sets. In each set the simulations run with the same parameters except that all the keys get randomly generated before each simulation. By randomly generating the keys for each simulation we get to explore multiple instances with similar traits (identical search space with different fitness landscapes).

We have three basic types of simulation sets. The *Normal* simulation set type consists of evolutionary runs with no bucketing. The *Func* simulation set type consists of evolutionary runs using locality sensitive function bucketing. in our experiments we tested several possible bucketing functions and discovered that one of the simplest functions we tried seems to work best. Our bucketing

Table 1: Runs using 80-bit genomes subdivided into 4 20-bit segments. In this table and in all following tables of simulation sets, each line represents results gathered from 100 consecutive identical simulations. All simulations below use a population size of 700 and run for 200 generations.

Keys	Trick Keys	Run type	# of buckets	Mean best fitness	Solution found
5^4	1	Normal	-	71.60	0
5^4	1	Func	81	76.45	16
5^4	1	K-Means	40	75.46	13

function is based on simply summing all the genes in the genome. For a base b genome of length n this function returns a number in the range $\{0, 1, ...bn\}$ for $bn + 1$ buckets. We can scale this number down to more manageable k by looking at the remainder *mod* k. The *K-Means* simulation set type consists of evolutionary runs using clustering based bucketing. We use the K-Means clustering algorithm to bucketize. The algorithm utilizes a simple binary distance function that counts and returns the number of different elements in the two genomes.

For all runs we used a personal dell laptop computer with an intel i7-4712HQ CPU 2.3GHz with 16GB of RAM. Each simulation ran on a single logical processor (out of 8) taking between 5 minutes to half an hour, depending on it's size and bucketing method used. Clustering based bucketing runs slowed down running time by a factor of between 1.5 and 10 compared with other simulations, depending on parameter setup.

5.1 Runtime Parameters

In all our simulation sets we use the following runtime parameters unless specifically stated.

- Each simulation set consists of 100 simulations.
- We use 0.8 as the factor for the fitness deceptive trick keys.
- We use tournament selection with a tournament size of 3.
- Point mutation (randomly swap one digit in the genome) with probability 0.8. No crossover.
- Population size and number of generations varied, both however are always in the hundreds.

6 RESULTS

Below in Tables 1–7 are the results of some of our simulation sets. Each line in the tables stands for an entire set of 100 simulations. We look at the differences between sets utilizing different bucketizing approaches and those using none. In some cases where the effect is not self-evident we ran a two-tailed t-test to test for the statistical significance of the different in means between sets. We set a harsh criterion of $p < 0.01$ for significance.

6.1 Results Solving Tricky Keys Problems

First we want to look at solving some hard deceptive problems. Below in Table 1 we see the results from simulations on a problem domain of size 2^{80} with 625 global optima (keys) and 1 deceptive local optimum (trick key).

Table 2: Runs using 60-bit genomes subdivided into 3 20-bit segments. All simulations below use a population size of 600 and run for 400 generations.

Keys	Trick Keys	Run type	# of buckets	Mean best fitness	Solution found
10^3	1	Normal	-	56.52	29
10^3	1	Func	61	59.92	98
10^3	1	K-Means	40	59.87	96

Table 3: Runs using 40-bit genomes unsegmented and with no trick keys. All simulations below use a population size of 400 and run for 300 generations.

Run type	# of buckets	Mean best fitness	Solution found	mean # of solutions
Normal	-	40.0	100	1.0
Func	41	40.0	100	2.63
K-Means	40	40.0	100	6.04

Table 4: Runs using base 4 unsegmented genomes of length 30 with no trick keys. All simulations below use a population size of 400 and run for 300 generations.

Run type	# of buckets	Mean best fitness	Solution found	mean # of solutions
Normal	-	30.0	100	1.01
Func	60	30.0	100	2.2
K-Means	40	29.98	98	3.43

These results show that while the EA without bucketing fails to find an optimum, the simulations that use function bucketing and clustering based bucketing succeed in the task in 16 and 13 out of 100 simulations, respectively. The average best fitness achieved in a simulation using either bucketing technique is also higher. The difference in means between the two bucketing techniques is significant ($p < 0.01$).

The problem above is quite hard to solve with the resources allotted. Below in Table 2 we have results from another run with 60-bit genomes and a much higher density of keys with 1000 global optima in the search space.

In this problem we see again that both bucketing strategies improve performance relative to the normal simulation set. In these problems, however, the bucketing runs can be counted on to quite reliably find an optimal solution. Function bucketing does a little better but the effect is not statistically significant.

6.2 Results Finding Multiple Keys

We are interested in finding out whether the use of bucketing methods can help an EA find more global optima than conventional EAs. We first look at problem spaces with multiple keys and no trick keys. Tables 3 and 4 show simulation sets running on problems with 10 keys and without any trick keys. Table 3 shows results of simulations on 40-bit problems. Table 3 shows results of simulations on problems with base 4 keys of length 30.

Table 5: Data about number of solutions found in experiment 60-bit experiment described in Table 2.

Run type	Mean best fitness	Solution found	mean # of solutions	mean # found only
Normal	56.52	29	0.32	1.10
Func	59.92	98	2.76	2.82
K-Means	59.87	96	3.37	3.51

Unsurprisingly in both these experiments an optimal point is found almost every time. Another feature that is evident here is that clustering based bucketing finds more solutions on average than function bucketing, that in turn finds more solutions than the normal runs. The normal runs typically find 1 solution. All these differences in number of solutions found for both experimental setups are statistically significant ($p < 0.00001$).

Next start We look again at our simulation from Table 2 but this time we are also interested in the number of keys found. Table 5 contains that information.

Here we see the same effects as before. Ignoring the normal EA that mostly fails to solve the problem, clustering-based bucketing finds more solutions than function bucketing. Even though the effect seems not as strong it is still statistically significant ($p < 0.01$).

Figure 2 shows how the results from Table 3 compare over the two simulation sets using the two different bucketing approaches. The simulations in each set are sorted according to number of solutions found.

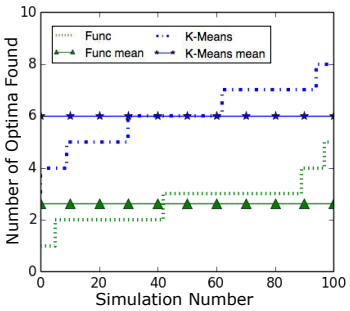

Figure 2: Comparison of number of global optima found by the different simulations using the function and clustering based bucketing.

In order to check how this effect transfers to bigger problem domains we ran experiments in domains with a bigger search space on deceptive problems of various difficulty. Table 6 shows results from experiments on 10 segment 100-bit genome domains (a problem space of size 2^{100}).

We can see in Table 6 that simulations using bucketing techniques tend to do better than normal simulations in the same block. Clustering based bucketing does not consistently find a higher number of solutions than function bucketing in these simulation sets and in fact in three of the four blocks it finds less. Function bucketing finds more solutions on average in the first block and this difference is statistically significant ($p < 0.01$). In the second block function bucketing does better as well but this difference does not pass our significance criterion ($p = 0.023813$).

The parameter setups in the last two blocks make the problem very hard for all approaches. We ran additional simulations sets for these problem spaces with a larger population size. Table 7 shows the results of these simulation sets.

In the simulation sets shown in Table 7 function bucketing seems to be doing a lot better than clustering based bucketing. with higher best fitness averages and a much higher ratio of finding an optimal solution in both blocks. The fitness advantage in the first block, though it is less than 0.6 is significant ($p < 0.001$). The higher fitness difference in the second block is also significant ($p < 0.00001$).

6.3 Dynamic Function Bucketing

As results show, on some problem spaces clustering based bucketing finds more local optima than function bucketing. When we tried to understand this strange phenomenon we came to wander if the reason was function bucketing is static while the clustering based bucketing is dynamic.

With this in mind we set out to make function bucketing dynamic in the hopes that of coming up with a new approach that will hopefully exhibit the advantages of both our approaches. The solution we came up with was using an alternating mask. On each generation the algorithm chooses a random mask. This mask added to each genome before the bucketing function is applied to it, thus changing the way the function divides the problem space into buckets. However when we ran simulations with our new approach we found that there was practically no difference in the results between the dynamic and static function bucketing techniques. For brevity these results are omitted from this work.

7 CONCLUDING REMARKS

In this work was we explored ways to manipulate the selection process in an EA in order to maintain better population diversity. We designated two criteria for success in achieving and maintaining effective population diversity in relation to a standard EA:

(1) Find global optimum or reach better fitness results on deceptive problem domains.
(2) Find more optimal solutions in problem domains with several optima.

We presented TrickyKeys, a new family of fast and easy to code deceptive optimization problems and suggested diversity maintenance techniques built around the our concept of bucket-based selection. We examined two approaches to bucketing, one based on locality sensitive functions and the other based on a clustering algorithm. Both approaches were successful in improving results over standard EAs and as experiments show both approaches satisfy both criteria defined for maintaining effective population diversity.

We saw differences in the performance of the two approaches to bucketing as implemented in this work. In some cases clustering-based bucketing has proven better at finding more solutions. In some of the harder problems function bucketing finds a global optimum with higher frequency and reaches better mean best fitness.

Table 6: Runs using 100-bit genomes subdivided into 10 10-bit segments. All simulations below use a population size of 300 and run for 200 generations. Experiment sets divided into four blocks of increasing problem difficulty.

Keys	Trick Keys	Run type	# of buckets	Mean best fitness	Solution found	mean # of solutions	mean # found only
10^{10}	1	Normal	-	99.76	89	4.0	4.49
10^{10}	1	Func	30	100.0	100	9.0	9.0
10^{10}	1	K-Means	15	100.0	100	7.92	7.92
4^{10}	1	Normal	-	98.0	37	0.68	1.84
4^{10}	1	Func	30	99.42	73	1.56	2.14
4^{10}	1	K-Means	15	99.16	61	1.11	1.82
2^{10}	1	Normal	-	94.87	2	0.02	1.0
2^{10}	1	Func	30	96.99	14	0.16	1.14
2^{10}	1	K-Means	15	97.32	17	0.18	1.06
2^{10}	2^{10}	Normal	-	93.42	4	0.04	1.0
2^{10}	2^{10}	Func	30	95.88	6	0.07	1.17
2^{10}	2^{10}	K-Means	15	94.87	3	0.03	1.0

Table 7: Runs using 100-bit genomes subdivided into 10-bit segments. Repeat of hardest sets from previous table with a bigger population and more generations. All simulations below use a population size of 600 and run for 500 generations

Keys	Trick Keys	Run type	# of buckets	Mean best fitness	Solution found	mean # of solutions	mean # found only
2^{10}	1	Normal	-	96.39	15	0.22	1.47
2^{10}	1	Func	60	99.73	86	2.03	2.36
2^{10}	1	K-Means	30	99.14	65	1.74	2.68
2^{10}	2^{10}	Normal	-	94.74	3	0.03	1.0
2^{10}	2^{10}	Func	60	99.56	76	1.5	1.97
2^{10}	2^{10}	K-Means	30	98.21	32	0.6	1.85

Our approach is still in its infancy but we believe it shows great potential. There is still much to be done and many possibilities to be explored. We plan to look into other methods of making bucketing dynamic, including changes in bucket sizes in mid-run. Another obvious avenue is to try bucketing techniques in other problem domains beyond the family of synthetic domains used here. One great challenge might be to use bucketing techniques in order to maintain phenotypic diversity in a problem space with generative or developmental encoding.

REFERENCES

[1] Amit Benbassat and Avishai Henik. 2016. Replicating the Stroop Effect Using a Developmental Spatial Neuroevolution System. In *International Conference on Parallel Problem Solving from Nature*. Springer International Publishing, Springer International Publishing, Edinburgh, UK, 602–612.

[2] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. 2013. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 35.

[3] David B D'Ambrosio and Kenneth O Stanley. 2007. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, New York, NY, USA, 974–981.

[4] Edward W Forgy. 1965. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics* 21 (1965), 768–769.

[5] David Goldberg, Kalyanmoy Deb, and Bradley Korb. 1989. Messy genetic algorithms: Motivation, analysis, and first results. *Complex systems* 3 (1989), 493–530.

[6] David E Goldberg, Kalyanmoy Deb, Hillol Kargupta, and Georges R Harik. 1993. RapidAccurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann, San Francisco, CA, USA, 56–64.

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

[8] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[9] James MacQueen and others. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA., University of California Press, Berkeley, CA, 281–297.

[10] Samir W Mahfoud. 1995. Niching methods for genetic algorithms. *Urbana* 51, 95001 (1995), 62–94.

[11] WN Martin, Jens Lienig, and James P Cohoon. 1997. C6. 3 Island (migration) models: evolutionary algorithms based on punctuated equilibria. *B ack et al. BFM97], Seiten C* 6 (1997), 101–fi?!124.

[12] Alan Pétrowski. 1996. A clearing procedure as a niching method for genetic algorithms. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*. IEEE, Nagoya, Japan, 798–803.

[13] Bruno Sareni and Laurent Krahenbuhl. 1998. Fitness sharing and niching methods revisited. *IEEE transactions on Evolutionary Computation* 2, 3 (1998), 97–106.

[14] Giovanni Squillero and Alberto Tonda. 2016. Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization. *Information Sciences* 329 (2016), 782–799.

[15] Kenneth O Stanley. 2007. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines* 8, 2 (2007), 131–162.

[16] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.